Genero Business Development Language User Guide Version 2.11

www.4js.com

Note: This documentation is for Genero 2.11. See the corresponding on-line documentation at the Web site **http://www.4js.com/online_documentation** for the latest updates. Please contact your nearest support center if you encounter problems or errors in the on-line documentation.

Table Of Contents

General

Introduction: BDL Concepts 1
Documentation Conventions 15
Language Features 18
The Dynamic User Interface 24
Installation and Setup 35
Tools and Components 48
Frequently Asked Questions 58
New Features of the Language 66
1.3x Migration Issues 99
2.0x Migration Issues 110
2.1x Migration Issues 120

Language Basics

Data Types 123
Literals 140
Operators 144
Expressions 174
Exceptions 181
Variables 188
Constants 199
Records 202
Arrays 204
User Types 210
Data Conversions 212
Built-in Classes 214

Applications

Compiling Programs 219
Programs 226
Database Schema Files 250
Globals 259
Flow Control 262
Functions 276
Reports 279
Localization 316
Localized Strings 326

Library

Built-in Functions 334
Utility Functions 371
Windows DDE Support 385
XML Utilities 396

SQL Management

Database Connections....397
Database Transactions....423
Static SQL Statements....430
Dynamic SQL Management....440
Database Result Set Processing (Cursor)....447
SQL Positioned Updates....460
SQL Insert Cursors....465
I/O SQL Instructions....473
SQL Programming....480

User Interface

The Interaction Model....525
Using Windows and Forms....535
Action Defaults....549
Presentation Styles....555
Form Specification Files....578
Form Specification File Attributes....649
Form Rendering....704
Menus....717
Displaying Data to Forms....726
Record Input....732
Array Display....749
Array Input....763
Query By Example....792
Multiple Dialogs....806
Prompt for Values....875
Displaying Messages....881
Toolbars....885
Topmenus....891
StartMenus....898
Canvas....903
Message Files....908
MDI Windows....911
Front End Functions....914
Front End Protocol....919

Built-in Classes

The Application class....927
The Channel class....930
The StringBuffer class....939
The StringTokenizer class....944
The TypeInfo class....947
The Interface class....949
The Window class....955
The Form class....959

The Dialog class 965
The ComboBox class 980
The DomDocument class 987
The DomNode class 990
The NodeList class 997
The SaxAttributes class 999
The SaxDocumentHandler class 1002
The XmlReader class 1006
The XmlWriter class 1009

Miscellaneous

Environment Variables 1013
The FGLPROFILE configuration file 1030
The Debugger 1035
The Profiler 1063
Optimization 1067
The Preprocessor 1073
File Extensions 1084
Error Messages 1085
General Terms used in this documentation 1145

BDL Tutorial

Genero BDL Tutorial Summary 1147
Tutorial Chapters 1148
Tutorial Chapter 1: Overview 1151
Tutorial Chapter 2: Using BDL 1156
Tutorial Chapter 3: Displaying Data (Windows/Forms) 1164
Tutorial Chapter 4: Query by Example 1177
Tutorial Chapter 5: Enhancing the Form 1197
Tutorial Chapter 6: Add/Update/Delete 1209
Tutorial Chapter 7: Array Display 1225
Tutorial Chapter 8: Array Input 1238
Tutorial Chapter 9: Reports 1251
Tutorial Chapter 10: Localization 1263
Tutorial Chapter 11: Master/Detail 1272
Tutorial Chapter 12: Changing the User Interface Dynamically 1296

ODI Adaptation Guides

ODI Adaptation Guide For Genero db 3.6x, 3.8x 1313
ODI Adaptation Guide For DB2 UDB 7.x, 8.x, 9x 1347
ODI Adaptation Guide For Oracle 8.x, 9.x, 10.x, 11.x 1388
ODI Adaptation Guide For SQL Server 2000, 2005, 2008 1435
ODI Adaptation Guide For PostgreSQL 8.0.2, 8.1.x, 8.2.x, 8.3.x 1472
ODI Adaptation Guide For MySQL 4.1.x, 5.0.x, 5.1.x 1498
ODI Adaptation Guide For Sybase ASA 8.x 1519

Standard Extensions

File Management Class 1547
Mathematical functions Class 1569

User Extensions

Implementing C-Extensions 1577
Genero FESQLC 1644

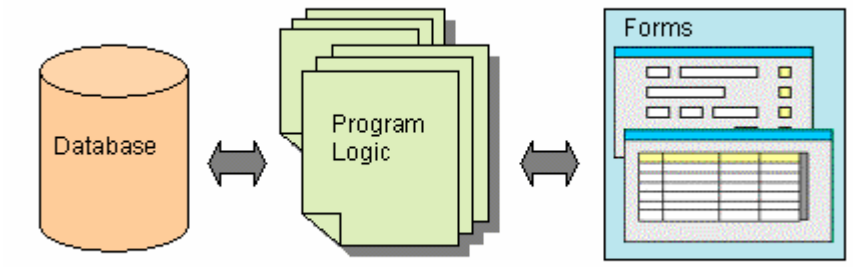
Introduction: BDL Concepts

Summary:

- Overview
- The Language
- Forms
- User Interface
- Compiling a BDL Application
- Deploying a BDL Application
- Resources for Programmers

Overview

You typically use Genero to build an *interactive database application*, a program that handles the interaction between a user and a database. The database schema that organizes data into relational tables gives shape to one side of the program. The needs of your user shape the other side. You write the program logic that bridges the gap between them.



An important feature of Genero BDL is the ease with which you can design applications that allow the user to access and modify data in a database. The Genero BDL language contains a set of SQL statements to manipulate the database, and interactive instructions that provide simple record input, read-only list handling, updateable list handling, and query by example (to search the database) using forms to facilitate interaction.

Genero BDL is compiled to *p-code*, which can be interpreted on different platforms by the *Dynamic Virtual Machine* (the Runtime system).

Separation of Business Logic and User Interface

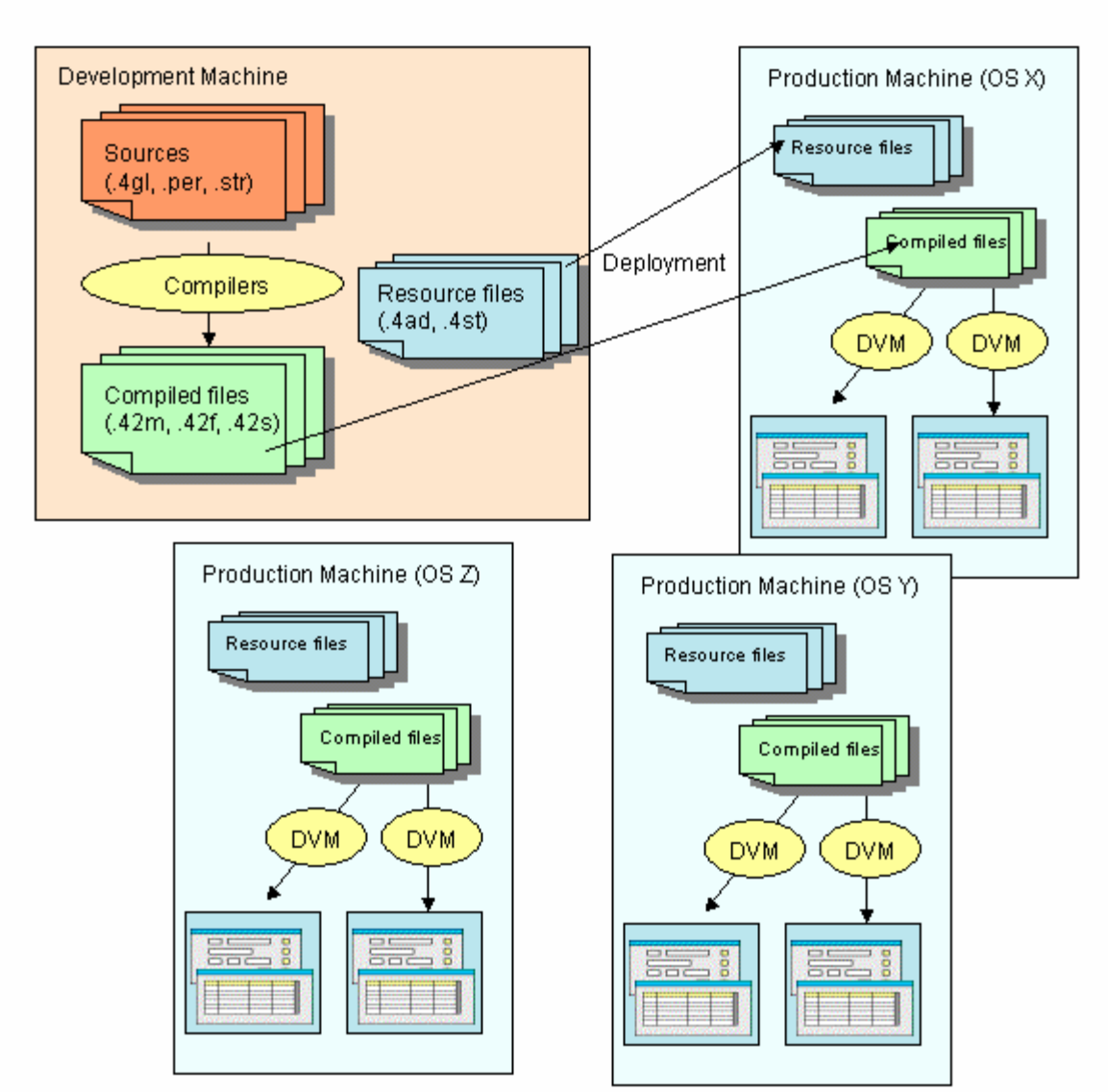
Genero separates business logic and the user interface to provide maximum flexibility:

- The *business logic* is written in text files (.4gl source code modules). High-level interactive instructions let you write a form controller in a few lines of code.
- *Forms* for the user interface are designed in a simple-to-understand and simple-to-read form definition syntax..
- *Action views* (buttons, menu items, toolbar icons) in the form definition can trigger *actions* defined in the business logic.
- Compiling a form definition file translates it into XML. The *XML-based presentation layer* ensures that user interface development is completely separated from deployment.
- The user interface can be manipulated at runtime, as a tree of objects called the *Abstract User Interface* (*AUI*).

Portability - write once, deploy anywhere

Genero provides the ability to support different kinds of display devices using the same source code. One production release supports all major versions of Unix, Linux, Windows NT/2000/XP and Mac OS X. The same application can be displayed with a graphical device (GUI mode) as well as on a simple dumb terminal (TUI mode).

A single code stream can be written to support HTML, Java, Windows, X.11, WML, MacIntosh OS X and ASCII interfaces simultaneously.



Reports

You can easily design and generate Reports. The output from a report can be formatted so that the eye of the reader can easily pick out the important facts. Page headers and footers, with page numbers, can be defined. Data can be grouped together, with group totals and subtotals shown. The output from a report can be sent to the screen, to a printer, to a file, or (through a pipe) to another program, and report output can even be redirected to an SAX filter in order to write XML.

Internationalization

Genero BDL supports multi-byte character sets by using the POSIX standard functions of the C library. Genero BDL uses BYTE-semantics to specify the length of a character string (i.e. CHAR(10) means 10 bytes). You must make sure that the database client

locale matches the runtime system locale. For more details about internationalization support, see Localization.

The *Localized Strings* feature allows you to customize your application for specific subsets of your user population, whether it is for a particular language or a particular business segment.

User Extensions

When the standard Genero built-in functions and classes are not sufficient, you can write your own plug-ins by using the Dynamic C Extensions. This allows you to implement specific function libraries in C, which can be called from the BDL modules. Typical User Extensions interface with C libraries to drive specific devices, such as barcode scanners or biometric identification devices.

The Language

Genero BDL is a high-level, fourth generation language with an open, readable syntax that encourages good individual or group programming style. You write your program logic in text files, or *program source modules*, which are compiled and linked into programs that can be executed by the Runtime system. Programs are easily enhanced and extended. This makes it easy for programmers to become productive quickly, no matter what programming languages they know. See Programs, Flow Control, Functions for additional information.

Database access

- A set of SQL statements are included as part of the language syntax and can be used directly in the source code, as a normal procedural instruction. The Static SQL Statements are parsed and validated at compile time. At runtime, these SQL statements are automatically prepared and executed by the runtime system. Program variables are detected by the compiler and handled as SQL parameters.
- Dynamic SQL management allows you to execute any SQL statement that is valid for your database version, in addition to those that are included as part of the language. The statement can be hard coded or created at runtime, with or without SQL parameters, returning or not returning a result set.
- Through the native drivers of the Open Database Interface, the same Genero program can open database connections to any of the supported databases.

For additional information, see SQL Programming.

Interactive Statements

Writing the code for interactive database applications has been simplified for you in Genero BDL; single statements automatically compile into the lines of program code

required for the common tasks associated with such applications. These interactive statements allow the program to respond to user input.

The DIALOG instruction allows parts of a form that have different functionality to be handled simultaneously.

Displaying data to the user

In Genero, programs manipulate Window and Form objects to define display areas for interactive statements within your program. The Abstract User Interface (AUI) tree contains a definition of these objects. You can open as many windows and forms as needed, subject only to the limits of memory and the maximum number of open files on the platform you are using.

The OPEN WINDOW statement creates and opens a new window on the user's screen. The runtime system maintains a stack of all open windows. When you execute this statement to open a new window, it is added to the window stack and becomes the current window.

You can modify the window stack with the CURRENT WINDOW and CLOSE WINDOW statements.

With the OPEN WINDOW ... WITH FORM statement you open a window on the screen, load a compiled form from disk into memory, and make it ready for use.

The interactive DISPLAY statement allows you to display program variable data in the fields of a form, for example, and then turn control over to the user for his subsequent action.

The interactive DISPLAY ARRAY and DIALOG statements allow the user to view the contents of an array of records, scrolling the records on the screen.

Allowing the user to enter and change data

The INPUT statement is an interactive statement (dialog) that enables the fields in a form for input, waits while the user types data into the fields, and proceeds after the user accepts or cancels the dialog. If the user accepts the dialog, the input that is automatically assigned to program variables can be used by your program to insert rows or change rows in a database, for example.

The interactive INPUT ARRAY statement allows the user to alter the contents of records in a screen array, and to insert and delete records. Your program can control and monitor these changes.

The DIALOG statement allows both INPUT and INPUT ARRAY functionality.

Allowing the user to Search a Database (Query by Example)

Genero BDL lets you take input from the user in more than one way; instead of literal values for the program to process, your user can enter search criteria for a query. The interactive CONSTRUCT and DIALOG statements allow the user to enter a value or a range of values for one or several form fields, and your program looks up the database rows that satisfy the requirements. You provide a single program variable to hold the result of the CONSTRUCT statement. Your program can combine this Boolean expression string with other text, to form a complete SELECT statement to fetch the desired database rows.

What you do with the fetched rows depends on the specific application. Often the reason is to select rows to be viewed by the user. In this case, the program could display each row individually in a form or grouped in a screen array; or, you might choose a set of rows for a report or a set of rows to be deleted or updated.

Responding to actions by the user

You can define a program routine (set of instructions) that is triggered by the user's actions. The actions can be displayed to the user as *action views* - buttons, toolbars, or pull-down menus in the application window. When the user makes a selection, the corresponding action is executed.

The interactive MENU statement can be used to define the list of actions that can be triggered. Or, the ON ACTION clause of interactive instructions, such as INPUT, INPUT ARRAY, CONSTRUCT, DIALOG, and DISPLAY ARRAY, can be used to specify the program routine to be executed for a given action.

Common actions, such as **accept** (dialog validation) and **cancel** (dialog cancellation), are already pre-defined for you in accordance with the interactive instruction.

Action Defaults allow you to define default decoration attributes (text, image) and functional attributes (accelerator keys) for the graphical objects associated with actions.

Predefined functions and classes

The BDL language provides built-in functions to perform many basic tasks, as well as built-in classes that can be used to manipulate the user interface. Dynamic C Extension libraries are part of the standard package; see File Manipulation functions and Mathematical functions.

Forms

The end-user of a program does not know about the database schema or your carefully designed program logic. As the user sees it, the application screens, and the menus that invoke them, are the application. The arrangement of fields, labels, and other form objects, and the behavior of the form as the user presses different keys or buttons and selects different menu options, create the personality of the program.

You can define the application screens, or forms, in text-based *Form Specification Files* (.per) These form files are translated by the Form Compiler to produce the *Runtime Form Files* (.42f) that are deployed in production environments. Since form files are separate from the other parts of your program, the *Runtime Form Files* can be used with different programs.

Unlike compiled program files, the translated *Runtime Form Files* are XML documents that describe the form elements, enabling portability across display devices. The XML file can also be written directly, or it can be generated or modified from your program code using the methods provided with Genero Built-in Classes.

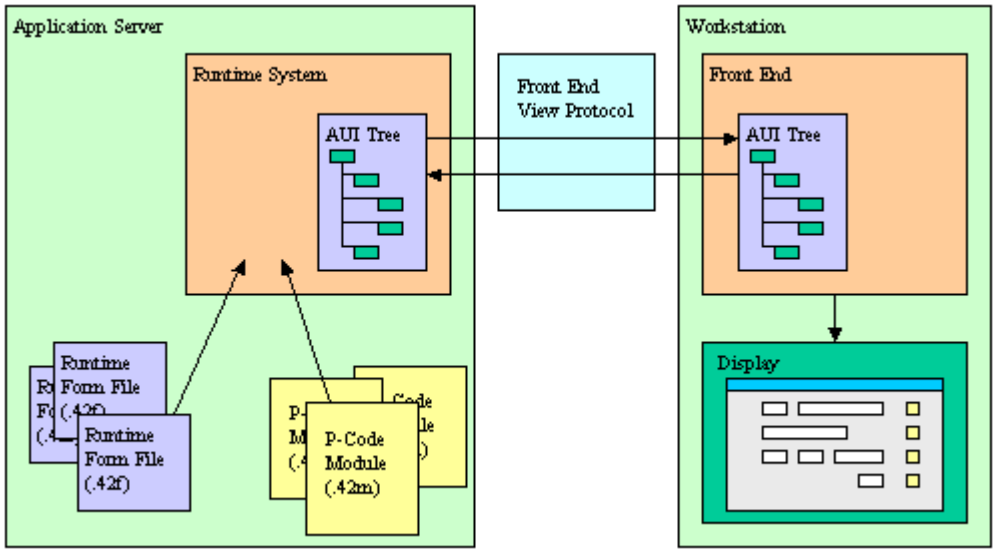
You can design your form to group objects in horizontal and vertical boxes, display it as a folder of pages, and use menus, toolbars, or buttons to trigger actions. You can associate an array of data in the program with an array of fields on the form (screen-array), so the user can see multiple rows of data on the screen. In addition to the form fields and screen records, your form can contain objects such as progressbars, checkboxes, radiogroups, comboboxes, and images. See Form Specification Files for additional information.

The appearance of form objects can be standardized using Presentation Styles and Action Defaults.

Different parts of the form can be handled simultaneously using the BDL instruction DIALOG.

The User Interface

What happens when a user executes a Genero BDL application? The Genero Runtime System creates the Abstract User Interface tree, and the Genero Front End makes this abstract tree visible on the Front End Client, such as the Genero Desktop Client, Genero Web Client, and Genero Java Client. When a user interaction statement takes control of the application, the copy of the tree on the Front End is automatically synchronized with the Runtime system tree by the *Front End Protocol*, an internal protocol used by the Runtime System.



The Genero BDL language provides Built-in Classes that implement methods to manage the objects on the user's screen. Methods can be invoked by passing parameters and/or returning values, allowing the User Interface to be modified completely at runtime.

Default XML files describe the appearance (decoration) of some of the graphic objects on the user's screen. These files may be customized, or replaced with your own versions.

- **default.4ad** - default decoration and accelerators for action views
- **default.4st** - default definition of presentation style attributes

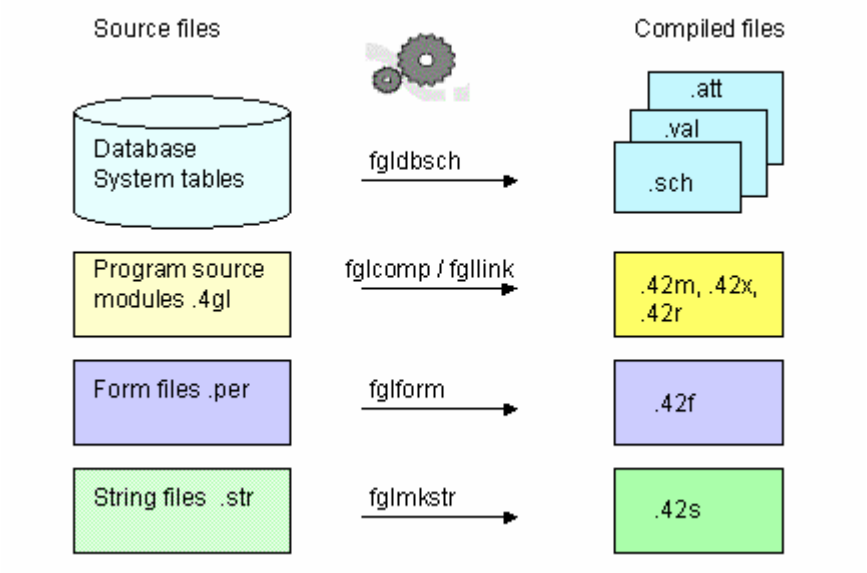
A special StartMenu, used to start different programs on the application server where the runtime system executes, can be defined in an XML file with the extension **.4sm**.

A set of XML Utilities are provided to allow you to create and modify XML documents.

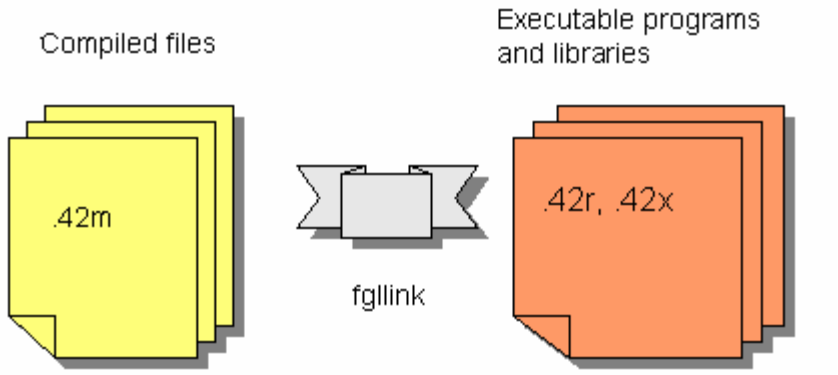
Compiling a Genero BDL Application

Your Genero BDL program can consist of a single source code module, but generally you will have multiple modules as well as form specification files and perhaps localized string files. Database schema files are also required, if you have defined program data types and variables in the terms of an existing database column or table by using the DEFINE ... LIKE statement.

The tools that Genero provides to compile the various files that will make up your application, and the file extensions of the source code and corresponding compiled files, are listed below, and an explanation follows:



The compiled source code modules can be linked into a program that can be executed by the Runtime System, or into a library that can be linked into other programs.



Database Schema Files - tool fgldbsch

Database Schema Files are used during program compilation to define data types, default values, display attributes and validation rules for form fields and program variables. You must generate database schema files each time the database structure changes, before compiling any other parts of your application.

The SCHEMA statement in program source files and form files identifies the database schema file to be used. The FGLDBPATH environment variable can be used to define a list of directories where the compiler can find database schema files.

The Schema Extractor (fgldbsch) is the tool provided to generate the database schema files from a real database. Database schema files are generated from the system tables.

Example:

```
fgldbsch -db <database_name>
```

It is important that the schema file of the development database corresponds to the production database; otherwise, the elements defined in the compiled version of your modules and forms will not match the table structures of the production database.

The primary file produced by the utility is:

***<database>*.sch** - the file containing the data type definition of all columns selected during schema extraction.

Program Source Modules - tools fglcomp, fgllink, fgl2p

Genero BDL provides its own source code *compiler,* which generates hardware-independent pseudo-machine code (P-code). This code is not directly executable by the operating system: it is interpreted by the Genero BDL *runtime system* (a program named **fglrun**, and called "runner").

The compiled P-code modules are binary. The files are not printable or editable as text.

- Tool fglcomp - Module compiler

  This tool compiles a program source module into a p-code version. The compiled module has an extension of **.42m**.

  If a compilation error occurs, the text file <***filename*>.err** flagging the errors is created. You can display directly the compilation errors by using the -M option.

  Example:

  ```
  fglcomp <modulename>.4gl
  ```

  Running the program:

  ```
  fglrun <modulename>.42m
  ```

  If your program consists of more than one module, the modules must be linked together prior to execution, using the **fgllink** tool.

- Tool fgllink - Module linker

  This tool assembles multiple p-code modules compiled with fglcomp into a single **.42r** program or a **.42x** library.

Example to create a program:

```
fgllink -o <programname>.42r <module1name>.42m <module2name>.42m
...
```

Running the program:

```
fglrun <programname>.42r
```

Example to create a library that can be linked into other programs:

```
fgllink -o <libraryname>.42x <module1name>.42m <module2name>.42m
...
```

- Tool fgl2p - As a convenience, **fgl2p** (the Program compiler) is provided to create programs or libraries in one command line. It uses the **fglcomp** and the **fgllink** tools to compile and link modules together. If compilation of any of the modules fails, the file <*modulename*>.err is created.

  Example to create a program:

  ```
  fgl2p -o <programname>.42r <module1name>.42m
  <module2name>.42m  ...
  ```

  Example to create a library:

  ```
  fgl2p -o <libraryname>.42x <module1name>.42m
  <module2name>.42m  ...
  ```

Form Specification Files - tool fglform

Form specification files are used by the Form Compiler (fglform) to produce the *Runtime Form Files* that are deployed in production environments. The form files have an extension of **.per** . Unlike compiled program files, the generated Runtime Form File is an XML document that describes the form elements. The Runtime Form Files have an extension of **.42f**.

If a compilation error occurs, the text file **<*filename*>.err** flagging the errors is created. You can directly display the compilation error by using the -M option.

Example:

```
fglform <formname>.per
```

C-like Source Preprocessor

The fgl preprocessor can be used with the 4gl compiler and the form compiler to transform your sources before compilation, based on preprocessor directives. It allows

you to include other files, to define macros that will be expanded when used in the source, and to compile conditionally.

See The FGL Preprocessor for details.

Localized String Files - tool fglmkstr

Source String Files (<*filename*>.str) containing the text of the Localized Strings that are used in your application must be compiled to binary files in order to be used at runtime. By default the runtime system expects that the Source String file will have the same filename as your program. The compiled file has an extension of **.42s**.

Example:

```
fglmkstr <filename>.str
```

As an assistance in creating the *Source String Files*, the **-m** option of the fglcomp and fglform tools can be used to extract all the localized strings that are used in your source code modules and forms.

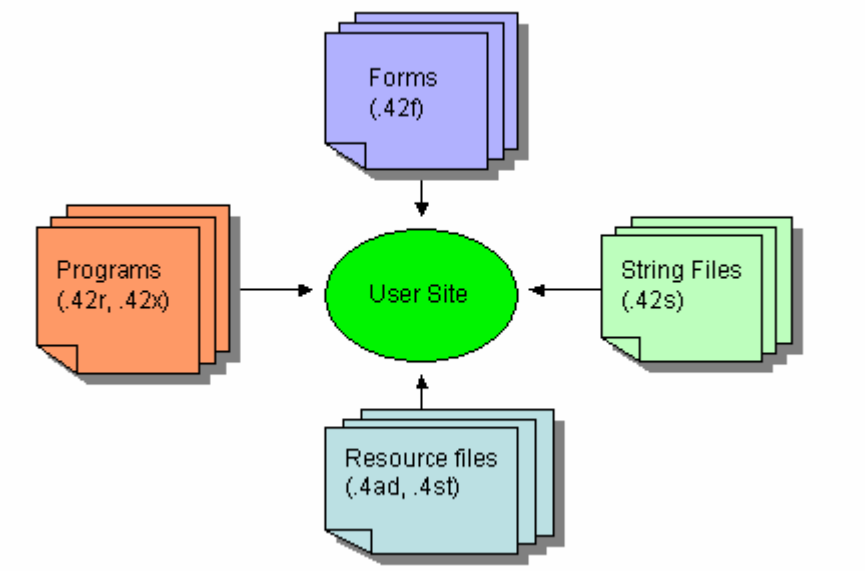
Example:

```
fglcomp -m <modulename>.4gl
```

The output file will be **<***modulename***>**.str . This extracted file can be edited to assign text to the localized strings, and combined with other extracted files into a single <*programname*>.str file.

Note: Windows or Unix-based *makefiles* may be used to automate the process of compiling and linking programs.

Deploying a Genero BDL Application



The following program files must be deployed at the user site:

- **.42r**, **.42x, .42m** - Executable programs and libraries, compiled modules
- **.42f** - Runtime Form Files
- **.42s** - compiled Localized String Files, if used in your applications
- **.4sm** - your custom Start Menu XML file, if created
- **.4ad**, **.4st** - these default XML files, provided with Genero, must be distributed with the runtime system files; if you have customized these files, or created your own versions, your versions must be deployed instead.

Front-end clients, such as Genero Desktop Client and Genero Web Client, make the application available to the user.

The FGLPROFILE configuration file can be used to change the behavior of programs, and environment variables can be set for Genero BDL.

Resources for Programmers

Documentation

This Business Development Language Manual (User Guide) is a complete guide to Genero BDL language features, with explanations and sample code:

- **Genero BDL Concepts** - the following pages may be especially helpful in understanding the concepts underlying Genero BDL:
  - Dynamic User Interface

    - Windows and Forms
    - Programs
    - Connections
    - Transactions
    - SQL Programming
    - Compiling Programs
    - Built-In Classes
    - Presentation Styles
- **The Business Development Language Tutorial** - contains examples of basic BDL program development, with sample code; see Summary
- **Migrating to Genero BDL** - the following pages may be especially helpful if you are migrating an application from an earlier product:
    - New Features
    - Frequently Asked Questions
    - Tools and Components
    - Layout - Form Rendering
    - 1.30 Migration Issues
    - 2.00 Migration Issues

  Note**:** The "Migration from I-4GL to Genero" section of the Four J's web site has suggestions, plans, checklists, and migration case studies.

- **Open Database Interface Adaptation guides** - these pages are provided for each relational database that Genero supports.

Code examples

- Short example programs in the **demo** subdirectory of the BDL software installation directory illustrate the use of Genero BDL features. These programs may be compiled and executed.
- The User Guide has extensive code samples.
- The Tutorial has code samples for each chapter with embedded comments.

Training

The Four J's web site lists the on-line and self-paced Genero training classes that are offered. Instructor-led classes are also available; please call your regional sales office for information.

Documentation Conventions

Summary:

- TUI Only Features
- De-supported Features
- Informix Specific Features
- Syntaxes
- Notes
- Warnings
- Tips
- Code Examples

TUI Only Features

TUI only features are marked with the red warning: **TUI Only!**

> **OPTIONS MENU LINE 3 TUI Only!**

Elements marked with this flag must only be used in programs designed for text-based terminals.

De-supported Features

Product features that are no longer supported are marked with the red warning: **De-supported!**

> **The WIDGET="BMP" attribute De-supported!**

Elements marked with this flag are no longer supported in the product.

Informix Specific Features

Features that are specific to Informix database servers are marked with the red warning: **Informix only!**

DATABASE dbname@dbserver Informix only!

Elements marked with this flag work only with Informix database servers, and are not recommended for multi-database programming.

Syntaxes

The term of '*syntax*' is global and indicates the way to use a product function. For example, it can be used to describe a language instruction or a system command:

Syntax:

```
CALL function ( [ parameter [,...] ] ) [ RETURNING variable
[,...] ]
```

Wildcard characters in syntax definitions are marked with an underscore:

Wildcards	Description
[ ]	Square braces indicate an optional element in the syntax.
{ | }	Curly braces indicate a list of possible elements separated by a pipe.
[...]	Indicates that the previous element can appear more than once.
[,...]	Previous element can appear more than once separated by a comma.

Notes

Notes hold a list of technical remarks about the product function:

Notes:

1. *identifier* is the name of the variable to be defined.
2. *datatype* can be any data type except complex types like TEXT or BYTE.
3. ...

Warnings

Warnings are important technical remarks, describing special behavior of the product function:

Warnings:

1. When a DATE, DATETIME or INTERVAL constant cannot be initialized correctly, it is set to NULL.
2. ...

Tips

Tips are hints to use the product function more efficiently:

Tips:

1. Do not include a NULL value in a Boolean expression.
2. ...

Code Examples

Code examples are written with line numbers and language syntax highlighting as follows:

Example 1:

```
01 MAIN
02   DEFINE a1 ARRAY[100] OF INTEGER,
03          a2 ARRAY[10,20] OF RECORD
04               id INTEGER,
05   ...
```

Language Features

Summary:

- Introduction
- Lettercase Insensitivity
- Whitespace Separators
- Quotation Marks
- Statement Terminator
- Character Set
- Comments
- Program Components
- SQL Support
- Identifiers
- Preprocessor Directives

Introduction

BDL is an English-like programming language designed for creating relational database applications.

The language includes high-level instructions to implement the user interface of the applications, generate reports, and execute SQL statements to communicate with database servers.

Lettercase Insensitivity

BDL is *case insensitive*, making no distinction between uppercase and lowercase letters, except within quoted strings. Use pairs of double (`"`) or single (`'`) quotation marks in the code to preserve the lettercase of character literals, filenames, and names of database entities.

You can mix uppercase and lowercase letters in the identifiers that you assign to language entities, but any uppercase letters in BDL identifiers are automatically shifted to lowercase during compilation.

Tips:

1. It is strongly recommended that you define a naming convention for your projects. For example, you can use underscore notation (`get_user_name`). If you plan to use the Java notation (`getUserName`), do not forget that BDL is case insensitive (`getusername` is the same identifier as `getUserName`).

Whitespace Separators

BDL is free-form, like C or Pascal, and generally ignores TAB characters, LINEFEED characters, comments, and extra blank spaces between statements or statement elements. You can freely use these whitespace characters to enhance the readability of your source code.

Blank (ASCII 32) characters act as delimiters in some contexts. Blank spaces must separate successive keywords or identifiers, but cannot appear within a keyword or identifier. Pairs of double (") or single (') quotation marks must delimit any character string that contains a blank (ASCII 32) or other whitespace character, such as LINEFEED or RETURN.

Quotation Marks

In BDL, string literals are delimited by single (') or double (") quotation marks:

```
'Valid character string'
"Another valid character string"
```

Do not mix double and single quotation marks as delimiters of the same string. For example, the following is not a valid character string:

```
'Not A valid character string"
```

In SQL statements, when accessing a non-Informix relational database, such as a DB2 database from IBM, double quotation marks might not be recognized as database object name delimiters. In the SQL language, the standard specifications recommend that you use single quotes for string literals and double quotes for database object identifiers like table or column names.

To include literal quotation marks within a quoted string, precede each literal quotation mark with the backslash (\), or else enclose the string between a pair of the opposite type of quotation marks:

```
01 MAIN
02   DISPLAY "Type 'Y' if you want to reformat your disk."
03   DISPLAY 'Type "Y" if you want to reformat your disk.'
04   DISPLAY 'Type \'Y\' if you want to reformat your disk.'
05 END MAIN
```

A string literal can be written on multiple lines. The compiler merges lines by removing the new-line character.

For more details, see String Literals.

Escape Symbols

The compiler treats a backslash (`\`) as the default escape symbol, and treats the immediately following symbol as a literal, rather than as having special significance. To specify anything that includes a literal backslash, enter double (`\\`) backslashes wherever a single backslash is required. Similarly, use `\\\\` to represent a literal double backslash.

For more details, see String Literals.

Statement Terminator

BDL requires no statement terminators, but you can use the semicolon (;) as a statement terminator in some cases, PREPARE and PRINT statements for example.

```
01 MAIN
02   DISPLAY "Hello, World"  DISPLAY "Hello, World"
03   DISPLAY "Hello, World"; DISPLAY "Hello, World"
04 END MAIN
```

Character Set

The language requires the ASCII character set, but also supports characters from the client locale in data values, identifiers, form specifications, and reports.

Comments

A comment is text in the source code to assist human readers, but which BDL ignores. (This meaning of *comment* is unrelated to the COMMENTS attribute in a form, or to the OPTIONS COMMENT LINE statement, both of which control on-screen text displays to assist users of the application.)

You can indicate comments in any of several ways:

- A comment can begin with the left-brace (`{`) and end with the right-brace (`}`) symbol. These can be on the same line or on different lines.
- The pound (`#`) symbol (sometimes called the "sharp symbol") can begin a comment that terminates at the end of the same line.
- You can use a pair of minus signs (`--`) to begin a comment that terminates at the end of the current line. (This comment indicator conforms to the ANSI standard for SQL.)

Warnings:

1. Within a quoted string, 4GL interprets comment indicators as literal characters, rather than as comment indicators.
2. You cannot use braces (`{ }`) to nest comments within comments.
3. Comments cannot appear in the SCREEN section of a form specification file.
4. The `#` symbol cannot indicate comments in an SQL statement block, nor in the text of a prepared statement.
5. You cannot specify consecutive minus signs (`--`) in arithmetic expressions, because BDL interprets what follows as a comment. Instead, use a blank space or parentheses to separate consecutive arithmetic minus signs.
6. The symbol that immediately follows the `--` comment indicator must not be the sharp (#) symbol, unless you intend to compile the same source file with the Informix 4GL product.

Tips:

1. For clarity and to simplify program maintenance, it is recommended that you document your code by including comments in your source files.
2. You can use comment indicators during program development to disable statements without deleting them from your source code modules.

Program Components

BDL programs are built from source code files with the language compiler, form compiler, and message compiler. Source code files can be:

- Form Specification Files (`.per`)
- Database Schema Files (`.sch`, `.att`, `.val`)
- Message Files (`.msg`)
- Source String Files (`.str`)
- Program Sources Files (`.4gl`)

In Form Specification Files, you define the layout of application screens. See Forms for more details.

The Database Schema Files describe the structure of the database tables. See Database Schema for more details.

The Message Files hold texts that can be loaded at runtime. Each text is identified by a number. See Message Files for more details.

The Localized Strings allow you to customize application strings, which are loaded automatically at runtime. Each string is identified by an identifier. See Localized Strings for more details.

In Program Source Files, you define the structure of the program with instruction blocks (like `MAIN`, `FUNCTION` or `REPORT`). The program starts from the `MAIN` block. The instruction blocks contain BDL instructions that are be executed by the runtime system in the order that they appear in the code. Program blocks cannot be nested, nor any program block divided among more than one source code module.

Some BDL instructions can include other instructions. Such instructions are called *compound statements*. Every compound statement of BDL supports the END keyword to mark the end of the compound statement construct within the source code module. Most compound statements also support the EXIT *statement* keywords, to transfer control of execution to the statement that follows the END *statement* keywords, where *statement* is the name of the compound statement. By definition, every compound statement can contain at least one statement block, a group of one or more consecutive SQL statements or other BDL statements. In the syntax diagram of a compound statement, a statement block always includes this element.

SQL Support

A limited syntax of SQL is supported directly by the BDL compiler, so you can write common SQL statements such as SELECT, INSERT, UPDATE or DELETE directly in your source code:

```
01 MAIN
02   DEFINE n INTEGER, s CHAR(20)
03   DATABASE stores
04   LET s = "Sansino"
05   SELECT COUNT(*) INTO n FROM customer WHERE custname = s
06   DISPLAY "Rows found: " || n
07 END MAIN
```

For SQL statements that have a syntax that is not supported directly by the compiler, the language provides SQL statement preparation from strings as in other languages.

```
01 MAIN
02   DEFINE txt CHAR(20)
03   DATABASE stores
04   LET txt = "SET DATE_FORMAT = YMD"
05   PREPARE sh FROM txt
06   EXECUTE sh
07 END MAIN
```

For more details about SQL statement preparation, see the Dynamic SQL Instructions.

Identifiers

A BDL identifier is a character string that is declared as the name of a program entity. In the default (U.S. English) locale, every 4GL identifier must conform to the following rules:

- It must include at least one character, without any limitation in size.
- Only ASCII letters, digits, and underscore (_) symbols are valid.
- Blanks, hyphens, and other non-alphanumeric characters are not allowed.
- The initial character must be a letter or an underscore.
- Identifiers are not case sensitive, so `my_Var` and `MY_vaR` both denote the same identifier.

Within non-English locales, however, BDL identifiers can include non-ASCII characters in identifiers, if those characters are defined in the code set of the locale that `CLIENT_LOCALE` specifies. In multibyte East Asian locales that support languages whose written form is not alphabet-based, such as Chinese, Japanese, or Korean, an identifier does not need to begin with a letter.

Preprocessor Directives

The language supports preprocessing instructions, which allow you to write macros and conditional compilation rules:

```
01 &include "myheader.4gl"
02 FUNCTION debug( msg )
03   DEFINE msg STRING
04 &ifdef DEBUG
05   DISPLAY msg
06 &endif
07 END FUNCTION
```

See The Preprocessor for more details.

The Dynamic User Interface

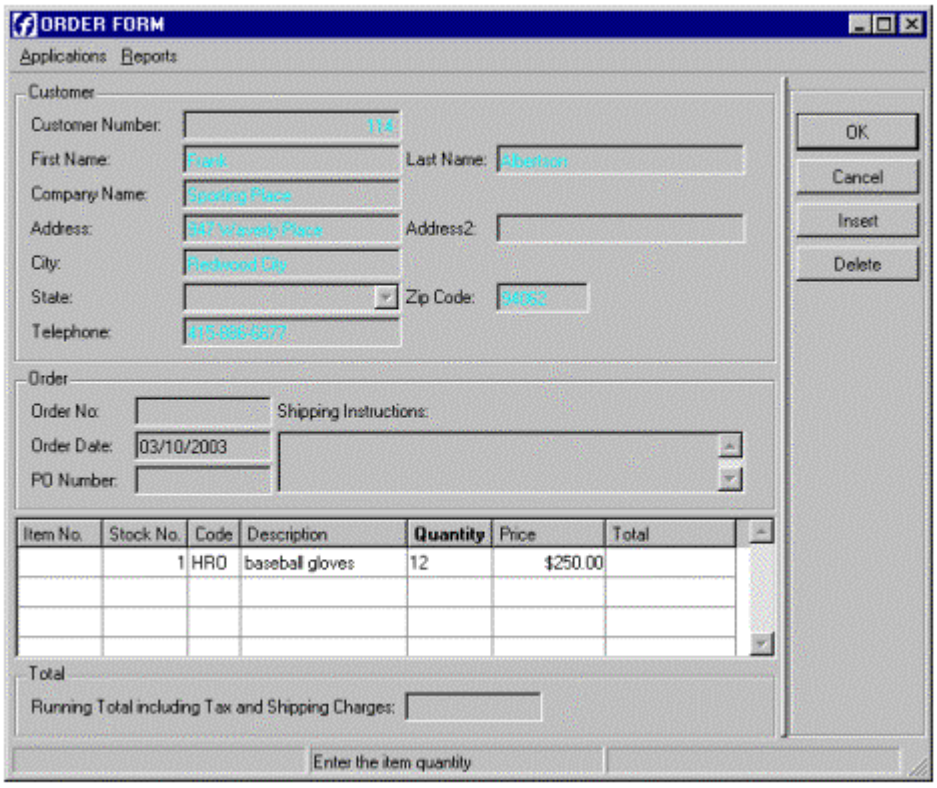
Summary:

- Graphical rendering
- The Dynamic User Interface
  - The concept
  - When is the front-end synchronized?
- Connecting to the front-end
  - Graphical and Text Mode
  - Defining the Target Front End
  - Front End Identification
  - Warning: Security Issue
  - Controlling Front End Connection
  - Front End Connection Lost
  - Front End Errors
- The Abstract User Interface
  - What does the Abstract User Interface tree contain?
  - Manipulating the Abstract User Interface
  - XML Node Type and Attribute Names
  - Actions in the Abstract user Interface tree
  - The Front End Protocol
- Special Features
  - Character Conversion Table
  - Automatic front-end startup

See also: Form Files, Windows and Forms, Interaction Model.

Graphical rendering

In Genero, the user interface is designed to provide a real graphical look and feel, compared to traditional Informix 4GL applications. However, graphical user interfaces and especially windows management is not compatible with the traditional 4GL user interface management, which was designed for character terminals. With the graphical interface of Genero, you can, for example, display windows as real movable and resizable windows, display labels with variable fonts, use toolbars and pull-down menus, or show error messages in a status bar. But this requires you to adapt the code and remove instructions like DISPLAY AT that make no sense in real GUI mode.



The Dynamic User Interface

The concept

The Dynamic User Interface (DUI) is a global concept for a new, open User Interface programming toolkit and deployment components, based on the usage of XML standards and built-in classes.

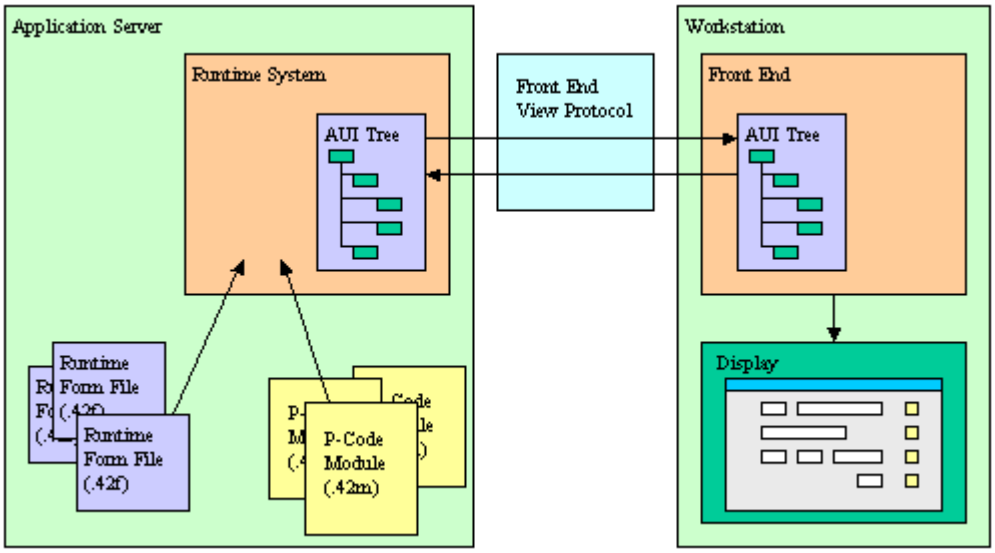
The purpose of the DUI is to support different kinds of display devices by using the same source code, introducing an abstract definition of the user interface that can be manipulated at runtime as a tree of user interface objects. This tree is called the Abstract User Interface.

The Runtime System is in charge of the Abstract User Interface tree and the Front End is in charge of making this abstract tree visible on the screen. The Front End gets a copy of that tree which is automatically synchronized by the runtime by using the Front End Protocol.

In development, application screens are defined by Form Specification Files. These files are used by the Form Compiler to produce the Runtime Form Files that can be deployed in production environments.

Architectural schema

The following schema describes the Dynamic User Interface concept, showing how the Abstract User Interface tree is shared by the Runtime System and the Front End:



When is the front-end synchronized?

The Abstract User Interface tree on the front-end is synchronized with the Runtime System AUI tree when a user interaction instruction takes the control. This means that the user will not see any display as long as the program is doing batch processing, until an interactive statement is reached.

For example, the following program shows nothing:

```
01 MAIN
02   DEFINE cnt INTEGER
03   OPEN WINDOW w WITH FORM "myform"
04   FOR cnt=1 TO 10
05     DISPLAY BY NAME cnt
06     SLEEP 1
07   END FOR
08 END MAIN
```

If you want to show something on the screen while the program is running in a batch procedure, you must force synchronization with the front-end, by calling the `refresh()` method of the Interface built-in class:

```
01 MAIN
02   DEFINE cnt INTEGER
03   OPEN WINDOW w WITH FORM "myform"
04   FOR cnt=1 TO 10
05     DISPLAY BY NAME cnt
06     CALL ui.Interface.refresh()   -- Sync the front-end!
```

```
07     SLEEP 1
08   END FOR
09 END MAIN
```

Connecting to the front-end

Graphical and Text Mode

By default, a Genero BDL application executes in *graphical mode* (GUI). However, you can run the applications in dumb terminals by using the text-based display, called *text mode* (TUI). To run the application in TUI mode, set the FGLGUI environment variable to zero.

Defining the Target Front-End

In GUI mode, when the first interactive instruction like MENU or INPUT is executed, the runtime system establishes a tcp connection to the front-end. The front-end acts as a graphical server for the runtime system.

On the runtime system side, the front-end is identified by the FGLSERVER environment variable. This variable defines the hostname of the machine where the front-end resides, and the number of the front-end instance to be used.

The syntax for FGLSERVER is **hostname[:servernum]**:

```
$ FGLSERVER=fox:1
$ fglrun myprog
```

The **servernum** parameter is a whole number that defines the instance of the front-end. It is actually defining a tcp port number, starting from 6400. For example, if **servernum** equals 2, the tcp port number used is 6402 (6400+2).

This is the standard/basic connection technique, but you can set up different types of configurations; for example, to have the front-end connect to an application server via ssh, to pass through firewalls over the internet. Refer to the front-end documentation for more details.

Front-End Identification

The front-end can open a terminal session on the application server to start a program from the user workstation. This is done by using a ssh, rlogin, or telnet terminal session. When the terminal session is open, the front-end sends a couple of shell commands to set environment variables like FGLSERVER before starting the Genero program to display the application on the front-end where the terminal session was initiated.

In this configuration, front-end identification takes place. The front-end identification prevents the display of application windows on a front-end that did not start the Genero application on the server. If the front-end was not identified, it would result in an important security problem, as anyone could run a fake application that could display on any front-end and ask for a password.

Warning (Security Issue): Front-end identification is achieved by setting two environment variables in the terminal session, which identify the front-end. The runtime system sends the first identifier back when connecting to the front-end, and the front-end sends the second id in the returning connection string. The Front-end checks the first id, and refuses the connection if that id does not correspond to the original id set in the terminal session. The runtime system checks the second id send by the front-end in the connection string, and refuses the connection if that id does not correspond to the environment variable set in the terminal session. There can be a security hole if users can overwrite the program or the shell script started by the front-end terminal session. It is then possible to change the front-end identification environment variables and FGLSERVER, in order to display the application on another workstation to read confidential data. As long as basic application users do not have read and write privileges on the program files, there is no risk. To make sure that program files on the server side are protected from basic users, create a special user on the server to manage the application program files, and give other users only read access to those files. As long as basic users cannot modify programs on the server side, there is no security issue.

Controlling Front-End Connection

If the front-end host machine is down or if its firewall drops connections for the port used by Genero, the program will stop with an error after a given timeout.

The connection timeout can be specified with the following FGLPROFILE entry:

```
gui.connection.timeout = seconds
```

The default timeout is 30 seconds.

Front-End Connection Lost

When the runtime system waits for a user action, but the end user does not do anything, the client sends a 'ping' event every 5 minutes to keep the tcp connection alive. This situation can happen if the user leaves the workstation for a while without closing the application.

If the client is not stopped properly (when killed by a system reboot, for example), the tcp connection is lost and the runtime system does not receive any more 'ping' events from the client. In this case, the runtime system waits for a specified time before it stops with fatal error **-8062**.

By default, the runtime system waits for 600 seconds (10 minutes).

You can configure this timeout with an FGLPROFILE entry:

```
gui.protocol.pingTimeout = 800
```

Warning: If you set this timeout to a value lower than the ping delay of the front-end, the program will stop with a fatal error after that timeout, even if the tcp connection is still alive. For example, with a front-end having a ping delay of 5 minutes, the minimum value for this parameter should be about 330 seconds (5 minutes + 30 seconds to make sure the client ping arrives).

Front-End Errors

When the Front End receives an invalid order, it stops the application. The Runtime System then stops and displays the following message:

```
Program stopped at 'xxx.4gl', line number yy.
FORMS statement error number -6313.
The UserInterface has been destroyed: <message>.
```

The following error messages can occur:

Message	Description
Application was terminated by user	The front-end has been stopped or the user has clicked on the "Terminate application" button.
Unexpected interface version sent by the runtime system	The runtime system and the front-end versions are not fully compatible.
The container '*container_name*' already exists	The same WCI container has been started twice.
The container '*container_name*' was destroyed	The parent WCI container has been stopped while some children are still running
The container '*container_name*' doesn't exist	The WCI parent of the current child doesn't exist.
Invalid AUI Tree: Multiple Start Menu nodes	The AUI Tree contains two Start Menu Nodes - *should not happen*.

The Abstract User Interface

The Abstract User Interface (AUI) is a DOM tree describing the objects of the User Interface of a Program at a given time. A copy of the AUI tree is held by both the Front End and the Runtime System. AUI Tree synchronization is automatically done by the Runtime System using the Front End Protocol. The programs can manipulate the AUI tree by using built-in classes and XML utilities.

What does the Abstract User Interface tree contain?

The Abstract User Interface defines a tree of objects organized by parent/child relationship. The different kinds of user interface objects are defined by attributes. The AUI tree can be serialized as text according to the XML standard notation.

The following example shows a part of an AUI tree defining a Toolbar serialized with the XML notation:

```
<ToolBar>
  <ToolBarItem name="f5" text="List" image="list" />
  <ToolBarSeparator/>
  <ToolBarItem name="Query" text="Query" image="search" />
  <ToolBarItem name="Add" text="Append" image="add" />
  ...
</ToolBar>
```

Manipulating the Abstract User Interface tree

The objects of the Abstract User Interface tree can be queried and modified at runtime with built-in classes like ui.Form, provided to manipulate form elements.

```
01 DEFINE w ui.Window
02 DEFINE f ui.Form
03 LET w = ui.Window.getCurrent()
04 LET f = w.getForm()
05 CALL f.setElementHidden("groupbox1",1)
```

In very special cases, you can also directly access the nodes of the AUI tree by using DOM API classes like DomDocument and DomNode. To get the user interface nodes at runtime, the language provides different kinds of API functions or methods, according to the context. For example, to get the root of the Abstract User Interface tree, call the ui.Interface.getRootNode() method. You can also get the current form node with ui.Form.getNode() or search for an element by name with the ui.Form.findNode() method.

XML Node Types and Attribute Names

By tradition BDL uses uppercase keywords, such as LABEL in form files, and the examples in this documentation reflect that convention. The BDL language itself is not case-sensitive. However, XML is case-sensitive, and by convention node types use uppercase/lowercase combinations to indicate word boundaries. In BDL, therefore, the nodes and attributes of an Abstract User Interface tree are handled as follows:

- **Node types** - the first letter of the node type is always capitalized. Subsequent letters are lower-case, unless the type consists of multiple words joined together.

In that case, the first letter of each of the multiple words is capitalized (the CamelCase convention). Examples: Label, FormField, DateEdit, Edit

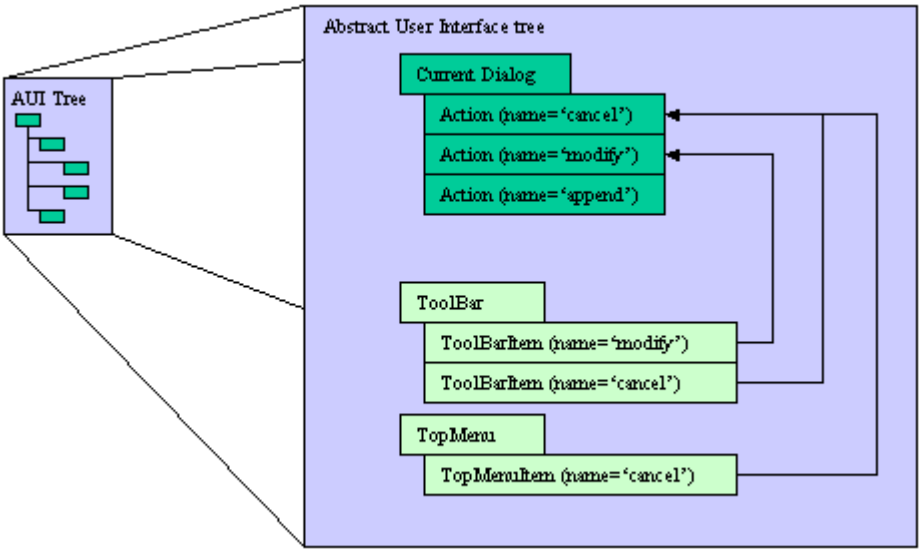
- **Attribute names** - the first letter of the name is always lower-case; subsequent letters are also lower-case, unless the name consists of multiple words joined together. In that case, the first letter of each subsequent word is capitalized (the Lower CamelCase convention). Examples: text, colName, width, tabIndex
- **Attribute values** - the values are enclosed in quotes, and BDL does not convert them..

Warning: If you reference Nodes or Attributes in your BDL code, you must always respect the naming conventions.

Actions in the Abstract User Interface tree

The Abstract User Interface identifies all possible actions that can be received by the current interactive instruction with a list of *Action* nodes. The list of possible actions are held by a *Dialog* node. An *Action* node is identified by the 'name' attribute and defines common properties such as the accelerator key, default image, and default text.

Interactive elements are bound to *Action* nodes by the 'name' attribute. For example, a Toolbar item (button) with the name 'cancel' is bound to the *Action* node having the name 'cancel', which in turn defines the accelerator key, the default text, and the default image for the button.



When an interactive element is used (such as a form field input, toolbar button click, or menu option selection), an *ActionEvent* node is sent to the runtime system. The name of the *ActionEvent* node identifies what *Action* occurred and the 'idRef' attribute indicates the source element of the action.

See also Front End Events for more details.

The Front End Protocol

The Front End Protocol (FEP) is an internal protocol used by the Runtime System to synchronize the Abstract User Interface representation on the Front End side. This protocol defines a simple set of operations to edit the Abstract User Interface tree. This protocol is based on a command processing principle (send command, receive answer) and can be serialized to be transported over any network protocol, like HTTP for example.

Both the Abstract User Interface and the Front End Protocol are public to allow third parties to develop their own Front Ends. This enables applications to be deployed on very specific Workstations.

Refer to Front End Protocol for more details about the operations supported by this communication protocol.

Special Features

This section describes special features regarding the user interface domain:

- Character Conversion Table
- Automatic Front-End Startup

Character Conversion Table

Definition

By default, the runtime system expects that the operating system running the programs uses the same character set as the operating system running the front-end. If the character sets are different, you can set an FGLPROFILE configuration parameter to enable character set mapping between the client and the runtime system, when using a single-byte character set runtime system.

Warning: This feature is provided for backward compatibility. With the new protocol, front-ends are able to identify the character set used by the runtime system and automatically make the codeset conversion.

The following FGLPROFILE entry defines the character table conversion file:

```
gui.chartable = "relative-file-path"
```

The $FGLDIR/etc directory is searched for this file. The runtime system automatically adds the ".ct" file extension.

Default value : NULL (no conversion).

Example:

```
gui.chartable = "iso/ansinogr"
```

The runtime system loads the character table from: **$FGLDIR/etc/iso/ansinogr.ct**

Warnings:

1. The runtime system automatically adds the ".ct" file extension.
2. Character set conversion does not occur when using TUI mode (FGLGUI=0).

Automatic front-end startup

Definition

The runtime system tries to open a connection to the graphical front-end according to the FGLSERVER environment variable. This requires having the front-end already started and listening to the TCP port defined according to FGLSERVER.

In some configurations, such as *X11 workstations* or *METAFRAME/Citrix Winframe* or *Microsoft Windows Terminal Server*, each user may want to start his own front-end to have a dedicated process. This can be done by starting the front-end automatically when the Genero program executes, according to the DISPLAY (X11) or SESSIONNAME/CLIENTNAME (WTSE) environment variables.

Usage:

By default the runtime system always tries to connect to a front-end according to FGLSERVER. If this variable is not set, it tries to connect to the "**localhost:0**" GUI server. If this still does not work, automatic front-end startup takes place, if the `gui.server.autostart.cmd` FGLPROFILE entry is set.

Warning: If the `gui.server.autostart.cmd` entry is not defined, automatic front-end startup does not occur.

To enable automatic front-end startup, you configure **gui.server.autostart.*** entries in FGLPROFILE.

The '**cmd**' entry can be used to define what command should be executed to start the front-end:

```
gui.server.autostart.cmd = "gdc -p %d -q -M"
```

Here, **%d** will be replaced by the TCP port the front-end must listen to.

By default the runtime system waits for two seconds before it tries to connect to the front-end. You can change this delay with the '**wait**' entry:

```
gui.server.autostart.wait = 5   -- wait five seconds
```

The runtime system tries to connect to the front-end ten times. You can change this with the '**repeat**' entry:

```
gui.server.autostart.repeat = 3 -- repeat three times
```

The following FGLPROFILE entries can be used to define workstation id to front-end id mapping:

```
gui.server.autostart.wsmap.max = 3
gui.server.autostart.wsmap.1.names = "fox:1.0,fox.sxb.4js.com:1.0"
gui.server.autostart.wsmap.2.names = "wolf:1.0,wolf.sxb.4js.com:1.0"
gui.server.autostart.wsmap.3.names = "wolf:2.0,wolf.sxb.4js.com:2.0"
```

The first '**wsmap.max**' entry defines the maximum number of front-end identifiers to look for. The '**wsmap.N.names**' entries define a mapping for each GUI server, where **N** is the front-end identifier. The value of those entries defines a comma-separated list of workstation names to match.

On X11 configurations, a workstation is identified by the **DISPLAY** environment variable. In the above example, "**fox:1.0**" identifies a workstation that will make the runtime start a front-end with the number **1**.

On Windows Terminal Server, the **CLIENTNAME** environment variable identifies the workstation. If no corresponding front-end id can be found in the '**wsmap**' entries, the front-end number defaults to the id of the session defined by the **SESSIONNAME** environment variable, plus one. The value of this variable has the form "***protocol#id***"; for example, "**RDP-Tcp#4**" would automatically define a front-end id of **5** (4+1).

Tips:

1. If the front-end processes are started on the same machine as the runtime system, you do not need to set the FGLSERVER environment variable. This will then default to '**localhost:*id***', where *id* will be detected according to the 'wsmap' workstation mapping entries.
2. If the front-end is executed on a middle-tier machine that is different from the application server, **MIDHOST** for example, you can set FGLSERVER to "MIDHOST" without a GUI server id. The workstation mapping will automatically find the id according to 'wsmap' settings.
3. Some clients, such as the Genero Desktop Client (GDC), raise the control panel to the top of the window stack when you try to restart it. In this case the program window might be hidden by the GDC control panel. To avoid this problem, you can use the -M option to start the GDC in minimized mode.

Installation and Setup

This chapter includes instructions for installing Genero BDL on either UNIX or Windows platforms.

Summary:

- 1. Supported Operating Systems
- 2. Hardware Requirements
  - 2.1 Network Card
  - 2.2 Memory and Processor
  - 2.3 Disk Space
- 3. Software Requirements
  - 3.1 Internet access
  - 3.2 Database Client Software
  - 3.3 C Compiler for C Extensions
- 4. Installing the Product
  - 4.1 Genero BDL Packages
  - 4.2 Pre-installation tasks
  - 4.3 Running the installation program
  - 4.4 Post-installation tasks
- 5. Licensing the Product
  - 5.1 License Basics
  - 5.2 Registering the license
  - 5.3 Getting license information
  - 5.4 Removing the license
  - 5.5 Re-installing the license
  - 5.6 The FGLDIR/lock directory
  - 5.7 Using a license server
- 6. Upgrading the Product
  - 6.1 Pre-upgrade tasks
  - 6.2 Licensing an upgraded installation
  - 6.3 Post-upgrade tasks
- 7. Operating System Specific Notes
  - HP/UX
  - IBM AIX
  - Microsoft Windows
  - SCO Unixware

See also: Tools and Components, Localization Support, Environment Variables.

---

1. Supported Operating Systems

Genero BDL is supported on a large brand of operating systems, such as Linux, IBM AIX, HP-UX, SUN Solaris and Microsoft Windows.

Each Genero BDL package is identified with an operating system code (hpx1100, w32vc71). You must install the Genero BDL package corresponding to the operating system that you use.

For the detailed list of supported operating systems, please refer to the Four J's support web site.

---

2. Hardware Requirements

Genero BDL does not require any particular hardware except a network card for license control.

2.1 Network Card

A network card is required by the license manager. It is not possible to install Genero BDL on a computer without a network card.

2.2 Memory and Processor

In a runtime environment, memory and processor requirements are dependent on the number of users and the type of database server. Each DVM process requires 2 Mb to 6 Mb, based on the database client software. For example, a typical requirement for a 30 user runtime environment with an Informix database server is a 500 MHz processor with 512 Mb RAM.

2.3 Disk Space

According to the operating system and the type of installation (development or runtime environment), the total disk space required can vary from 20 Mb to 25 Mb.

Warning: During installation, about 15 Mb are needed in the current temporary directory.

---

3. Software Requirements

Genero BDL requires the following software to be installed on the system:

1. An internet access

2. The database client software
3. A C compiler if you need to compile C extensions

3.1 Internet access

In order to license the product online from the Four J's Web site (http://www.4js.com), you need an Internet browser and an Internet connection.

3.2 Database Client Software

3.2.1 The database client software

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

Below is a list of database client software examples:

- Informix CSDK (with ESQL/C)
- Genero db client (ODBC driver)
- DB2 Connect (with CLI)
- Oracle Client (with OCI)
- Microsoft SQL Server Client (with ODBC driver)
- PostgreSQL client (libpq)
- MySQL client (libmysqlclient)

3.2.2 The database client library must be a shared object

Starting with Genero 2.00, database drivers are provided as pre-linked shared libraries. There is no need to link a runner or driver on site, but the database client software must provide a shared library corresponding to the one used to link the driver. The table in the next section lists supported database client software versions and the corresponding shared libraries that must exist on the system.

3.2.3 Supported database client versions

This table shows the database drivers with the corresponding database client version and shared libraries:

Driver names	Database client software version	Unix shared libraries	Microsoft Windows DLLs
dbmads3x	Genero DB Client 3.x	libaodbc.so	aodbc.dll
dbmasa8x	Sybase ASA Client 8.x	libdblib8.so	dblibtm.dll
dbmdb28x	DB2 Client 8.x	libdb2.so	db2cli.dll

dbmdb29x	DB2 Client 9.x	libdb2.so	db2cli.dll
dbmmsv8x	SQL Server Client 8.x (SQL Server 2000)	**N/A**	odbc32.dll / SQLSRV32.DLL
dbmmsv9x	SQL Server Client 9.x, old driver (SQL Server 2005)	**N/A**	odbc32.dll / SQLSRV32.DLL
dbmsnc9x	SQL Server Client 9.x, native client (SQL Server 2005)	**N/A**	odbc32.dll / SQLNCLI.DLL
dbmftm9x	SQL Server Client 9.x, FreeTDS client	libtdsodbc.so	**N/A**
dbmifx9x	Informix CSDK 2.80 and higher	libifsql.so, libifasf.so, libifgen.so, libifos.so, libifgls.so, libifglx.so	isqlt09a.dll
dbmora81x	Oracle Client 8.1.x	libclntsh.so	oci.dll
dbmora82x	Oracle Client 8.2.x	libclntsh.so	oci.dll
dbmora92x	Oracle Client 9.2.x	libclntsh.so	oci.dll
dbmoraA1x	Oracle Client 10.1.x	libclntsh.so	oci.dll
dbmoraA2x	Oracle Client 10.2.x	libclntsh.so	oci.dll
dbmpgs81x	PostgreSQL Client 8.1.x	libpq.so	libpq.dll
dbmpgs82x	PostgreSQL Client 8.2.x	libpq.so	libpq.dll
dbmpgs83x	PostgreSQL Client 8.3.x	libpq.so	libpq.dll
dbmmys50x	MySQL Client 5.0.x	libmysqlclient.so	libmysql.dll
dbmmys51x	MySQL Client 5.1.x	libmysqlclient.so	libmysql.dll
dbmmys60x	MySQL Client 6.0.x	libmysqlclient.so	libmysql.dll
dbmmys61x	MySQL Client 6.1.x	libmysqlclient.so	libmysql.dll
dbmodc3x	Generic ODBC Client (ODBC 3.x)	libodbc.so	odbc32.dll

See also Operating System Specific Notes.

3.3 C Compiler for C extensions

If you have C Extensions, you need a C compiler and linker to build the extension library.

For more details about C extensions, see "C Extensions" section in this documentation.

3.3.1 C compiler On UNIX platforms:

Warning: On UNIX platforms, you need a cc compiler on the system where you create the C extension libraries. Some systems may not have a C compiler by default. Make sure you have a C compiler on the system.

3.3.2 C compiler On Microsoft Windows platforms:

Warning: On Windows platforms, it is mandatory to install Microsoft Visual C++ version 7.1 or higher on the system where you create the C extension libraries. You must install the appropriate Genero FGL package according to the version of Visual C++ you have installed. For example, when using Visual C++ 8, you must install the package marked by the `w32vc80` operating system identifier.

---

4. Installing the Product

4.1 Genero BDL packages

The software is provided in self-extractible installation programs. On UNIX platforms, the installation program is a shell script (with a .sh extension). On Windows platforms, it is an executable program (with a .exe extension).

Genero BDL package files follow a specific naming convention:

```
fjs-product-version-osident.extension
```

where:

1. *product* is the product identifier.
2. *version* is the release number of the software (1.10.1a).
3. *osident* is the operating system identifier.
4. *extension* is **sh** on UNIX platforms and **exe** on Windows platforms.

Examples:

```
fjs-fgl-1.10.1a-a640510.sh
fjs-fgl-1.10.1a-lnxlc22.sh
fjs-fgl-1.10.1a-wnt0430.exe
```

4.2 Pre-installation tasks

Before launching the installation program, make sure:

1. You have a license number and a license key for Genero BDL development or runtime.
   See Licensing for more details.
2. You are using a supported operating system.
3. You are connected to the system as a user with sufficient privileges to install the software in the target directory.
4. Your configuration matches all hardware requirements and software requirements.
5. You have access to all needed DLLs (PATH) or shared libraries (LD_LIBRARY_PATH).
6. You have set the environment variables for the database client software (INFORMIXDIR/INFORMIXC, ORACLE_HOME, DB2DIR, PGDIR, LD_LIBRARY_PATH).
7. You can use the C compiler if you need to create C Extensions.

Warning: Before starting the installation program, make sure that the database client environment variables are set.

4.3 Running the installation program

The product is provided as an auto-extractible installation program (product files and installation program are provided in the same file). The name of the package includes the operating system type and version. Ensure the package name corresponds to your system before starting the installation program.

4.3.1 Installing on UNIX platforms

On Unix platforms, Genero J's BDL is provided as an auto-extractible shell script. Distribution files and installation program are provided in the same file.

The installation program has options. Display the installation program options using the -h option:

```
$ /bin/sh fjs-fgl-1.10.1a-aix0430.sh -h
```

To perform the installation, run the auto-extractible shell script with the -i option:

```
$ /bin/sh fjs-fgl-1.10.1a-aix0430.sh -i
```

The installation program determines the operating system and checks that all the system requirements are met before starting to copy the product files to your disk.

At this point, follow the online instructions.

4.3.2 Installing on Microsoft Windows platforms

On Microsoft Windows, Genero BDL is provided with a standard Windows setup program. Distribution files and installation program are provided in the same file.

To perform the installation, login as a user with Administrator privileges and simply start the executable program in the "Start" + "Run" window:

```
fjs-fgl-1.10.1a-wnt0430.exe
```

At this point, follow the online instructions.

4.4 Post-installation tasks

After installing the product, you can look at the files provided in FGLDIR/release directory.

It is recommended that you first read the license terms provided in the "license.txt" file. Read this file carefully before using the product in production.

The release notes are in the "readme.txt" file. This file contains important last-minute information that may not be found in the documentation.

Development team changes are provided in the "changes.txt" file. This file contain detailed technical information about changes in the BDL source. You should only reference this file when you cannot find an answer in the "readme.txt" file.

According to the database server you want to connect to, you will need to set up the correct database driver in FGLPROFILE. The default database driver is Informix. For more details about database driver configuration, see Connections.

---

5. Licensing the Product

5.1 License basics

During the installation, you are prompted to license the software. A license must be entered before you can use the product.

During the first installation, you need the license number and license number key supplied with the product package. For example:

```
F4Z#X34006TG + GFAS9FD78XDT
```

When upgrading, the product is installed over the existing directory having a valid license. You do not have to re-enter the license keys.

Warning: Neither the serial number nor the installation number will ever contain the letter O: They can only contain the digit 0 (zero).

5.2 Registering the license

To perform a full licensing, you will be prompted for the **license number** and **license number key**. An **installation number** will be generated from the license number and license number key. Go to the http://www.4js.com web site to get the **installation number key** (or contact your local Four J's support center if you fail to get the key from the web site). Enter the installation number key to complete licensing.

You have 30 days to enter the installation number key. If you cannot get the installation number key, you will have to complete the licensing manually by using the following command:

```
$ fglWrt -k installation-number-key
```

5.3 Getting license information

The following command shows the current installation number:

```
$ fglWrt -a info
```

5.4 Removing a license

An existing license can be dropped using the following command:

```
$ fglWrt -d
```

5.5 Re-installing a license

To re-install a license, use the following command:

```
$ fglWrt –l
```

5.6 The FGLDIR/lock directory

When running a BDL program, the license manager uses the **FGLDIR/lock** directory to store information (number of active users). This directory must have access rights for any user running a BDL program. If it does not exist, it is automatically created.

By default, the **FGLDIR/lock** directory is created with `rwxrwxrwx` rights, to let any user access the directory and create files. If you want to restrict the access to a specific group or user, you can use the `FGLWRTUMASK` environment variable to force fglWrt to use a specific mask when creating the lock directory:

```
$ FGLWRTUMASK="022"; export FGLWRTUMASK
```

Warning: The FGLWRTUMASK environment variable must be set for any user executing BDL programs, because the FGLDIR/lock directory can be re-created by any user at first BDL program execution.

5.7 Using a license server

You don't need to install a locale license with **fglWrt** if you can access a license server on the network.

To make the runtime system use a license server, you must set the following FGLPROFILE entries:

Configuration parameter	Description
`flm.server = "<hostname>"`	Defines the name of the license server machine
`flm.license.number = "<number>"`	The license number (see basics)
`flm.license.key = "<key>"`	The license key (see basics)

If needed, you can specify the following optional parameters:

Configuration parameter	Description
`flm.service = <port>`	TCP port number used by license server (default is 6399)
`flm.check = <count>`	Number of iterations between two controls of the user list (default is 10)
`flm.ping = <milliseconds>`	Timeout (ms) for ping to detect license server machine (default is 3000)
`flm.ps = "<ps command>"`	Command to get the number of processes (default is "ps -ae")

6. Upgrading the Product

6.1 Pre-upgrade tasks

1. Verify the FGLDIR environment is set to the directory you want to upgrade.
2. Verify the user rights (you should login as the owner of the current installed files) and ensure all binaries can be overwritten.
3. Stop all running programs before starting the installation.

6.2 Licensing an upgraded installation

It is not necessary to re-enter the license of the product, as long the new version is installed into an existing installation directory and the new version to be installed is not a major version number change.

Warning: If you upgrade to a new release with a major version number change, you will have to re-license the product again.

6.3 Post-upgrade tasks

When migrating to a major Genero FGL version (for example, from 1.20 to 1.33), you must recompile the sources and form files. While recompilation is not needed when migrating to minor versions (for example, from 1.32 to 1.33), it is recommended to benefit from potential p-code optimizations of the new version.

If required, you may need to re-create the C Extension libraries. Starting with version 2.00 C extension libraries must be provided as dynamically loadable modules and thus should not required a rebuild. However, if Genero C API header files have changed, consider recompiling you C extensions. Check FGLDIR/include/f2c for C API header file changes.

7. Operating System Specific Notes

HP/UX

Thread Local Storage in shared libraries

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

PostgreSQL on HP/UX LP64

On HP/UX LP64, the PostgreSQL database driver should be linked with the **libxnet** library if you want to use networking. You can force the usage of libxnet by setting the **LD_PRELOAD** environment variable to **/lib/pa20_64/libxnet.sl**.

IBM AIX

LIBPATH environment variable

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

Shared libraries archives

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

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

IBM provides a document describing linking on AIX systems. It is recommended that you read this document.

http://www.ibm.com/servers/esdd/pdfs/**aix_ll**.**pdf**

The dump command

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

```
$ dump -Hv -X64 dbmora92x.so
```

Unloading shared libraries from memory

In production environments, AIX loads shared libraries into the system shared library segment in order to improve program load time. Once a shared library is loaded, other

programs using the same library are just attached to that memory segment. This works fine as long as you don't need to change the shared library (to replace it with a new version for example).

Once a shared library is loaded by the system, you cannot copy the executable file unless you unload the library from the system memory. Thus, you will probably need to unload the Genero shared libraries before installing a new version of the software. This problem occurs when installing in the same directory, but can also happen when installing in a different directory. As shared libraries will have the same name, AIX will not allow to load multiple versions of the same library. Therefore, before installing a new version of Genero, make sure all shared libraries are unloaded from the system memory.

To get the list of shared libraries currently loaded into memory, use the **genkld** command. The **genld** command collects the list of all processes and reports the list of loaded objects corresponding to each process. Then, use the **slibclean** command to unload a shared library from the system shared library segment.

POSIX Threads and shared libraries

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

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

Microsoft Windows

Microsoft Visual C version

You need Microsoft Visual C++ compiler to create C Extensions. Make sure you have installed the correct Genero package, according to the MSVC version you have installed on your system. MSVC runtime libraries of different VC++ versions are not compatible. Refer to the name of the Genero package to check the VC++ version compatibility.

Searching binary dependencies

Microsoft Visual C provides the **dumpbin** utility.

To check for DLL dependency, you can use the **/dependents** option:

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

Dump of file mylib.dll

File Type: EXECUTABLE IMAGE
```

```
Image has the following dependencies:

isqlt09a.dll
MSVCR71.dll
KERNEL32.dll

Summary

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

SCO Unixware

Supported locales

Unixware 7.1 only supports a subset of the UTF-8 character set, named *ISO-10646-Minimum-European-Subset*. This character set does not include all characters defined by UTF-8. Therefore, you can experience problems when running an application using UTF-8 characters that are not in the *ISO-10646-Minimum-European-Subset* character set.

Tools and Components

Summary:

- Product Information Viewer (fpi)
- Runtime System Program (fglrun)
- Form Compiler (fglform)
- Program Compiler (fgl2p)
- Module Compiler (fglcomp)
- Module Linker (fgllink)
- Message Compiler (fglmkmsg)
- Database Schema Extractor (fgldbsch)
- Localized String File Compiler (fglmkstr)

See also: Installation and Setup.

---

fpi

Purpose:

The `fpi` utility program is provided to display product information, such as the version number.

Syntax:

```
fpi [options]
```

Notes:

1. *options* are described below.

Options:

Option	Description
`-V`	Displays version information for the tool.
`-h`	Displays options for the tool. Short help.
`-l`	Displays version information for all BDL tools and components.

Usage:

This tool displays the product version number. This version number is useful if you need to identify the installed software. For example, you must reference the product version when you declare a bug.

fglrun

Purpose:

The `fglrun` tool is the runtime system program that executes p-code BDL programs.

Syntax:

```
fglrun [options] program[.42r] [argument [...]]
```

Notes:

1. *options* are described below.
2. *program*.42r is the program name.
3. *argument* is a program argument passed to the program.
4. The **.42r** form file extension is optional.

Options:

Option	Description
`-V`	Display version information for the tool.
`-h`	Displays options for the tool. Short help.
`-i { mbcs }`	Displays information. `-i mbcs` displays information about multi-byte character set settings.
`-d`	Start in debug mode. See Debugger for more details.
`-e extfile [,...]`	Specify a C-Extension module to be loaded. This option can take a comma-separated list of extensions.
`-l`	Link pcode nodules together, see Compiling Programs.
`-o outfile`	Specify the output file for the link mode (-l option).
`-b`	Displays compiler version information of the module, see Compiling Programs.
`-p`	Generate profiling information to stderr (UNIX only). See Profiler for more details.
`-s`	Displays size information in bytes about the module, see Optimization.
`-M`	Display a memory usage diagnostic when program ends, see Optimization.
`-m`	Check for memory leaks. If leaks are found, displays memory usage diagnostic and stops with status 1, see Optimization.

Usage:

This tool executes BDL programs.

```
fglrun myprogram.42r -x 123
```

fglform

Purpose:

The `fglform` tool compiles form specification files into XML formatted files used by the programs.

Syntax:

```
fglform [options] srcfile[.per]
```

Notes:

1. *options* are described below.
2. *srcfile*.per is the form specification file.
3. The **.per** form file extension is optional.

Warning:

1. All **.per** form specification files used by the program must be compiled before usage.

Options:

Option	Description
`-V`	Display version information for the tool.
`-h`	Displays options for the tool. Short help.
`-i { mbcs }`	Displays information. `-i mbcs` displays information about multi-byte character set settings.
`-m`	Extract localized strings.
`-M`	Write error messages to standard output instead of creating a **.err** error file.
`-W { all }`	Display warning messages. Only `-W all` option is supported for now.
`-E`	Preprocess only.
`-p option`	Preprocessing option. Here *option* can be one of: - nopp: Disable preprocessing. - noli: No line number information (only with -E option). - fglpp: Use # syntax instead of & syntax.
`-I path`	Provides a path to search for include files.
`-D ident`	Defines the macro 'ident' with the value 1.

Usage:

This tool compiles a **.per form specification file** into a **.42f** compiled version:

```
fglform custform.per
```

The **.42f** compiled version is an XML formatted file used by BDL programs when a form definition is loaded with the OPEN FORM or OPEN WINDOW WITH FORM instructions.

---

fglmkmsg

Purpose:

The `fglmkmsg` tool compiles message files into a binary version used by the BDL programs.

Syntax:

```
fglmkmsg [options] srcfile [outfile]
```

Notes:

1. *options* are described below.
2. *srcfile* is the source message file.
3. *outfile* is the destination file.

Warning:

1. All **.msg** message files used by the program must be compiled before usage.

Options:

Option	Description
`-V`	Display version information for the tool.
`-h`	Displays options for the tool. Short help.
`-r msgfile`	De-compiles a binary message file.

Usage:

This tool compiles a **.msg message file** into a **.iem** compiled version:

```
fglmkmsg mess01.msg
```

For backward compatibility, you can specify the output file as second argument:

```
fglmkmsg mess01.msg mess01.iem
```

The **.iem** compiled version can be used by BDL programs, for example, when the `HELP` clause is used in a MENU or INPUT instruction.

See message files for more details.

---

fglcomp

Purpose:

The `fglcomp` tool compiles BDL program sources files into a p-code version.

Syntax:

```
fglcomp [options] srcfile[.4gl]
```

Notes:

1. *options* are described below.
2. *srcfile*.4gl is the program source file.
3. The **.4gl** extension is optional.

Warnings:

1. The **.42m** p-code modules must be linked together with fgllink or fgl2p in order to create a runable program.

Options:

Option	Description
`-V`	Display version information for the tool.
`-h`	Displays options for the tool. Short help.
`-i { mbcs }`	Displays information. `-i mbcs` displays information about multi-byte character set settings.
`-S`	Dump Static SQL statements found in the source.
`-m`	Extract localized strings.
`-M`	Write error messages to standard output instead of creating a **.err** error file.
`-W what`	Display warning messages. For a complete description, see below.
`-E`	Preprocess only.
`-p option`	Preprocessing option. Here *option* can be one of: - nopp: Disable preprocessing. - noli: No line number information (only with -E option). - fglpp: Use # syntax instead of & syntax.

`-G`	Produce .c and .h globals interface files for C Extensions.
`-I path`	Provides a path to search for include files.
`-D ident`	Defines the macro 'ident' with the value 1.

Usage:

This tool compiles a **.4gl** into a **.42m** p-code module that can be linked to other modules to create a program or a library.

```
fglcomp customers.4gl
```

If a compilation error occurs, the tool generates a file that has the **.err** extension, with the error message inserted at the line where the error occurred. You can change this behavior by using the `-M` option to display the error message to the standard output.

To create a executable program, the fgllink or fgl2p tool must be used to link the **.42m** compiled file with other modules.

The -W option

The `-W` option can be used to check for wrong language usage, that must be supported for backward compatibility. When used, this option helps to write better source code.

The argument following `-W` option can be one of `return`, `unused`, `stdsql`, `print`, `error` and `all`.

- Using `-W all` enables all warning flags.
- Using `-W error` makes the compiler stop if any warning is raised, as if an error occurred.
- The `-W unused` option displays a message for all unused variables.
- The `-W return` option displays a warning if the same function returns different number of values with several RETURN..
- The `-W stdsql` option displays a message for all non-portable SQL statements or language instructions.
- The `-W print` option displays a message when the PRINT instruction is used outside a `REPORT`.

The `-W` option also supports the negative form of arguments by using the `no-` prefix as in: `no-return`, `no-unused`, `no-stdsql`. You might need to use these negative form in order to disable some warning when using the `-W all` option:

```
fglcomp -Wall -Wno-stdsql customers.4gl
```

The order of warning arguments is important: switches will be enabled/disabled in the order of appearance in the command line. Using the negative form of warning arguments *before* `-W all` makes no sense.

fgllink

Purpose:

The `fgllink` tool assembles p-code modules compiled with fglcomp into a **.42r** program or a **.42x** library.

Syntax:

To create a library:

```
fgllink [options] -o outfile.42x module.42m [...]
```

To create a program:

```
fgllink [options] -o outfile.42r { module.42m | library.42x } [...]
```

Notes:

1. *options* are described below.
2. *outfile*.42r is the name of the program to be created.
3. *outfile*.42x is the name of the library to be created.
4. *module*.42m is a p-code module compiled with fglcomp.
5. *library*.42x is the name of a library to be linked.

Options:

Option	Description
`-V`	Display version information for the tool.
`-h`	Displays options for the tool. Short help.
`-o outfile.ext`	Output file specification, where *ext* can be **42r** for a program or **42x** for a library.
otheroption	Other options are passed to fglrun for linking.

Usage:

This tool links **.42m** p-code modules together to create a **.42x** library or a **.42r** program file.

```
fgllink -o myprog.42x module1.42m module2.42m lib1.42x
```

Note that `fgllink` is just a wrapper calling fglrun with the **-l** option.

---

fgl2p

Purpose:

The `fgl2p` tool compiles source files and assembles p-code modules into a **.42r** program or a **.42x** library.

Syntax:

To create a library:

```
fgl2p [options] -o outfile.42x { pcodem.42m | srcfile.4gl } [...]
```

To create a program:

```
fgl2p [options] -o outfile.42r { pcodem.42m | srcfile.4gl | library.42x
} [...]
```

Notes:

1. *options* are described below.
2. *outfile*.42r is the name of the program to be created.
3. *outfile*.42x is the name of the library to be created.
4. *pcodem*.42m is a p-code module compiled with fglcomp.
5. *source*.4gl is a program source file.
6. *library*.42x is the name of a library to be linked.

Options:

Option	Description
`-V`	Display version information for the tool.
`-h`	Displays options for the tool. Short help.
`otheroption`	Other options are passed to the linker or compiler.

Usage:

This tool can compile **.4gl** source files and link **.42m** p-code modules together, to create a **.42x** library or a **.42r** program file.

```
fgl2p -o myprog.42x module1.4gl module2.42m lib1.42x
```

This tool is provided for convenience, in order to create programs or libraries in one command line. It uses the fglcomp and the fgllink tools to compile and link modules together.

fgldbsch

Purpose

The *Database schema extractor* is the tool provided to generate the Database Schema Files from an existing database.

Syntax:

```
fgldbsch -db dbname [options]
```

Notes:

1. *dbname* is the name of the database from which the schema is to be extracted.
2. *options* are described below.

Options:

Option	Description
`-V`	Display version information for the tool.
`-h`	Displays options for the tool. Short help.
`-H`	Display long help.
`-v`	Enable verbose mode (display information messages).
`-ct`	Display data type conversion tables.
`-db dbname`	Specify the database as *dbname*. This option is required to generate the schema files.
`-dv dbdriver`	Specify the database driver to be used.
`-un user`	Define the user name for database connection as *user*.
`-up pswd`	Define the user password for database connection as *pswd*.
`-ow owner`	Define the owner of the database tables as *owner*.
`-cv string`	Specify the data type conversion rules by character positions in *string*.
`-of name`	Specify output files prefix, default is database name.
`-tn tabname`	Extract the description of a specific table.
`-ie`	Ignore tables with columns having data types that cannot be converted.
`-cu`	Generate upper case table and column names.
`-cl`	Generate lower case table and column names.
`-cc`	Generate case-sensitive table and column names.
`-st`	Generate database system tables.

Usage:

The `fgldbsch` tool extracts the schema description for any database supported by the product. For more details about generated schema files, see Database Schema Files.

---

fglmkstr

Purpose:

The `fglmkstr` tool compiles localized string files.

Syntax:

```
fglmkstr [options] source-file[.str]
```

Notes:

1. *options* are described below.
2. *source-file* is the **.str** string file. You can omit the file extension.

Options:

Option	Description
-V	Display version information for the tool.
-h	Displays options for the tool. Short help.

Usage:

This tool is used to compile **.str** localized string files into **.42s** files.

For more details, see Localized Strings.

Frequently Asked Questions

This page contains questions frequently asked when migrating applications from BDL V3 to Genero BDL.

FAQ001: When using Genero, why do I have a different display than with BDL V3?
FAQ002: Why does an empty window always appear?
FAQ003: Why do some COMMAND KEY buttons no longer appear?
FAQ004: Why aren't the elements of my forms aligned properly?
FAQ005: Why doesn't the ESC key validate my input?
FAQ006: Why doesn't the CTRL-C key cancel my input?
FAQ007: Why do the gui.* FGLPROFILE entries have no effect?
FAQ008: Why do I get a link error when using the FGL_GETKEY() function?

FAQ001: When using Genero, why do I have a different display than with BDL V3?

Explanation:

Genero introduces major Graphical User Interface enhancements that sometimes require code modification. With BDL V3, application windows created with the OPEN WINDOW instruction were displayed as static boxes in the main graphical window. In the new GUI mode of Genero, application windows are displayed as independent, resizable graphical windows.

Links: Dynamic User Interface, Windows, Application class.

FAQ002: Why does an empty window always appear?

Description:

An additional empty window appears when I explicitly create a window with OPEN WINDOW (following the new window management rules).

```
01 MAIN
02   OPEN WINDOW w1 AT 1,1 WITH FORM "form1"
03   MENU "Example"
04    COMMAND "Exit"
05     EXIT MENU
06   END MENU
07   CLOSE WINDOW w1
08 END MAIN
```

Explanation:

In the new standard GUI mode, all Windows are displayed as real front-end windows, including the default `SCREEN` Window. When an application starts, the runtime system creates this default SCREEN Window, as in version 3. This is required because some applications use the `SCREEN` Window to display forms (they do not use the OPEN WINDOW instruction to create new windows). So, to facilitate V3 to Genero migration, the runtime system must keep the default `SCREEN` window creation; otherwise, existing applications would fail if their code was not modified.

Solution:

You can either execute a CLOSE WINDOW SCREEN at the beginning of the program, to close the default window created by the runtime system, or use the OPEN FORM + DISPLAY FORM instructions, to display the main form in the default `SCREEN` window.

Example:

```
01 MAIN
03   OPEN FORM f FORM "form1"
03   DISPLAY FORM f
04   MENU "Example"
05    COMMAND "Exit"
06     EXIT MENU
07   END MENU
08 END MAIN
```

FAQ003: Why do some COMMAND KEY buttons no longer appear?

Description:

When creating a MENU with `COMMAND KEY(`*`keyname`*`) "`*`option`*`"` clause, the button for *keyname* is no longer displayed:

```
01 MAIN
02   MENU "Example"
03     COMMAND "First"
04       EXIT PROGRAM
05     COMMAND KEY (F5) "Second"
06       EXIT PROGRAM
07     COMMAND KEY (F6) -- Third is a hidden option
08       EXIT PROGRAM
09   END MENU
10 END MAIN
```

Explanation:

In BDL Version 3, when using the MENU instruction, several buttons are displayed for each clause of the type `COMMAND KEY(keyname) "option"`: one for the menu option, and others for each associated key.

When using Genero, for a named MENU option defined with `COMMAND KEY`, the buttons of associated keys are no longer displayed (F5 in our example), because there is already a button created for the named menu option. The so called "hidden menu options" created by a `COMMAND KEY(keyname)` clause (F6 in our example) are not displayed as long as you do not associate a label, for example with the FGL_SETKEYLABEL() function.

FAQ004: Why aren't the elements of my forms aligned properly?

Description:

In my forms, I used to align labels and fields by character, for typical terminal display. But now, when using the new LAYOUT section, some elements are not aligned as expected. In the following example, the beginning of the field `f001` is expected in the column near the end of the digit-based text of the first line, but the field is actually displayed just after the label "`Name:`":

```
01 DATABASE FORMONLY
02
03 LAYOUT
04   GRID {
05   01234567890123456789
06    Name:             [f001       ]
07   }
08   END
09 END
10
11 ATTRIBUTES
12 f001 = formonly.field1 TYPE CHAR;
13 END
```

Explanation:

By default, BDL Genero displays form elements with proportional fonts, using layout managers to align these elements inside the window. In some cases, this requires a review of the content of form screens when using the new layout management, because the layout is based on new alignment rules which are more abstract and automatic than the character-based grids in Version 3.

In most cases, the form compiler is able to analyze the layout section of form specification files in order to produce an acceptable presentation, but sometimes you will have to touch the form files to give hints for the alignment of elements.

Solution:

In the above example, the field `f001` is aligned according to the label appearing on the same line. By adding one space before the field position, the form compiler will understand that the field must be aligned to the text in the first line:

```
01 DATABASE FORMONLY
02
03 LAYOUT
04  GRID {
05  01234567890123456789
06   Name:              [f001       ]
07  }
08  END
09 END
10
11 ATTRIBUTES
12 f001 = formonly.field1 TYPE CHAR;
13 END
```

In the next example, the fields are automatically aligned to the text in the first line:

```
01 DATABASE FORMONLY
02
03 LAYOUT
04  GRID {
05                 First         Last
06    Name:        [f001      ]  [f002       ]
07  }
08  END
09 END
10
11 ATTRIBUTES
12 f001 = formonly.field1 TYPE CHAR;
13 f002 = formonly.field2 TYPE CHAR;
14 END
```

FAQ005: Why doesn't the ESC key validate my input?

Description:

The traditional 4GL ESC key does not validate an INPUT, but cancels it instead!

Explanation:

To follow front end platform standards (like Microsoft Windows for example), Genero must reserve the ESC key as the standard key to cancel the current interactive statement.

Solution:

You can change the accelerator keys for the 'accept' action with Action Defaults. However, is not recommended to change the defaults, because ESC is the standard key to be used to cancel a dialog in GUI applications.

FAQ006: Why doesn't the CTRL-C key cancel my input?

Description:

The traditional 4GL CTRL-C key does not cancel an INPUT.

Explanation:

To follow front end platform standards (like Microsoft Windows for example), Genero BDL must reserve the CTRL-C key as the standard key to copy the current selected text to the clipboard, for cut and paste.

Solution:

You can change the accelerator keys for the 'cancel' action with Action Defaults. However, is not recommended to change the defaults, because ESC is the standard key to be used to cancel a dialog in GUI applications.

FAQ007: Why do the gui.* FGLPROFILE entries have no effect?

Description:

The `gui.*` and some other FGLPROFILE entries related to graphics no longer have effect.

Explanation:

These entries are related to the old user interface. They are no longer supported. In version 3, the `gui.*` entries were interpreted by the front end. As the user interface has completely been re-designed, the `gui.*` entries have been removed, too .

Solution:

Review the definition of these entries and use the new possibilities of the Dynamic User Interface.

Entry	Replacement
`menu.Style`	None, no longer

	needed.
`key.`*`key-name`*`.order`	None, no longer needed.
`Menu.defKeys`	None, no longer needed.
`InputArray.defKeys`	None, no longer needed.
`DisplayArray.defKeys`	None, no longer needed.
`Input.defKeys`	None, no longer needed.
`Construct.defKeys`	None, no longer needed.
`Prompt.defKeys`	None, no longer needed.
`Sleep.defKeys`	None, no longer needed.
`GetKey.defKeys`	None, no longer needed.
`gui.local.edit.*`	None, no longer needed.
`gui.preventClose.message`	None, *cancel* action is sent when the user closes a window.
`gui.chartable`	None, no longer needed.
`gui.whatch.delay`	None.
`gui.useOOB.interrupt`	None.
`gui.containerType`	None.
`gui.containerName`	None.
`gui.mdi.*`	None.
`gui.screen.clientPositioning`	None.
`gui.screen.size.*`	None.
`gui.screen.x`	None.
`gui.screen.incrx`	None.
`gui.screen.y`	None.
`gui.screen.incry`	None.
`gui.screen.withwm`	None.
`gui.workSpaceFrame.noList`	None.
`gui.workSpaceFrame.screenArray.optimalStretch`	None.
`gui.workSpaceFrame.screenArray.compressed`	None.
`gui.controlFrame.visible`	None.
`gui.controlFrame.position`	None.
`gui.controlFrame.scrollVisible`	None.
`gui.controlFrame.scroll.*`	None.

`gui.bubbleHelp.*`	None, front end specific.
`gui.directory.images`	None, front end specific.
`gui.toolbar.dir`	None, front end specific.
`gui.toolbar.*`	The new toolbar definition.
`gui.menu.*`	None.
`gui.menuButton.*`	None.
`gui.button.*`	None.
`gui.empty.button.visible`	None.
`gui.keyButton.*`	None, front end specific.
`gui.key.add_function`	None.
`gui.key.interrupt`	None.
`gui.key.doubleClick.left`	None.
`gui.key.click.right`	None.
`gui.key.`*`num`*`.translate`	None.
`gui.key.copy`	None, front end specific.
`gui.key.paste`	None, front end specific.
`gui.key.cut`	None, front end specific.
`gui.form.foldertab.*`	None.
`gui.key.forldertab.input.sendNextField`	None.
`gui.key.foldertab.num.selection`	None.
`gui.mswindow.button`	None, front end specific.
`gui.fieldButton.style`	None, front end specific.
`gui.BMPButton.style`	None, front end specific.
`gui.key.radiocheck.invokeexit`	None.
`gui.entry.style`	None, front end specific.
`gui.interaction.inputarray.usehighlightcolor`	None, front end specific.
`gui.mswindow.scrollbar`	None, front end specific.
`gui.scrollbar.expandwindow`	None, front end specific.
`gui.statusbar.*`	None, front end specific.

`gui.display.*`	None.
`gui.user.font.choice`	None.

FAQ008: Why do I get a link error when using the FGL_GETKEY() function?

Description:

This function is no longer supported; it has been removed from the language.

Explanation:

That function waited for a key-press from the user, but this kind of interaction does not fit into the new user interface protocol.

Solution:

Review the program and use standard interactive instructions to manage the interaction with the user.
See the Dynamic User Interface concept.

New Features of the Language

- **Product line 2.1x**
  - Version 2.11
  - Version 2.10

  Product line 2.0x

  - Version 2.02
  - Version 2.01
  - Version 2.00
- **Product line 1.3x**
  - Version 1.33
  - Version 1.32
  - Version 1.31
  - Version 1.30
- **Product line 1.2x**
  - Version 1.20
- **Product line 1.1x**
  - Version 1.10

See also: FAQ List.

Version 2.11

- New database drivers
- Static SQL syntax extension
  - Derived tables and derived column list
  - New transaction isolation levels of Informix 11
  - The CAST operator
- New preprocessor option to remove line number information
- Connecting to Oracle as SYSDBA or SYSOPER
- New methods for ui.ComboBox
- Make current row visible after sort in lists
- Reading pcode build information of older versions

New database drivers

The following database drivers are supported by Genero version 2.11:

- **dbmpgs83x** for a PostgreSQL 8.3.x client.
- **dbmmys51x** for a MySQL 5.1.x client.
- **dbmftm90** for a FreeTDS 0.82 client connecting to SQL Server 2005.
- **dbmsncA0** for a SQL Server Native client connecting to SQL Server 2008.
- **dbmoraB1** for a Oracle 11g client.

Static SQL syntax extension

Derived tables and derived column list

Genero FGL static SQL syntax now supports derived tables and derived column lists in the FROM clause, for example:

```
   SELECT * FROM (SELECT * FROM customers ORDER BY cust_num) AS
t(c1,c2,c3,...)
```

See database server documentation for more details about this SQL feature. Note that Informix 11 does not support the full ANSI SQL 92 specification for derived columns, while other databases like DB2 do. For this reason, fglcomp allows the ANSI standard syntax.

New transaction isolation levels of Informix 11

The SET ISOLATION statement now supports the new Informix 11 clauses for the COMMITTED READ option:

```
   SET ISOLATION TO COMMITTED READ [LAST COMMITTED] [RETAIN UPDATE
LOCKS]
```

When connecting to a non-Informix database, the LAST COMMITTED and RETAIN UPDATE LOCKS are ignored; other databases do not support these options, and have the same behavior as when these options are used with Informix 11.

The CAST operator

The CAST operator can now be used in static SQL statements:

```
   CAST ( expression AS sql-data-type )
```

Note that only Informix data types are supported after the AS keyword.

New preprocessor option to remove line number information

You can now remove line number information with **-p noln** when preprocessing sources to get a readable output:

```
   fglcomp -E -p noln mymodule.4gl
```

Connecting to Oracle as SYSDBA or SYSOPER

In order to execute database administration tasks, you can now connect to Oracle as SYSDBA or SYSOPER with the CONNECT instruction:

```
   CONNECT TO "dbname" USER "scott/SYSDBA" USING "tiger"
```

New methods for ui.ComboBox

The ui.ComboBox class has been extended with new methods: getTextOf() and getIndexOf().

Make current row visible after sort in lists

A new FGLPROFILE entry has been added to force the current row to be automatically shown after a sort in a table:

```
Dialog.currentRowVisibleAfterSort = 1
```

By default, the offset does not change and the current row may disappear from the window. When this new parameter is used, the current row will always be visible. For more details, see Runtime Configuration.

Reading pcode build information of older versions

The **-b** option of fglrun has been extended to recognize headers of pcode modules compiled with older versions of FGL. For more details, see Compiling Programs. Additionally, fglform now writes build information in the 42f files, to identify on the production site what FGL version was used to compile forms.

Version 2.10

- Multiple Dialogs
- TRY/CATCH pseudo statement
- WHENEVER ... RAISE
- SQL Server Native Client driver
- Support for SPLITTER attribute
- Double-click in tables
- New X conversion code in fgldbsch
- SQL Interruption in database drivers
- NULL pointer exceptions can be trapped
- Client socket interface in Channels
- Stack trace
- GUI connection timeout
- Assigning a value to a TEXT variable
- New presentation styles
- fglrun -s option now displays more information
- fglrun -e option takes list of extensions
- Detecting data changes immediately
- Controlling data validation for actions
- New MINWIDTH, MINHEIGHT attribute in forms
- Avoid automatic temporary row in INPUT ARRAY
- New implicit navigation actions in INPUT ARRAY and DISPLAY ARRAY

- New DOM methods to serialize or parse strings
- New I/O methods to read/write TEXT or BYTE from/to files

Multiple Dialogs

A new DIALOG instruction handles different parts of a form simultaneously. See also ui.Dialog class.

TRY/CATCH pseudo statement

The TRY/CATCH pseudo statement can handle exceptions raised by the runtime system.

WHENEVER .... RAISE

Instructs the DVM that an uncaught exception will be handled by the caller of the function. See Exceptions.

SQL Server Native Client driver

Support for SQL Server 2005 Native Client is now provided.

Support for SPLITTER attribute

HBox and VBox containers can now have a splitter. See also Layout Tags.

Double-click in tables

With the new DOUBLECLICK table attribute, it is now possible to send a specific action when the user double-clicks on a row.

New X conversion code in fgldbsch

The fgldbsch tool now supports the X conversion code to ignore table columns of a specific type. This is useful for ROWID-like columns such as SQL Server's *uniqueidentifier* columns.

SQL Interruption in database drivers

Before version **2.10**, SQL interruption was not supported well for Oracle, SQL Server, DB2 and Genero db databases. SQL interruption is now available with all databases providing an API to cancel a long-running query.

For more details, see SQL Programming.

NULL pointer exceptions can be trapped

Error **-8083** will be raised if you try to call an object method with a variable that does not reference an object (that contains NULL):

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

In previous versions, this raised a fatal NULL pointer error. This exception can now be trapped with WHENEVER ERROR.

Client socket interface in Channels

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

Stack trace

For debugging purpose, you can now get the stack trace of the program with the Application.getStackTrace() method.

GUI Connection Timeout

It is now possible to define a timeout delay for front-end connections with the following FGLPROFILE entry:

```
gui.connection.timeout = seconds
```

See Dynamic User Interface for more details.

Assigning a value to a TEXT variable

Before version **2.10**, it was only possible to assign a TEXT to a TEXT variable. Now you can assign STRING, CHAR and VARCHAR values to a TEXT variable. For more details about data type conversions, see the Data Conversion Table.

New Presentation styles

Presentation Styles have been extended:

- The style attribute "**position**" for Windows can be set to "**previous**"
- TextEdit now has the "**textSyntaxHighlight**" attribute (value can be "**per**", more to come...)
- All widgets can now use the "**localAccelerators**" global style attribute to interpret standard navigation and editor keys (like Home/End) without firing an action that uses the same keys as accelerators.

fglrun -s option now displays more information

The -s option of fglrun now reports more information about sizes. See Optimization for more details.

fglrun -e option takes list of extensions

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

```
fglrun -e ext1,ext2,ext3 -e ext4,ext5 myprogram
```

See C Extensions for more details.

Detecting data changes immediately

It is now possible to get an action event when the user modifies the value of a field, with the predefined dialogtouched action.

Controlling data validation for actions

You can now use the validate="no" action default attribute to prevent data validation when executing an action.

New MINWIDTH, MINHEIGHT attributes in forms

It is now possible to define a minimum width and height for forms with the MINWIDTH, MINHEIGHT attributes.

Avoid automatic temporary row in INPUT ARRAY

With the new AUTO APPEND attribute, you can now avoid the automatic creation of a temporary row in INPUT ARRAY.

New implicit navigation actions in INPUT ARRAY and DISPLAY ARRAY

When a DISPLAY ARRAY or INPUT ARRAY is executed, the runtime system now creates two more implicit actions to navigate in the list: **firstrow** and **lastrow**. In previous versions, only **nextrow** and **prevrow** actions were created. You can now bind four buttons to these actions to get a typical list navigation toolbar. Note that the default action views are hidden for these navigation actions.

New DOM methods to serialize or parse strings

The parse() and toString() methods are now available for a DomNode object, and a DomDocument object can be created with createFromString().

New I/O methods to read/write TEXT or BYTE from/to files

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

Version 2.02

- New Static SQL Commands
- Global Variables in C Extensions
- Localization of runtime system error messages
- Debugger enhancement
- Tab index can be zero
- New FGLPROFILE entry for Oracle driver
- New FGLPROFILE entry to define temporary table emulation type

New Static SQL Commands

Some common SQL statements have been added to the static SQL syntax, such as TRUNCATE TABLE, RENAME INDEX, CREATE / ALTER / DROP / RENAME SEQUENCE. See Static SQL Commands.

Global Variables in C Extensions

You can now share global variables between the FGL source and the C Extension, by using the -G option of fglcomp. See Global variables in C Extensions.

Localization of runtime system error messages

It is now possible to customize the runtime system error messages according to the current locale. See Localization for more details.

Debugger enhancement

New debugger commands (ptype).

You can now avoid switching into debug mode with SIGTRAP (Unix) or CTRL-Break (Windows) with the new fglrun.ignoreDebuggerEvent FGLPROFILE entry.

Tab index can be zero

You can now specify a TABINDEX of zero to exclude the form item from the tagging list. See TABINDEX for more details.

New FGLPROFILE entry for Oracle driver

It is now possible to specify the SELECT statement producing the unique session identifier which is used for temporary table names.
See Database vendor specific parameters for more details.

New FGLPROFILE entry to define temporary table emulation type

To emulate Informix temporary tables, you can now set the **temptables.emulation** parameter to use GLOBAL TEMPORARY TABLES instead of permanent tables.
See temporary table emulation for more details.

Version 2.01

- FESQLC compiler (V2)
- DB2 V9.x support
- PostgreSQL V 8.2.x support
- Extension of the form layout tag syntax
- Negative form of warning flags in fglcomp
- Run supports ComSpec variables on Windows

FESQLC compiler (V2)

ESQL/C Compiler. See FESQLC compiler (V2).

DB2 V9.x support

Support of DB2 V9.x. See DB2 V9.x support.

PostgreSQL V 8.2.x support

Support of PostgreSQL 8.2.x. See PostgreSQL V 8.2.x support.

Extension of the form layout tag syntax

The layout tag syntax in grids has been extended to support an ending tag to get better control of form layout.

Negative form of warning flags in fglcomp

The fglcomp compiler now supports a negative form for warning arguments.

Run supports ComSpec variable on Windows

When using the RUN command, the ComSpec environment variable is now used under Windows platforms.

Version 2.00

- Dynamic Runner Architecture
- User defined types
- New Widgets
- Extended Schema Files
- File Management Functions
- Math Functions
- Stored procedure calls
- Informix-like C API library
- Memory usage optimization
- The IMPORT instruction
- WIDTH and HEIGHT attributes in Images
- New debugger commands
- Improved Presentation Styles
- Compiler supports constraints in CREATE TABLE
- Automatic front-end startup
- New channel function to detect EOF
- Responding to CTRL_LOGOFF_EVENT on Windows
- New compiler warning options
- Fourth accelerator definition
- Conditional TTY attributes for all widgets
- New FGL_SETENV() built-in function
- Support for entities in XML reader and writer
- Schema extractor supports now Informix LVARCHAR

Dynamic Runner Architecture

Runner now uses shared libraries; you no longer need to link a runner. See Dynamic Runner Architecture.

User defined types

You can now define your own data type with records or arrays. See User defined types.

New Widgets

New widgets have been added: SLIDER, SPINEDIT, TIMEEDIT.

Extended Schema Files

Database Schema files have been extended for Genero (FIELD item type). See Extended Schema Files.

File Management Functions

File management function library provided as loadable extension. See File Management Functions.

Math Functions

Mathematical function library provided as loadable extension. See Math Functions.

Stored procedure calls

It is now possible to call stored procedures with output parameters. See Stored procedure calls.

Informix-like C API library

C extension support has been extended with Informix-like C API functions. See Informix-like C API library.

Memory usage optimization

The runtime system now shares several static elements among all processes, reducing the memory usage. The shared elements are: Data type definitions, string constants and debug information. For example, when a program defines a string containing a long SQL statement, all Genero processes will share the same string, which is allocated only once.

The IMPORT instruction

To declare a C extension module, you must now use the IMPORT instruction at the beginning of a module.

WIDTH and HEIGHT attributes in Images

You can now specify the WIDTH and HEIGHT attributes for IMAGE form items, as a replacement for PIXELWIDTH / PIXELHEIGHT.

New debugger commands

New commands have been added to the debugger (call, ignore).

Improved Presentation Styles

You can now specify pseudo selectors such as *focus*, *active*, *inactive*, *input*, *display* for fields and *odd* / *even* states for table rows.

Some new style attributes were added:

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

For more details, see Presentation Styles.

Compiler supports constraints in CREATE TABLE

It is now possible to specify primary key, foreign key and check constraints in static CREATE TABLE statements:

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

Automatic front-end startup

In X11 or Windowse TSE environments, you can now automatically start up the front-end with FGLPROFILE entries. See Dynamic User Interface for more details.

New channel function to detect EOF

The Channel class now has an isEof() method to detect end of file.

Responding to CTRL_LOGOFF_EVENT on Windows

It is now possible to ignore the CTRL_LOGOFF_EVENT events on Windows platforms.

New compiler warning options

The fglcomp compiler has new warning flags: See fglcomp for more details.

Fourth accelerator definition

You can now define a fourth accelerator for an action in actions defaults or in the form files.

Conditional TTY attributes for all widgets

It is now possible to specify TTY attributes (COLOR, REVERSE) and conditional TTY attributes (COLOR WHERE) for all type of fields.
See Form Specification Files and COLOR WHERE attribute for more details.

New FGL_SETENV() built-in function

A new built-in function has been added to set an environment variable: FGL_SETENV().

Support for entities in XML reader and writer

The XML reader and writer classes have been extended to properly support markup language entities (like HTML's ** **).

Schema extractor supports now Informix LVARCHAR

The fgldbsch tool can now extract database tables with LVARCHAR columns. The LVARCHAR type is converted to VARCHAR2(n>255) in the .sch file.

Version 1.33

- TypeInfo class
- Generic ODBC support
- MySQL 5 support
- Genero DB 3.4 support
- PostgreSQL 8.1 support
- SQL Server 2005 support
- New license manager
- FESQLC compiler (V1)
- Binary mode in Channel class
- New header files for C extensions
- Block fetch with SQL Server
- Third accelerator definition

TypeInfo class

A class to serialize program variables. See TypeInfo class.

Generic ODBC support

A generic ODBC database driver is now available (code is **odc**). See Generic ODBC support.

MySQL 5 support

MySQL version 5 is now supported. See MySQL 5. support.

Genero DB 3.4 support

Genero DB version 3.4 is now supported. See Genero DB support.

PostgreSQL 8.1 support

PostgreSQL version 8.1 is now supported. See PostgreSQL 8.1 support.

SQL Server 2005 support

Microsoft SQL Server 2005 is now supported. See SQL Server 2005 support.

New license manager

New license manager supporting strict licensing. See New license manager.

FESQLC compiler (V1)

ESQL/C compiler.

Binary mode in Channel class

The base.Channel class now supports a binary mode with the 'b' option, to control CR/LF translation when using DOS files.

New header files for C extensions

Distribution of Datetime.h, Interval.h, loc_t.h header files in FGLDIR/include/f2c.

Block fetch with SQL Server

You can now pre-fetch rows by block with SQL Server to get better performance. Use the following FGLPROFILE entry to specify the maximum number of rows the driver can pre-fetch:

```
dbi.database.<dbname>.msv.prefetch.rows = <count>
```

See "Database vendor specific parameters" in Connections for more details.

Third accelerator definition

You can now define a third accelerator for an action in actions defaults or in the form files.

Version 1.32

- PostgreSQL 8.0 support
- File transfer functions
- Debugger enhancement
- Preprocessor is now integrated in compilers

PostgreSQL 8.0 support

PostgreSQL version 8.0 is now supported (8.0.2 and higher). See PostgreSQL 8.0 support.

File transfer functions

Get/Put functions to transfer files from/to the front-end. See File transfer functions.

Debugger enhancement

New debugger commands (watch with condition, whatis).

Preprocessor is now integrated in compilers

The preprocessor is now part of the compilers and is always enabled. Preprocessing directives start with an ampersand character (**&**).

Version 1.31

- Front-end Protocol Compression
- MySQL 4.1.x support
- Oracle 10g support
- Dynamic C extensions
- New built-in functions

- Interruption handling
- New Dialog method
- Front-end identification

Front-end Protocol Compression

Faster user interface communication. See Front-end Protocol Compression.

MySQL 4.1.x support

MySQL version 4.1.x is now supported, 3.23 is de-supported. See MySQL 4.1.x support.

Oracle 10g support

Oracle version 10g is now supported. See Oracle 10g support.

Dynamic C extensions

C extensions can be loaded dynamically, no need to re-link runner. See Dynamic C extensions.

New built-in functions

The FGL_WIDTH built-in function computes the number of print columns needed to represent a single or multi-byte character.

Interruption handling

Interruption handling with SSH port forwarding - only supported with GDC 1.31!

New Dialog method

New method ui.Form.setFieldStyle() to set a style for a field.

Front-end identification

Improved front-end identification when connecting to GUI client.

Version 1.30

- Preprocessor
- Layout Enhancements

- Presentation Styles
- Localization Support
- Action defaults in forms
- Dialog Control
- Sybase ASA Support
- PostgreSQL 7.4 support
- Build information in 42m modules
- MySQL 3.23 support for Windows platforms
- Upshift / Downshift in Comboboxes
- Message compiler does not require output file any longer
- Breakpoint in source code
- Row highlighting in tables
- Method base.Array.appendElement()
- Compiled Localized String file extension 42s
- Assignment operator
- New fglcomp option for SQL
- Compiler generates standard UPDATE syntax
- Defining color attributes for each table cell
- Form methods in ui.Window
- Method base.StringBuffer.replace()
- Methods base.Channel.readLine() and base.Channel.writeline()
- Dynamic arrays used as data model in INPUT ARRAY / DISPLAY ARRAY
- TITLE attribute for fields
- FGLLDPATH used during link
- Method ui.Dialog.setDefaultUnbuffered()
- Action Defaults applied by DVM
- DATEEDIT supports DBDATE & FORMAT
- New predefined action 'close'
- Tabbing order in TABLEs during INPUT ARRAY
- Preprocessor raises errors for invalid # macros
- ACCEPT xx instruction
- ACCEPT / CANCEL dialog attribute
- Preprocessor disabled by default
- INPUT ARRAY now has default 'append' action
- Linker option -O removed
- Method ui.Window.createForm()
- TopMenu attributes in .per
- Specifying real field size in forms
- Version number in UI protocol
- C-like source preprocessor
- MENU node available in BEFORE MENU
- New HBox tags
- Form layout extensions
- New Table definition attributes
- New ORIENTATION attribute for RADIOGROUPs
- Reviewed fglrun.setenv environment variables handling in FGLPROFILE
- MENU COMMAND generates lowercase action name
- Method ui.Interface.loadTopMenu()
- ON CHANGE fired on click
- New ui.Dialog built-in class
- New ui.Form methods

- Array sub-script operator now returns the sub-array
- Dynamic arrays passed by reference to functions
- Control MDI children with ui.Interface
- Cancel INSERT in AFTER INSERT
- Toolbar and Topenu now have the hidden attribute
- NEXT FIELD CURRENT

Preprocessor

Integrated preprocessor allows use of #include and #define/#ifdef macros. See Preprocessor.

Layout Enhancements

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

Presentation Styles

Decoration attribute can be defined in a style file to set fonts and colors. See Presentation Styles.

Localization Support

Localization Support (multi-byte character sets). See Localization Support.

Action defaults in forms

Action defaults can be specified in forms. See Action defaults in forms.

Dialog Control

Dialog built-in class to provide better control over interactive instructions. See Dialog Control.

Sybase ASA Support

New drivers to connect to Sybase Adaptive Server Anywhere V7 and V8. See Sybase ASA Support

PostgreSQL 7.4 support

Support for PostgreSQL 7.4 with parameterized queries. See PostgreSQL 7.4 support.

Build information in 42m modules

The fglcomp compiler now adds build information in 42m modules. Compiler version of a 42m module can be checked on site by using the fglrun with the -b option:

```
$ fglrun -b module.42m
2004-05-17 10:42:05 1.30.2a-620.10 /devel/tests/module.4gl
```

MySQL 3.23 support for Windows platforms

A MySQL 3.23 driver is now provided for Windows platforms (was previously only provided on Linux).

Upshift/Downshift in Comboboxes

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

Message compiler does not require output file any longer

The fglmkmsg tool now has the same behavior as other tools like fglcomp and fglform: If you give only the source file, the message compiler uses the same file name for the compiled output file, adding the **.iem** extension.

Breakpoint in source code

New BREAKPOINT instruction to stop a program at a given position when using the debugger. It is ignored when not running in debug mode.

Row highlighting in tables

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

Method base.Array.appendElement()

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

Compiled Localized String file extension = 42s

Compiled Localized String files now have the .42s extension. Previous extension was .4ls.

Assignment Operator

New assignment operator := has been added to the language. You can now assign variables in expressions:
IF (i := (j+1)) == 2 THEN

New fglcomp option for SQL

The fglcomp compiler now has a new option to detect non-standard SQL syntax:
fglcomp -W stdsql module.4gl

Compiler generates standard UPDATE syntax

The fglcomp compiler now converts static SQL updates like:

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

to a standard syntax:

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

See also SQL Programming.

Defining color attributes for each table cell

The new method ui.Dialog.SetCellAttributes() lets you define colors for each cell of a table.

Form methods in ui.Window

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

Method base.StringBuffer.replace()

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

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

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

Methods base.Channel.readLine() and base.Channel.writeLine()

New methods to read/write complete lines in Channel built-in class: readLine() and writeLine().

Dynamic arrays used as data model in INPUT ARRAY / DISPLAY ARRAY

When using a dynamic array in INPUT ARRAY or DISPLAY ARRAY, the number of rows is defined by the size of the dynamic array. The SET_COUNT() or COUNT attributes are ignored.

TITLE attribute for fields

The new form field attribute TITLE can be used to specify a table column label with a localized string.

FGLLDPATH used during link

The FGLLDPATH variable is now used during link

Method ui.Dialog.setDefaultUnbuffered()

New class method ui.Dialog.setDefaultUnbuffered() to set the default for the UNBUFFERED mode.

Action Defaults applied by DVM

Action Defaults now applied at element creation by the runtime system. In previous versions this was done dynamically by the front-end. Now, changing an action default node at runtime has no effect on existing elements.

DATEEDIT supports DBDATE & FORMAT

The DATEEDIT field type now supports DBDATE/CENTURY settings and the FORMAT attribute.

New predefined action 'close'

New default action 'close' to control Window closing. You can now write the following to control window closing:

```
ON ACTION close
```

See Windows and Forms.

Tabbing order in TABLEs during INPUT ARRAY

INPUT ARRAY using TABLE container now needs FIELD ORDER FORM attribute to keep tabbing order consistent with visual order of columns.

Preprocessor raises errors for invalid # macros

If enabled, the preprocessor now raises an error when # comments are used in the source. You must replace all # comments by -- comments before using the preprocessor.

ACCEPT xx instruction

New instructions ACCEPT INPUT / ACCEPT CONSTRUCT / ACCEPT DISPLAY to validate a dialog by program.

```
ON ACTION doit
   ACCEPT INPUT
```

ACCEPT / CANCEL dialog attribute

New dialog attribute ACCEPT / CANCEL to avoid creation of default actions 'accept' and 'cancel'.
See Record Input control instructions.

Preprocessor disabled by default

The Preprocessor is now disabled by default; there are no FGLPP / FGLPPOPTIONS environment variables, you must use the -p option of fglcomp/fglform.

INPUT ARRAY now has default 'append' action

New default action 'append' in INPUT ARRAY. Allows you to add a row at the end of the list.

Linker option -O removed

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

Method ui.Window.createForm()

New method ui.Window.createForm() to create an empty form object in order to build forms from scratch at runtime.

TopMenu attributes in .per

TopMenu definition in forms now allows attributes in parenthesis.

Specifying real field size in forms

The form layout syntax now allows you to specify the real width of form items. By default, BUTTONEDIT, COMBOBOX and DATEEDIT get a real width as follows:

```
  if nbchars>2 : width = nbchars - 2; otherwise width = nbchars
```

(Here nbchars is the number of characters used in the layout definition.)

Now you can specify the real width by using a dash '-' in the tag:

```
 1234567
[f01  - ]      width = 5, grid cells used = 7
```

This works also in hbox tags and screen arrays.

Version number in UI protocol

User interface protocol is now controlled with a version number, to check compatibility between the front end and runtime system.

C-like source preprocessor

New integrated preprocessor: The form and source compiler now integrates a preprocessor! You can use macros as in C, such as #include, #define, #ifdef, etc.

MENU node available in BEFORE MENU

Important remark: Before build 530 the MENU has attached the WINDOW when returning from the BEFORE MENU actions. Since build 530 the WINDOW must exist before the MENU statement. So now the MENU node is available in the BEFORE MENU block, but a WINDOW opened or made CURRENT in the BEFORE MENU block will NOT be used.

New HBox tags

Layout GRID now accepts HBox tags to group items horizontally.

Form layout extensions

- Elements in grids now have cell columns and lines plus width & height.
- Form VERSION attribute to distinguish form revisions.
- Layout SPACING attribute to define space between widgets.
- The DEFAULT SAMPLE instruction.
- New form item attributes, like SAMPLE, JUSTIFY, SIZEPOLICY ...

New Table definition attributes

- You can now specify HIDDEN = USER as 'hidden to the user by default'.
- Table columns now have new attribute UNMOVABLE to avoid moving.
- WANTCOLUMNSANCHORED replaced by UNMOVABLECOLUMNS.
- WANTCOLUMNSVISIBLE replaced by UNHIDABLECOLUMNS.
- Tables now accept a WIDTH and HEIGHT attribute to specify a size.

New ORIENTATION attribute for RADIOGROUPs

RADIOGROUP fields now support the attribute ORIENTATION = { VERTICAL | HORIZONTAL }.

Reviewed fglrun.setenv environment variables handling in FGLPROFILE

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

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

MENU COMMAND generates lowercase action name

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

Method ui.Interface.loadTopMenu()

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

ON CHANGE fired on click

The ON CHANGE block is now fired when the user clicks on a checkbox, radiogroup, or changes the item in a combobox.

New ui.Dialog built-in class

New ui.Dialog built-in class available with the DIALOG keyword in all interactive instructions. You can now activate/deactivate fields and actions during a dialog:

```
INPUT ...
     AFTER FIELD field1
     CALL DIALOG.setFieldActive("field2",rec.field1 IS NOT NULL)
     CALL DIALOG.setActionActive("check",rec.field1 IS NOT NULL)
```

New ui.Form methods

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

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

Array sub-script operator now returns the sub-array

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

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

Dynamic arrays passed by reference to functions

Dynamic arrays are now passed by reference to functions. You can change a dynamic array in a function when it is passed as an argument.

Control MDI children with ui.Interface

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

CANCEL INSERT in AFTER INSERT

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

Toolbar and Topmenu now have the hidden attribute

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

Warning: Hiding a toolbar or topmenu option does not prevent the use of the accelerator of the action. Use ui.Dialog.setActionActive()!

NEXT FIELD CURRENT

New keyword for NEXT FIELD: NEXT FIELD CURRENT. Gives control back to the dialog instruction without moving to another field.

Version 1.20

- Debugger
- Program Profiler
- Localized Strings
- Unbuffered Dialogs
- Paged Display Array
- Action Defaults
- Client-side settings saved for each program
- APPEND ROW dialog attribute
- KEEP CURRENT ROW dialog attribute
- UNHIDABLE attribute for image and labels
- TERMINATE REPORT / EXIT REPORT
- TINYINT data type with SQL Server
- Toolbars can be defined in forms
- Topmenus can be defined in forms
- Build version number
- Get a help message text
- Set the current row
- Interruption handling
- StatusBar definition with style attribute
- Field order form
- Runtime system re-written in C
- Passing arrays as function parameter
- Compiler supports ANSI outer joins
- Methods for StringBuffer
- Default items created for COMBOBOX
- ON IDLE clause in dialogs
- Order of INPUT ARRAY trigger execution
- New ui.ComboBox class
- Predefined actions in lists: nextrow / prevrow
- FOREACH infinite loop
- Record comparison
- ON CHANGE trigger
- Program icon
- Form compilation warnings
- FORMAT attribute in LABELs
- SQLSTATE and SQLERRMESSAGE
- Front End Function calls
- New ui.Form built-in class
- TABINDEX for tabbing order
- LSTR operator
- SFMT operator
- ON ROW CHANGE trigger
- New StringTokenizer class
- Faster linker
- Global constants
- ON ACTION in MENUs
- New Application class
- New Channel class

- Predefined 'help' action

Debugger

Integrated debugger with gdb syntax to interface with graphical tools like ddd. See Debugger.

Program Profiler

The Program Profiler can be used to generate statistics of program execution, to find the bottlenecks in the source code.

Localized Strings

Internationalizes your application in different languages with localized strings.

Localized Strings are now supported. You can identify strings to be localized, with the % notation:

```
LAYOUT ( TEXT= %"custlist" )
```

See Localized Strings.

Unbuffered Dialogs

Interactive instructions support the UNBUFFERED mode, to synchronise data model and view automatically. Dialogs can now use the UNBUFFERED attribute, that simplifies INPUT, DISPLAY ARRAY and INPUT ARRAY coding; input/display buffer is no longer used. When you set a variable, the value is automatically displayed to the field. See Unbuffered Dialogs.

Paged Display Array

DISPLAY ARRAY can now work in buffered mode, to avoid loading a big array when you have a lot of rows to display. The DISPLAY ARRAY instruction now has a new ON FILL BUFFER block that can be used with dynamic arrays to feed the dialog with data rows on demand. See Paged Display Array.

Action Defaults

Centralize default attributes for actions in Action Defaults files.

Client side settings saved for each program

Client side settings are now saved in registry according to the 'name' attribute of UserInterface, which can be set with ui.Interface.setName() method. By default UserInterface.name is not set to the name of the program.

APPEND ROW dialog attribute

New attribute APPEND ROW = TRUE/FALSE for INPUT ARRAY instruction. Defines if the user is allowed to add rows at the end of the list.

KEEP CURRENT ROW dialog attribute

New attribute KEEP CURRENT ROW = TRUE/FALSE for DISPLAY ARRAY and INPUT ARRAY instructions. Defines if the current row must remain highlighted when leaving the dialog. The default is FALSE.

UNHIDABLE attribute for image and labels

Image and labels now support the UNHIDABLE attribute for table columns.

TERMINATE REPORT / EXIT REPORT

New report instructions TERMINATE REPORT / EXIT REPORT. Use the EXIT REPORT statement to terminate a report within a REPORT definition. Both statements have the following effects:
- Terminate the processing of the current report.
- Delete any intermediate files or temporary tables that were created while processing the report.

TINYINT data type with SQL Server

SQL Server driver now supports the TINYINT data type.

Toolbars can be defined in forms

You can now define Toolbars in form specification files.

Topmenus can be defined in forms

You can now define Topmenus in form specification files.

Build version number

The FGL_GETVERSION() function returns the internal version number of the runtime system.

Get a help message text

The FGL_GETHELP() function returns the message text for a give help number.

Set the current row

The FGL_SET_ARR_CURR() function changes the current row in DISPLAY ARRAY or INPUT ARRAY.

Interruption handling

Users can now send an interruption request from the client to the program, to stop long running queries, reports and other BDL procedures, by testing the int_flag variable. The client is using an OOB signal.

StatusBar definition with style attribute

There is now a new window style attribute for statusbar layout specification. You can now set statusBarType attribute in the 4st style file for Windows, in order to control the display of status bars.

Field order form

New OPTIONS clause FIELD ORDER FORM provided to use the TABINDEX attribute to define the field tabbing order. FIELD ORDER FORM can also be used at the dialog level as dialog attribute.

Runtime system re-written in C

Runtime system has been re-written in pure C language, g++ 3.2 and corresponding gnu libs (libstdc++, libsupc++, ...) are no longer needed; a runner can be linked with a native cc compiler. See Installation and Setup.

Passing arrays as function parameter

Arrays can be passed as parameters, all elements are expanded.

Compiler supports now ANSI outer joins

You can now write static SQL statements using ANSI outer joins:

```
SELECT .. FROM a LEFT OUTER JOIN b ON a.key=b.key
```

Methods for StringBuffer

New methods for StringBuffer class: base.StringBuffer.replaceAt() and base.StringBuffer.insertAt().

Default items created for COMBOBOX

For COMBOBOX form items, a default ITEMS list is created by fglform when an INCLUDE list is used.

ON IDLE clause in dialogs

The ON IDLE clause can be used to execute a block of instructions after a timeout.

Order of INPUT ARRAY trigger execution

New logical order of execution for INPUT ARRAY triggers:

1. BEFORE INPUT
2. BEFORE ROW
3. BEFORE INSERT
4. BEFORE FIELD

New ui.ComboBox class

New ui.ComboBox class has been added, to configure COMBOBOX fields at runtime.

Predefined actions in lists: nextrow / prevrow

DISPLAY ARRAY and INPUT ARRAY instructions now automatically use two predefined actions nextrow and prevrow, which allow binding action views for navigation.

FOREACH infinite loop

FOREACH that raises an error no longer loops infinitely.

Record comparison

Operators equal (= or ==) and not equal (<> or !=) now can be used with records. All members will be compared. If two members are NULL the result of this member comparison results in TRUE.

ON CHANGE trigger

ON CHANGE field trigger in INPUT and INPUT ARRAY. Same as AFTER FIELD, but only fired if the value has changed.

Program icon

New image attribute in UserInterface node, for the program icon. Can be set with ui.Interface.setImage().

Form compilation warnings

New option -W for fglform to show warnings.

FORMAT attribute in LABELs

LABELs can now have a FORMAT attribute.

SQLSTATE and SQLERRMESSAGE

New SQLSTATE and SQLERRMESSAGE operators, to give SQL execution information.

Front End Function calls

You can now call predefined functions in the front-end, by using the ui.Interface.frontCall()method.
See also Front End Functions.

New ui.Form built-in class

New ui.Form built-in class to handle forms.

TABINDEX for tabbing order

New TABINDEX field attribute to define the tabbing order in forms.

LSTR operator

New LSTR operator to get a localized string by name:

```
DISPLAY LSTR("custno_comment")
```

SFMT operator

New SFMT operator to format strings with parameters:

```
DISPLAY SFMT("Could not find %1 in %2.",filename,dirname)
```

ON ROW CHANGE trigger

New ON ROW CHANGE clause in INPUT ARRAY. This trigger will be executed if at least one value in the row has been modified. The ON ROW CHANGE code is be executed just before the AFTER ROW clause.

New StringTokenizer class

The StringTokenizer class can be used to parse strings for tokens.

Faster linker

Linker is now faster when having program modules with a huge number of functions.

Global constants

CONSTANTs can now be defined as GLOBALs.

ON ACTION in MENUs

MENU instruction now supports ON ACTION clause, to write abstract menus as simple action handlers.

New Application class

The base.Application class provides an interface to the program properties.

New Channel class

New definition of the interface for Channels, now based on objects:

```
DEFINE c base.Channel
LET c = base.Channel.create()
CALL c.openFile("data.txt","r")
```

Predefined 'help' action

New 'help' predefined action, to start help viewer for HELP clauses in dialog instructions.

```
INPUT BY NAME .... HELP 12423 -- Creates action 'help'
```

Version 1.10

- Dynamic User Interface
- Interactive Instruction Extensions
- Built-in Classes
- Constant Definitions
- Extended Form Files
- Dynamic Arrays

- XML utilities
- STRING data type
- Defining MDI containers
- SCHEMA instruction

Dynamic User Interface

The Dynamic User Interface is the major new concept in Genero. It is the basement for the new graphical user interface. See Dynamic User Interface.

Interactive Instruction Extensions

Classical interactive instructions such as INPUT, INPUT ARRAY, DISPLAY ARRAY, CONSTRUCT have been extended with new control blocks and control instructions. See Interactive Instruction Extensions.

Built-in Classes

The language supports now built-in classes, a new object-oriented way to program in BDL. See Built-in Classes.

Constant Definitions

It is now possible to define constants, as in other languages. See Constant Definitions.

Extended Form Files

You can now define complex layouts with the extended PER files. See Extended Form Files.

Dynamic Arrays

The language now supports dynamic arrays with automatic memory allocation. DISPLAY ARRAY can now work in buffered mode, to avoid to load a big array when you have a lot of rows to display. See Dynamic Arrays.

XML utilities

A set of XML Utilities are provided in the runtime library as built-in classes.

STRING data type

A new STRING data type is now available, to simplify utility function coding.

Defining MDI containers

Defining Window Containers (MDI) is a simple way to group programs.

SCHEMA instruction

The new SCHEMA instruction allows you to specific a database schema without having an implicit connection when the program executes.

1.3x Migration Issues

This page describes migration issues when you are moving from version **1.2x** to version **1.3x** of Genero BDL.

Summary:

- Front-end compatibility
- HBox Tags
- HBox Tags limitations
- Elements inside HBoxes get their real sizes
- Width of ButtonEdit/DateEdit/ComboBox
- Default Sample
- SizePolicy for ComboBoxes
- Action Defaults at Form level
- MySQL 3.23 is desupported

Front-end compatibility

When migrating to a **1.3x** runtime system, you need to upgrade all front-end clients to any **1.3x** version. Front-end clients and runtime systems are compatible if the major version number and the first minor number are the same (**X.Y?**). A front-end of version **1.31** works with a **1.30** runtime system, but if you try to use a **1.20** client with a **1.30** runtime system, you will get an error message.

HBox tags

HBox Tags allow you to stack form items horizontally without being influenced by elements above or below. In an HBox there is a free mix of Form Fields, labels, and Spacer Items possible.

A typical usage of an HBox Tag is to have zip-code/city form fields side by side with predictable spacing in-between.

The "classic" layout would look like the following form definition:

```
<G "User Data(version 1.20)"                                        >
 Last Name     [l_name           ]First Name[f_name          ]
 Street        [street                                       ]
 City          [city          ]Zip Code[zip  ]
 Phone(private)[phone           ] At work [                 ]
 Code          [aa]-[ab]-[ac]
```

In the screenshot you will notice that the distance between "l_name" and "First Name" is smaller than between "First Name" and "f_name". How can this be? Two lines below there is the "zip" field which affects this distance.

If we put HBox Tags around the fields we want to group horizontally together, we get the predictable spacing between "l_name", "First Name" and "f_name".

```
<G "User Data in HBoxes stacked"                                    >
Last Name      [l_nameh          :"First Name":f_nameh         ]
Street         [streeth                                       :]
City           [cityh         :"Zip Code":ziph :              ]
Phone(private)[phoneh           :"At work":phonewh           : ]
Code           [ba:"-":bb:"-":bc:                             ]
```

Here "l_nameh","First Name" and "f_nameh" are together in one HBox; the ":" colon acts as a separator between the 3 elements.

The width of an element is calculated from the space between "[" and ":" (width of cityh is 14), or from the space between ":" and ":" (width of "bb" is 2), or from the space between ":" and "]" (width of "f_nameh" is 16). The "zip" field in the version 1.20 example has a width of five and the "ziph" field has also a width of five.

In the second Groupbox in the screenshot you will notice that the HBox is smaller than the first one, even though it uses two characters more in the screen definition. The reason is that each HBox occupies only ONE cell in the parent grid, and the content in one HBox is independent of the content in another HBox. This relaxes the parent grid; it has to align only the edges of the HBoxes and the labels left of the HBoxes. The two extra characters in the Form file for the second Group come from the fact that the labels need quoting to distinguish them from field definitions. Of course, you could use a Label field if the two extra characters are unwanted (which is done in the third Groupbox).

The third Groupbox shows how the alignment in an HBox can be affected by putting empty elements (: :) inside the HBox Tag:

```
<G "User Data in HBoxes right part right aligned"                   >
Last Name      [l_nameh2         : :lfirsth2:f_nameh2          ]
Street         [streeth2                                       ]
City           [cityh2        :                    :lzip:ziph2]
Phone(private)[phoneh2          :     :latw:phonewh2          ]
Code           [ca:   "-"        :cb:       "-"           :cc]
```

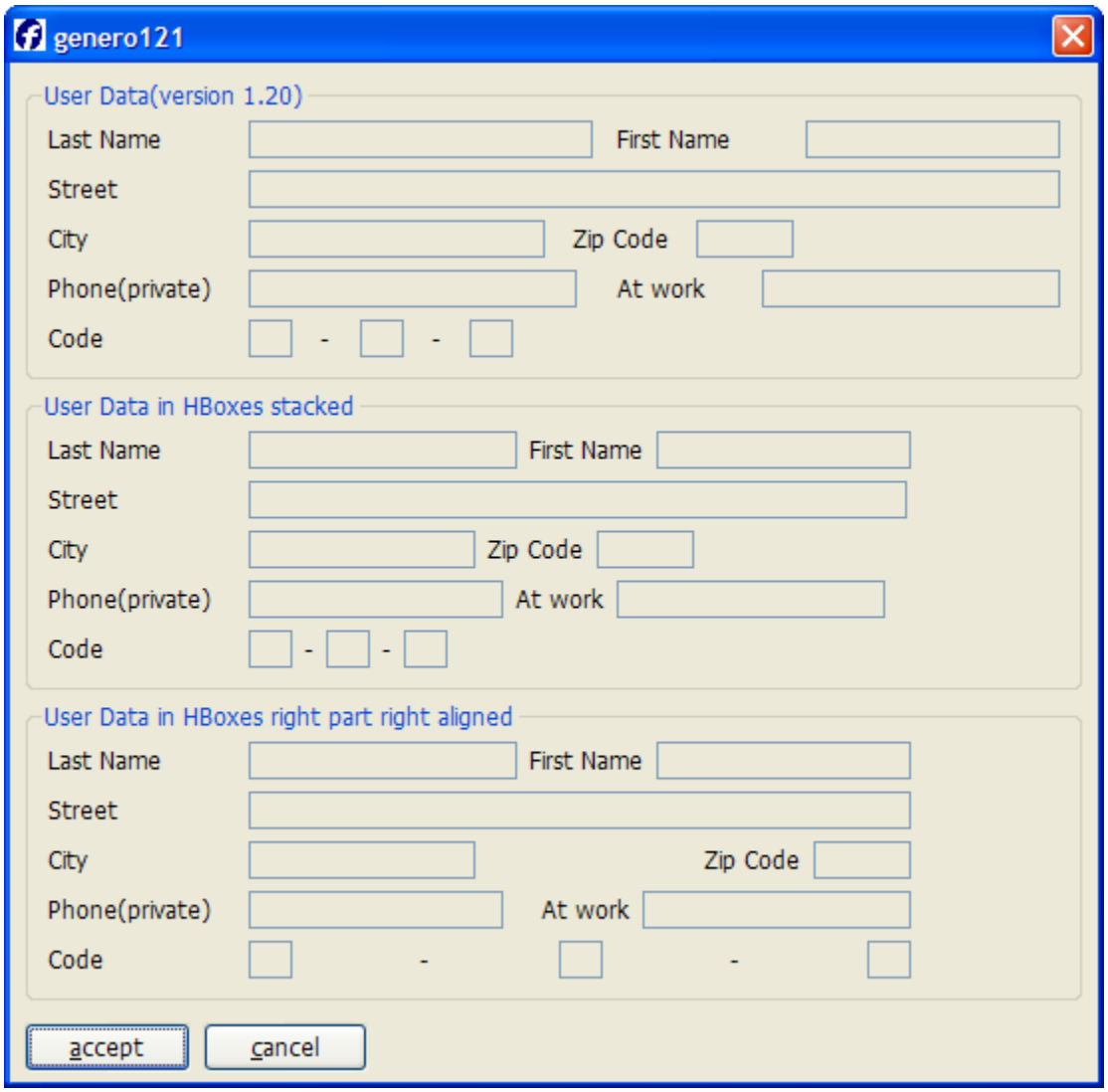
Between "l_nameh2" and "lfirsth2" there are two ":" signs with a white space between them. This means: put a Spacer Item between l_nameh2 and lfirsth2, which gets all the additional space if the HBox is bigger than the sum of l_nameh2, lfirsth2 and f_nameh2. The number of spaces, however, has no effect. The spacer item between cityh2 and lzip has the same force as the spacer between l_nameh2 and lfirsth2.

You can treat a spacer item like a spring. The spacer item between cityh2 and lzip presses cityh2 to the left-hand side, and the rest of the fields to the right-hand side. In the "Code" line there is more than one spacer item; they share the additional space

among them. (The "Code" HBox sample in the third line is only to show how spacer items work; we always advise using "Code" as in the second Groupbox, or to use a picture)

In general we advise using the approach shown in the second Groupbox: stack the items horizontally by replacing field ends with ":". This is the easy way to remove unwanted horizontal spacing.

The resulting screenshot:



HBox Tags limitations

- HBox Tags don't work for fields of Screen Arrays or Tables; you will get a form compiler error. The reason is that the current AUI structure does not allow this.

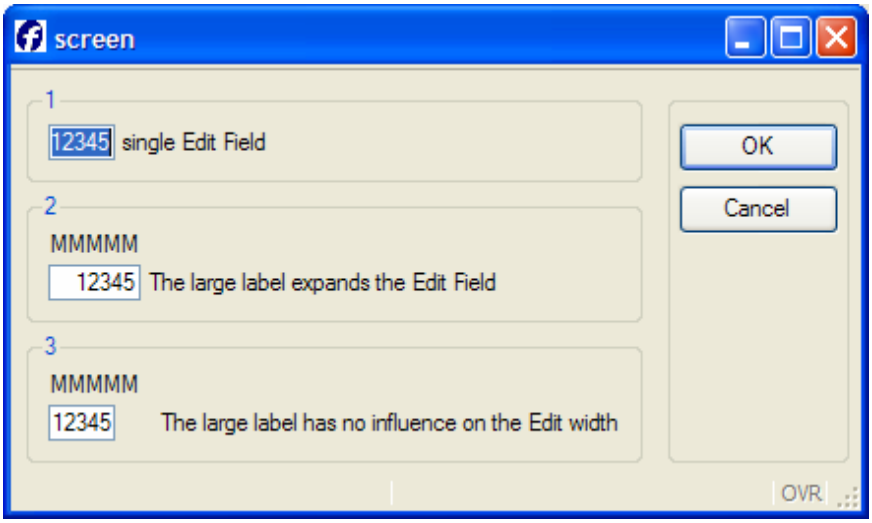
The front end needs a `Matrix` element directly in a `Grid` or a `ScrollGrid` to perform the necessary positioning calculations for the individual fields.

Elements inside HBoxes get their real sizes

A big advantage in using elements in an HBox is that the fields get their real sizes according to the .per definition.

```
LAYOUT
GRID
{
<G g1 >
[a    ] single Edit Field

<G g2 >
 MMMMM
[b    ] The large label expands the Edit Field

<G g3 >
 MMMMM
[c   :]The large label has no influence on the Edit width

}
END
END
ATTRIBUTES
EDIT a = formonly.a, sample="0", default="12345";
EDIT b = formonly.b, sample="0", default="12345";
EDIT c = formonly.c, sample="0", default="12345";
END
```

In the second Groupbox, the edit field is expanded to be as large as the label above; using an HBox prevents this:



Note: in this example, we use a sample of "0" to display *exactly* five numbers.

Width of ButtonEdit/DateEdit/ComboBox

The problem with BUTTONEDIT, DATEEDIT and COMBOBOX in previous versions is that a field `[b  ]` got the width 3, the same width as an edit field with the same layout.

For example:

```
LAYOUT
GRID
{
  [e  ]
  [b  ]
}
END
END
ATTRIBUTES
EDIT e=formonly.e;
BUTTONEDIT b=formonly.b;
END
```

In this example, the outer (visual) width of both elements was the same, but the edit portion of "b" was much smaller, because the button did not count at all. (In practice this meant that on average only one and a half characters of "b" was visible). However, you could input 3 characters! This made a `BUTTONEDIT` where you could see only one character and input only one character without tricks impossible.

Now, for the Button, the Form Compiler subtracts two character positions from the width of `BUTTONEDIT`/`COMBOBOX`/`DATEEDIT`. This is possible because now the form compiler differentiates the width of the widget from the width of the entry part.

In fact, there is no *visual* difference between version 1.20 and 1.30 regarding this example, but in version 1.30 you can only enter one character, which is visually more correct.

In the example the `BUTTONEDIT` aligns with the Edit; that's why the Edit part of the `BUTTONEDIT` is usually still a bit bigger than one character (this depends on the button size, but if a button edit is contained by an HBox, it will get the exact size of "width" multiplied by the average character pixel width.

To express the `BUTTONEDIT`/`COMBOBOX`/`DATEEDIT` layout more visually, it is possible to specify:

```
  [e  ]
  [b- ]
```

the "-" sign marks the end of the edit portion and the beginning of the button portion (edit width ="1", widget width ="3").

The two characters are also subtracted for a BUTTONEDIT which is child of an HBox.

```
[b  :]
```

gets also width="1" , but no widget width, because the HBox stacks the elements horizontally without needing widget width definition.

The two extra characters are only used to show the real size relations more WYSIWYG, and to have the same calculation as in a field without an HBox parent.

```
[e1:e2:e3:     ]
[b1  :b2  :b3  ]
```

shows that three BUTTONEDIT fields are much larger than three EDIT fields with the same width.

You can even write:

```
[e1:e2:e3:     ]
[b1- :b2- :b3- ]
```

or:

```
[e1:e2:e3:  ]
[b1-:b2-:b3-]
```

to use slim buttons and

```
[e1:e2:e3:        ]
[b1-  :b2-  :b3-  ]
```

if one uses large buttons to get the maximum WYSIWYG effect.

Please note that buttons do not grow if two characters "- " is expanded to three characters "- "; the button always computes its size from the image used, it's just to reserve more space in the form to match the real size.

Default Sample

If no SAMPLE attribute is specified in the form files, the client uses an algorithm to compute the field width. In this case, a very pessimistic algorithm is used to compute the field widths: The client assumes a default SAMPLE of "M" for the first six characters and then "0" for the subsequent characters and applies this algorithm to all fields, except some field types like DATEEDIT fields.

The default algorithm tends to produce larger forms compared to forms used in BDL V3 and very first versions of Genero. Do not hesitate to modify the SAMPLE attribute in the form file, to make your fields shorter.

If you do not want to touch all your forms, a more tailored automatic solution would be to specify a ui.form.setDefaultInitiallizer() function, to set the SAMPLE depending on the AUI tag. In the following example small `UPSHIFT` fields get a sample of "M"; all other fields get a sample of "0". This will preserve the original width for `UPSHIFT` fields, however numeric and normal String fields will get the sample of "0" and make the overall width of the form smaller.

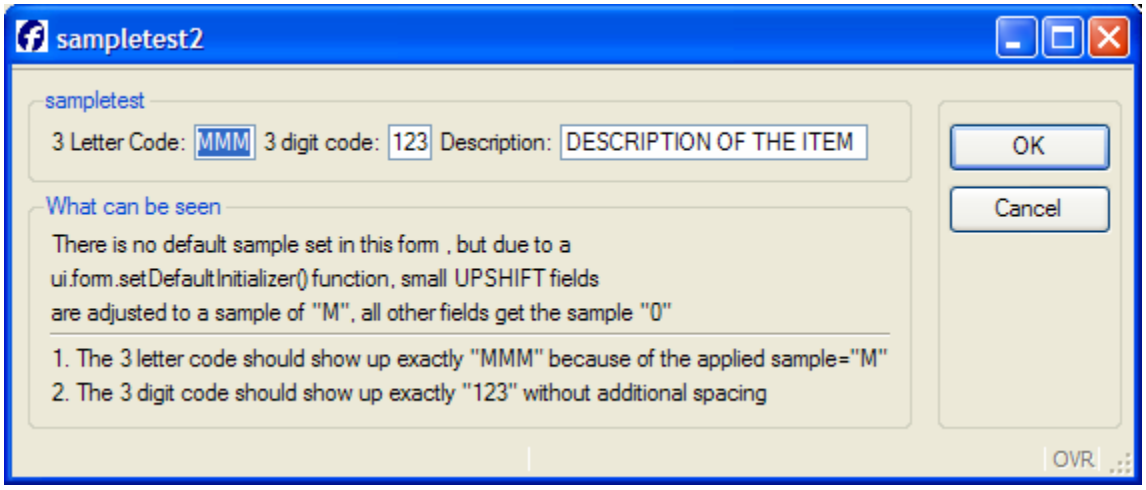
Program:

```
# this demo program shows how to affect the "sample" attribute in a
# ui.form.setDefaultInitializer function
# the main concern is to set a default sample of "0" and to
# correct the sample attribute for small UPSHIFT fields to "M"
# to be able to display full uppercase letter for fields with a small
width

MAIN
  DEFINE three_char_upshift CHAR(3)
  DEFINE three_digit_number Integer
  DEFINE longstring CHAR(100)
  CALL ui.form.setDefaultInitializer("myinit")
  OPEN form f from "sampletest2"
  DISPLAY form f
  INPUT BY NAME three_char_upshift,three_digit_number,longstring
END MAIN

FUNCTION myInit(f)
  DEFINE f ui.Form
  CALL checkSampleRecursive(f.getNode())
END FUNCTION

FUNCTION checkSampleRecursive(node)
  DEFINE node,child om.DomNode
  LET child= node.getFirstChild()
  WHILE child IS NOT NULL
    CALL checkSampleRecursive(child)
    CALL setSample(child)
    LET child=child.getNext()
  END WHILE
END FUNCTION

FUNCTION setSample(node)
  DEFINE node,parent om.DomNode
  LET parent=node.getParent()
  -- only set the "sample" for FormFields in this example
  IF parent.getTagName()<>"FormField" THEN
    RETURN
  END IF
  IF node.getAttribute("shift")="up" AND node.getAttribute("width")<=6
THEN
    CALL node.setAttribute("sample","M")
```

```
  ELSE
    CALL node.setAttribute("sample","0")
  END IF
  DISPLAY "set sample attribute of ",node.getId()," to
\"",node.getAttribute("sample"),"\""
END FUNCTION
```

Form File:

```
LAYOUT(text="sampletest2")
GRID
{
 <G
sampletest
         >
  3 Letter Code: [a ] 3 digit code:[b ] Description:[longstring ]

 <G "What can be
seen"                                                              >
  There is no default sample set in this form, but due to a
  ui.form.setDefaultInitializer function, small UPSHIFT fields
  are adjusted to a sample of "M", all other fields get the sample "0"
  ------------------------------------------------------------------
--------------
  1. The 3 letter code should show up exactly "MMM" because of the
applied sample="M"
  2. The 3 letter digit code should show up exactly "123" without
additional  spacing
}
END
END
ATTRIBUTES
EDIT a=formonly.three_char_upshift,UPSHIFT,default="MMM";
EDIT b=formonly.three_digit_number,default="123";
EDIT longstring=formonly.longstring,UPSHIFT,default="DESCRIPTION OF THE
ITEM",SCROLL;
END
```



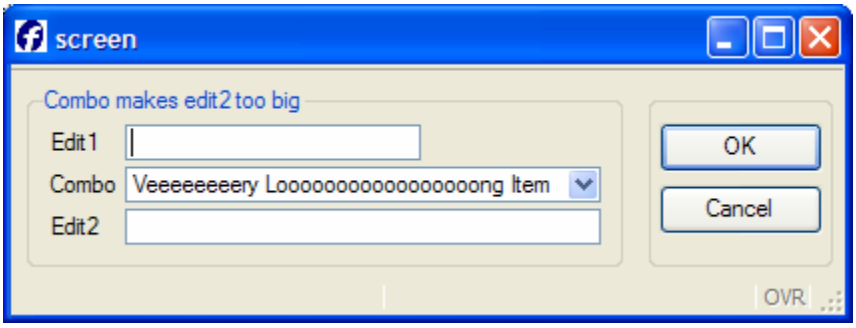
Please refer to the SAMPLE documentation for more information.

Size Policy for comboboxes

COMBOBOX items were VERY special in previous versions because they adapted their size to the maximum item of the value list. On one hand, this is very convenient because the programmer doesn't have to find the biggest string in the value list, and to estimate how large it will be on the screen (with proportional fonts the string with the highest number of characters is not automatically the largest string). On the other hand, this behavior often led to an unpredictable layout if the programmer didn't reserve enough space for the COMBOBOX.

The SIZEPOLICY attribute gives better control of the result.

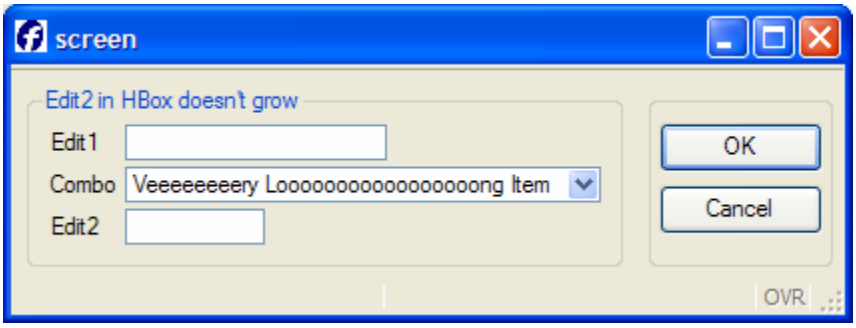
```
<G "Combo makes edit2 too big" >
 [edit1]
 [combo   ]
 [edit2   ]
...
ATTRIBUTES
EDIT edit1=formonly.edit1;
COMBOBOX combo=formonly.combo,
   ITEMS=((0,"Veeeeeeeery Looooooooooooooooong Item"),(1,"hallo")),
   DEFAULT=0;
EDIT edit2=formonly.edit2;
END
```



In this case, the "combo" field gets very large as does "edit2", because it ends in the same grid column. It will confuse the end user if he can input only eight characters and the field is apparently much bigger. Two possibilities exist to surround this:

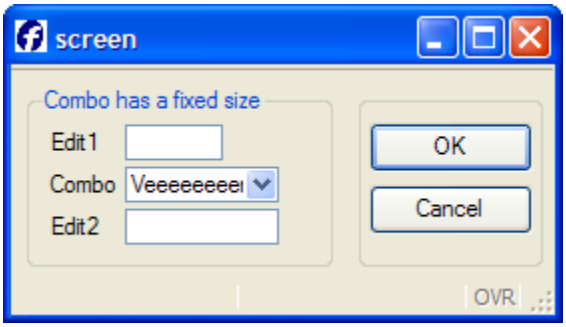
Use an HBox to prevent the edit2 from growing, and use HBoxes for all fields which start together with combo and are as large or bigger than combo

```
<G "Edit2 in HBox doesn't grow" >
[edit1]
[combo   :]
[edit2   :]
...
```



Use the new SIZEPOLICY attribute, and set it to `fixed` to prevent `combo` from getting bigger than the initial six characters (6+Button)

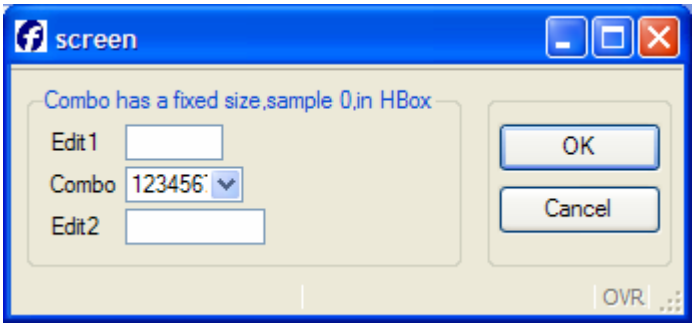
```
<G "Combo has a fixed size" >
...
[combo    ]
[edit2    ]
...
ATTRIBUTES
...
COMBOBOX combo=formonly.combo,
   ITEMS = ((0,"Veeeeeeeery Looooooooooooooooong Item"),(1,"hallo")),
   DEFAULT=0, SIZEPOLOCY=FIXED ;
....
```



Note that in this example the `edit2` dictates the maximum size of `combo`, because even if the SIZEPOLICY is *fixed*, the elements are aligned by the Grid.

To prevent this and have *exactly* six characters (numbers) in the ComboBox, you need to decouple `combo` from `edit2` by using an HBox.

```
<G "Combo has a fixed size,sample 0,in HBox" >
...
Combo [combo    :]
Edit2 [edit2    :]
...
COMBOBOX combo=formonly.combo,
   ITEMS = ((0,"12345678 Looooooooooooooooong Item"),(1,"hallo")),
   DEFAULT=0, SIZEPOLICY=FIXED, SAMPLE="0";
```



Now the wanted six numbers are displayed and `combo` does not grow to the size of `edit2`.

Please refer to the SIZEPOLICY documentation for more information.

Action Defaults at Form level

It is now possible to define action defaults in forms. In previous versions you had to define a global action default file; this works for defining common global action attributes, but there is a need to define specific action attributes in some forms. A typical zoom window may have search and navigation actions, while data input windows need to define add/delete/update actions instead.

It is now possible to define an action default section in the form file, and you can also load action defaults with ui.Form.loadActionDefaults().

Tips:

1. Use the preprocessor to include action default sections in your forms.

MySQL 3.23 is de-supported

Version **1.32** supports **MySQL 3.23.x**, but there is no support for recent MySQL versions.

Version **1.33** now de-supports MySQL **3.23.x**, but supports **MySQL 4.1.2** and higher (**5.0**).

For technical reasons, MySQL **4.0.x** cannot be supported.

2.0x Migration Issues

This page describes migration issues when you are moving from version **1.xx** to version **2.0x** of Genero BDL.

Summary:

1. De-supported platforms
2. fglmkrtm installation tool removed
3. fglinstall installation tool removed
4. Runner creation is no longer needed
5. Linking the utility functions library
6. Static C Extensions are de-supported
7. Dynamic C Extensions usage changes
8. WantColumnsAnchored attribute is de-supported
9. PixelWidth / PixelHeight attributes are de-supported
10. Pre-fetch parameters with Oracle
11. PostgreSQL 7 is de-supported
12. Adabas D is de-supported
13. Preprocessor directive syntax changed
14. Static SQL cache is removed
15. Connection database schema specification
16. FGLDBSCH schema extraction tool
17. Global and module variables using the same name
18. Connection Parameters in FGLPROFILE when using Informix
19. SQL Server 7 is de-supported
20. OPEN USING followed by FOREACH
21. Inconsistent USING clauses
22. FESQLC not provided in 2.00
23. Usage of RUN IN FORM MODE
24. TTY and COLOR WHERE attribute

---

1. De-supported platforms

The following platforms are no longer supported by Genero BDL **2.0x**:

- SUN Solaris 8-bit and 32-bit.
- IBM AIX 4.3.3 and AIX 32-bit.
- SCO Unixware lower than 7.1.3.

For information about supported platforms, please refer to the Installation and Setup section of this manual.

---

2. Setup installation tool removed

The **fglmkrtm** tool has been removed from the distribution. This tool was provided in previous versions to create a fglrun runner with the correct database driver. In version **2.0x**, database drivers are now always loaded dynamically. Refer to Connections for more details about database driver configuration.

---

3. fglinstall installation tool removed

The **fglinstall** tool has been removed from the distribution. This tool was provided in previous versions to compile product message files, form files, and program modules provided in the distribution. The compiled versions of all these files are now included in the package.

---

4. Runner creation is no longer needed

In version **2.0x**, you do not need to build a runner. The architecture is now based on shared libraries (or DLLs on Windows), and the database drivers are automatically loaded according to FGLPROFILE configuration parameters.

If you have C Extensions, you must re-build them as shared libraries. Refer to C Extensions for more details.

Warning: Database vendor client libraries (libclntsh, libcli, libpq, libaodbc) must be provided as shared objects (or DLL on Windows).

---

5. Linking the utility functions library

In version **1.3x**, some utility functions (canvas draw* and database db_* functions) were linked automatically to the 42r program when using fglrun -l or fgllink. These functions are implemented in the fgldraw.4gl and fgldbutl.4gl modules, which were linked in the **libfgl.42x** library and loaded automatically at runtime by fglrun.

In version **2.0x**, all utility functions are now in the **libfgl4js.42x** library. So, if you use the draw* or db_* utility functions, you must now add the **libfgl4js.42x** library explicitly when using fglrun -l or fgllink, or you can use the fgl2p tool to link .42r programs. The fgl2p tool links the program with the **libfgl4js.42x** library by default.

Refer to Utility Functions for more details.

---

6. Static C Extensions are de-supported

In version **2.0x**, Static C Extensions must be re-written as Dynamic C Extensions.

Refer to C Extensions for more details.

---

7. Dynamic C Extensions usage changes

In version **1.3x**, you must use FGLPROFILE entries to specify Dynamic C Extensions to be loaded at runtime.

In version **2.0x**, Dynamic C Extensions are automatically loaded according to IMPORT instructions. The FGLPROFILE entries are no longer used.

Warning: Global variables (userData) can no longer be shared between the runtime system and the C extensions. You must use functions to pass global variable values.

There is no longer a need to define the FGL_API_MAIN macro in the extension interface file.

All C data type definitions are now centralized in the fglExt.h header file, header files like Date.h, MyDecimal.h have been removed from the distribution.

Refer to C Extensions for more details.

---

8. WantColumnAnchored attribute is de-supported

In version **1.3x**, the `WANTCOLUMNSANCHORED` attribute was undocumented but still supported by the language, to simplify migration from **1.20**.

In version **2.0x**, the `WANTCOLUMNSANCHORED` attribute is de-supported; you must use UNMOVABLECOLUMNS to specify that table columns cannot be moved around by the user.

---

9. PixelWidth / PixelHeight attribute is de-supported

In version **1.3x**, the `PIXELWIDTH` and `PIXELHEIGHT` attributes were used to specify the real size of an IMAGE form item.

In version **2.0x**, you must use the WIDTH and HEIGHT attributes to specify the size of an image:

In the **.per** form file:

```
01 IMAGE img1 = FORMONLY.image1, HEIGHT = 100 PIXELS, WIDTH = 100
PIXELS;
```

The `PIXELWIDTH` and `PIXELHEIGHT` attributes are still supported by the form compiler, but are deprecated and will be removed in a future version.

---

10. Pre-fetch parameters with Oracle

Pre-fetch parameters allow an application to automatically fetch rows from the Oracle database when opening a cursor.

In version **1.3x**, the default pre-fetch parameters are 50 rows and 65535 bytes for the pre-fetch buffer. Some customers experienced a huge memory usage with those default values, when using a lot of cursors: It appears that the Oracle client is allocating a buffer of pre-fetch.memory (i.e. 64 Kbytes) for each cursor.

In version **2.0x**, the default is 10 rows and 0 (zero) bytes for the pre-fetch buffer (memory), meaning that memory is not included in computing the number of rows to pre-fetch.

For more details, refer to Connections.

---

11. PostgreSQL 7 is de-supported

Version **1.3x** supports **PostgreSQL 7.1.x**, **7.2.x**, **7.3.x**, **7.4.x**.

Version **2.0x** now de-supports all PostgreSQL versions **7**, but supports **PostgreSQL 8** and higher.

Remark: **PostgreSQL 8** is available on Windows platforms.

---

12. Adabas D is de-supported

Version **2.0x** no longer supports **Adabas D 12**.

---

13. Preprocessor directive syntax changed

In version **1.3x**, the preprocessor directives start with a (#) sharp character, to be compliant with standard preprocessors (like **cpp**). This caused too many conflicts with standard language comments that use the same character:

```
01 #include "myheader.4gl"
02 # This is a comment
```

In version **2.0x**, the preprocessor directives start with an ampersand character (&):

```
01 &include "myheader.4gl"
02 FUNCTION debug( msg )
03   DEFINE msg STRING
04 &ifdef DEBUG
05   DISPLAY msg
06 &endif
07 END FUNCTION
```

The preprocessor is now integrated in the compiler, to achieve faster compilation.

Warning: To simplify the migration, the # sharp character is still supported when using the -p fglpp option of compiler. However, you should review your source code and use the & character instead; # sharp will be de-supported in a future version.

---

14. Static SQL cache is removed

In version **1.3x**, the size of the static SQL cache is defined by a FGLPROFILE entry:

```
dbi.sql.static.optimization.cache.size = max
```

This entry was provided to optimize SQL execution without touching code using a lot of static SQL statements, especially when using non-Informix databases where the execution of static SQL statements is slower than with Informix. This is useful for fast migrations, but there were a lot of side effects and unexpected errors (refer to "Connections" in **1.3x** documentation for more details).

In version **2.0x**, the Static SQL Cache has been removed for the reasons described above. Programs continue to run without changing the code, but if you want to optimize program execution, you must use Dynamic SQL (PREPARE + EXECUTE) as described in "SQL Programming".

---

15. Connection database schema specification

Version **1.3x** has an FGLPROFILE entry to specify the database schema at runtime:

```
dbi.database.dbname.schema = "schema-name"
```

This entry could be used to select the native database schema after connecting to the server, for Oracle and Db2 only.

In version **2.0x**, this entry is now specific to the Oracle and Db2 database driver configuration parameters:

```
dbi.database.dbname.ora.schema = "schema-name"
dbi.database.dbname.db2.schema = "schema-name"
```

For other database servers, this configuration parameter is not defined.

Warning: It is no longer possible to specify the "schema" parameter in the connection string (dbname+schema='name').

For more details, refer to Connections.

---

16. FGLDBSCH schema extraction tool

Unique tool

Version **1.3x** provides two schema extractors: fglschema and fgldbsch. The first can only extract schemas from Informix databases, while the second one can extract schemas from all supported databases.

In version **2.0x**, the fgldbsch tool has been extended to support the old fglschema options, and fglschema has been replaced by a simple script calling fgldbsch. When you call fglschema, you actually call fgldbsch. We recommend that you use fgldbsch with its specific options.

System tables

In 2.0x, fgldbsch does not extract system tables by default. You must specify the -st option to get the system tables description in the schema files.

Remote synonyms

The original fglschema tool was searching for *remote synonyms* with Informix databases. The fgldbsch tool of **2.0x** does not search for remote synonyms.

Public and private synonyms

Since bug fix **#5021** (build **620.313**), fgldbsch does not extract *private synonyms* anymore. Only *public synonyms* are extracted. The **.sch** schema files do not contain table owners, and if two *private synonyms* have the same names, there is no way to distinguish them in the schema files. Therefore, to avoid any mistakes, *private synonyms* are not extracted anymore.

See also: Database Schema.

---

17. Global and module variables using the same name

An important compiler bug has been fixed in **2.0x**. This bug is referenced as **#5752**: When you declare a module variable with the same name as a global variable, a compilation error must be thrown.

This is critical to avoid confusion with the variable usage:

```
01 GLOBALS
02   DEFINE level INTEGER
03 END GLOBALS
01 GLOBALS "globals.4gl"
02 DEFINE level INTEGER
03 FUNCTION func1()
04   LET level = 123  -- is this the global or the module variable?
05 END FUNCTION
```

In version **1.3x**, the compiler did not detect this and the module variable was used, but one might want to use the global variable instead!

If you have module variables defined with the same name as global variables, the compiler now raises the following error:

```
-4319: The symbol 'variable-name' has been defined more than once.
```

You can easily fix this by renaming the module variable. There is no risk to do this modification, since the module variable was used in **1.3x**, not the global variable.

Remark: The compiler now also detects duplicate global variable declaration. Just remove the duplicated lines in your source.

---

18. Connection parameters in FGLPROFILE when using Informix

In version **1.3x**, the dbi.database.* connection parameters defined in FGLPROFILE are ignored by the Informix drivers.

In version **2.0x**, the dbi.database.* connection parameters defined in FGLPROFILE are used by the Informix driver, as well as other database vendor drivers. For example, if you connect to the database "stores", and you have the following entries defined, the driver tries to connect as "user1" with password "alpha":

```
dbi.database.stores.username = "user1"
dbi.database.stores.password = "alpha"
```

You typically get SQL errors -387 or -329 when the wrong database login or the wrong database name is used.

For more details, refer to Connections.

---

19. SQL Server 7 desupported

In version **2.0x**, **Microsoft SQL Server 7** is no longer supported. Microsoft support of this version ended the **first of January 2006**.

Only **SQL Server 2000** and **2005** are supported by Genero **2.0x**. You must upgrade your server to one of these versions of SQL Server.

---

20. OPEN USING followed by FOREACH

Warning: This issue applies to non-Informix databases only.

In version **1.3x**, you could use an OPEN USING instruction to specify the SQL parameters of a following FOREACH:

```
01 PREPARE st1 FROM "SELECT * FROM tab WHERE col>?"
02 DECLARE cu1 CURSOR FOR st1
03 OPEN cu1 USING var
04 FOREACH cu1 INTO rec.*
05   DISPLAY rec.*
06 END FOREACH
```

In this case, the FOREACH instruction was reusing the parameters provided in the last OPEN instruction. Supporting such a feature is complex and deters the proper improvement of the database interface and database drivers. It was provided to support compatibility with very old Informix 4GL compilers, but it is not proper SQL programming.

The database interface of version **2.0x** has been rewritten for better performance. The above usage of FOREACH is no longer supported.

To work around this issue, you can safely remove the OPEN instruction and put the USING clause in the FOREACH instruction:

```
01 PREPARE st1 FROM "SELECT * FROM tab WHERE col>?"
02 DECLARE cu1 CURSOR FOR st1
03 FOREACH cu1 USING var INTO rec.*
04   DISPLAY rec.*
05 END FOREACH
```

---

21. Inconsistent USING clauses

Warning: This issue applies to non-Informix databases only.

In version **1.3x**, it was possible to execute a prepared statement with the variable list changing at each EXECUTE statement:

```
01 DEFINE var1 DECIMAL(6,2)
02 DEFINE var2 CHAR(10)
03 DEFINE var3 DATE
04 PREPARE st1 FROM "INSERT INTO tab1 VALUES ( ?. ?, ? )"
05 EXECUTE st1 USING var1, var2, var3
06 EXECUTE st1 USING var2, var3, var1   -- different order = different
data types
```

The database interface of version **2.0x** has been rewritten for better performance. Having data types changing at each execute is no longer supported.

Error **-254** will be raised if different data types are used in subsequent EXECUTE statements (with the same statement name).

---

22. FESQLC not provided in 2.00

Genero FGL version **1.33** has included an ESQL/C preprocessor named FESQLC. This component provides both SQL support and C API functions to manipulate complex types as dec_t.

Because of runtime system architecture changes, the FESQLC preprocessor could not be shipped with **2.00**. However, version **2.01** provides again the FESQLC tool. Note also that version **2.00** implements Informix specific C API functions, to deploy your application without the Informix client software. For more details about the supported C API functions, have a look at the C Extensions page.

---

23. Usage of RUN IN FORM MODE

In version **1.3x**, RUN...IN FORM MODE was recommended to run interactive applications.

In version **2.0x**, RUN ... IN LINE MODE is recommended to run interactive applications. The RUN command should be used as follows (in both GUI and TUI mode):

1. When starting an interactive program, either use RUN ... IN LINE MODE or, if the default mode is LINE MODE, use the RUN instruction without any option.
2. When starting a batch program that does not display any message, you should use RUN ... IN FORM MODE.

For more details about the RUN options, see the RUN instruction.

---

24. TTY and COLOR WHERE attribute

In version **1.3x**, only some field types like EDIT or TEXTEDIT could support TTY attributes (COLOR, REVERSE), and the conditional COLOR WHERE attribute.

In version **2.0x**, all type of fields now allow TTY attributes and the conditional COLOR WHERE attribute. So when using any ATTRIBUTE(<tty-attribute>) in programs, all fields will now be affected. For example, CHECKBOX and RADIOGROUP fields will now get a colored background, while in **1.3x** it was not the case.

2.1x Migration Issues

This page describes migration issues when you are moving from version **2.0x** to version **2.1x** of Genero BDL.

Summary:

1. De-supported platforms
2. New firstrow/lastrow implicit actions

---

1. De-supported platforms

The following platforms are no longer supported by Genero BDL **2.1x**:

- AIX 32 bit (all versions)
- HP/UX 32 bit (all versions)
- SCO Open Server lower than 5.0.7.
- Microsoft Windows Visual C++ 6 (de-supported by Microsoft since September 2005).

For information about supported platforms, please refer to the Installation and Setup section of this manual.

---

2. New firstrow/lastrow implicit actions

In prior versions, **firstrow** and **lastrow** actions were handled by the front-ends as local actions. The firstrow and lastrow actions had the Home / End accelerators defined in the default.4ad file. But these accelerators conflict with the Home/End field editor accelerators if the controller is an INPUT ARRAY. The conflict was handled by the front-ends. If the current dialog was a `DISPLAY ARRAY`, the front-end used Home/End as navigation accelerators to move to the first or last row; when the current dialog was an `INPUT ARRAY`, the front-end used Home/End as local text editor shortcuts to move to the beginning or to the end of the text in the current field.

Version 2.10 now defines the firstrow and lastrow actions as server-side predefined actions when using DISPLAY ARRAY or INPUT ARRAY. The default accelerators in the FGLDIR/lib/defaults.4ad Action Defaults file are now Control-Home + Home for firstrow and Control-End + End for lastrow. If the current widget is a text editor, the editor or navigation accelerators (like Home and End) take precedence over the action accelerators. This way, during an INPUT ARRAY, Home and End will be used as editor accelerators.

If you have defined your own default.4ad file, you have probably kept the original defaults for firstrow and lastrow actions as the accelerators Home and End. In this case,

only Home and End accelerators are defined for firstrow and lastrow actions. As a result, the user cannot move to the first row or last row with Control-Home or Control-End during `INPUT ARRAY`. To solve this problem, define the same accelerators for firstrow and lastrow actions as in FGLDIR/lib/default.4ad.

You can use the Style "localAccelerators" to define how the field editor must behave. Set to "yes" (by default), the local accelerators will be used (for instance, the Home key will move the cursor to the beginning of the field). Set to "no", the action accelerators will have higher priority (The Home key will change the current row to the first row).

Data Types

The data types supported by the language:

Data Type	Description
String Data Types	
CHAR	Fixed size character strings
VARCHAR	Variable size character strings
STRING	Dynamic size character strings
Date and Datetime Data Types	
DATE	Simple calendar dates
DATETIME	High precision date and hour data
INTERVAL	High precision time intervals
Numeric Data Types	
INTEGER	4 byte integer
SMALLINT	2 byte integer
FLOAT	8 byte floating point decimal
SMALLFLOAT	4 byte floating point decimal
DECIMAL	High precision decimals
MONEY	High precision decimals with currency formatting
Large Data Types	
BYTE	Large binary data (images)
TEXT	Large text data (documents)

See also: Data Conversions, Variables, Programs.

CHAR data type

Purpose:

The `CHAR` data type is a fixed-length character string data type.

Syntax:

```
CHAR[ACTER] [ (size) ]
```

Notes:

1. `CHAR` and `CHARACTER` are synonyms.

2. *size* defines the length of the variable; the number of bytes allocated for the variable. The upper limit is **65534**.
3. When *size* is not specified, the default length is 1 character.

Usage:

CHAR variables are initialized to NULL in functions, modules and globals.

The *size* defines the number of **bytes** the variable can store. It is important to distinguish bytes from characters, because in a multi-byte character set, one character may be encoded on several bytes. For example, in the ISO-8859-1 character set, "forêt" uses 5 bytes, while in the UTF-8 multi-byte character set, the same word occupies 6 bytes, because the "ê" letter is coded with two bytes.

CHAR variables are always filled with trailing blanks, but the trailing blanks are not significant in comparisons:

```
01 MAIN
02   DEFINE c CHAR(10)
03   LET c = "abcdef"
04   DISPLAY "[", c ,"]"      -- displays [abcdef    ]
05   IF c == "abcdef" THEN    -- this is TRUE
06      DISPLAY "equals"
07   END IF
08 END MAIN
```

VARCHAR data type

Purpose:

The VARCHAR data type is a variable-length character string data type, with a maximum size.

Syntax:

```
VARCHAR [ ( size, [ reserve ] ) ]
```

Notes:

1. The *size* defines the maximum length of the variable; the maximum number of bytes the variable can store. The upper limit is **65534**.
2. *reserve* is not used, however its inclusion in the syntax for a VARCHAR variable is permitted for compatibility with the SQL data type.
3. When *size* is not specified, the default length is 1 character.

Usage:

VARCHAR variables are initialized to NULL in functions, modules and globals.

The *size* defines the maximum number of **bytes** the variable can store. It is important to distinguish bytes from characters, because in a multi-byte character set, one character may be encoded on several bytes. For example, in the ISO-8859-1 character set, "forêt" uses 5 bytes, while in the UTF-8 multi-byte character set, the same word occupies 6 bytes, because the "ê" letter is coded with two bytes.

VARCHAR variables store trailing blanks (i.e. "abc " is different from "abc"). Trailing blanks are displayed or printed in reports, but they are not significant in comparisons:

```
01 MAIN
02   DEFINE vc VARCHAR(10)
03   LET vc = "abc  "          -- two trailing blanks
04   DISPLAY "[", vc ,"]"      -- displays [abc  ]
05   IF vc == "abc" THEN       -- this is TRUE
06      DISPLAY "equals"
07   END IF
08 END MAIN
```

When you insert character data from VARCHAR variables into VARCHAR columns in a database table, the trailing blanks are kept. Likewise, when you fetch VARCHAR column values into VARCHAR variables, trailing blanks are kept.

```
01 MAIN
02   DEFINE vc VARCHAR(10)
03   DATABASE test1
04   CREATE TABLE table1 ( k INT, x VARCHAR(10) )
05   LET vc = "abc  "          -- two trailing blanks
06   INSERT INTO table1 VALUES ( 1, vc )
07   SELECT x INTO vc FROM table1 WHERE k = 1
08   DISPLAY "[", vc ,"]"      -- displays [abc  ]
09 END MAIN
```

Warning: In SQL statements, the behavior of the comparison operators when using VARCHAR values differs from one database to the other. Informix is ignoring trailing blanks, but most other databases take trailing blanks of VARCHAR values into account. See SQL Programming for more details.

STRING data type

Purpose:

The STRING data type is a variable-length, dynamically allocated character string data type, without limitation.

Syntax:

```
STRING
```

Usage:

The behavior of a `STRING` variable is similar to the `VARCHAR` data type. For example, as `VARCHAR` variables, `STRING` variables have significant trailing blanks (i.e. `"abc  "` is different from `"abc"`). There is no size limitation, it depends on available memory.

`STRING` variables are initialized to NULL in functions, modules and globals.

The `STRING` data type is typically used to implement utility functions manipulating character string with unknown size. It cannot be used to store SQL character string data, because databases have rules that need a maximum size as for `CHAR` and `VARCHAR` types.

Warning: The `STRING` data type cannot be used as SQL parameter of fetch buffer, not can it be used as form field.

Variables declared with the `STRING` data type can use built-in class methods such as `getLength()` or `toUpperCase()`.

Methods:

Warning: The `STRING` methods are all based on byte-semantics. In a multi-byte environment, the getLength() method returns the number of bytes, which can be different from the number of characters.

Object Methods	
Name	**Description**
`append( str STRING )` `  RETURNING STRING`	Returns a new string made by adding *str* to the end of the current string.
`equals( src STRING )` `  RETURNING INTEGER`	Returns TRUE if the string passed as parameters matches the current string. If one of the strings is NULL the method returns FALSE.
`equalsIgnoreCase( src STRING )` `  RETURNING INTEGER`	Returns TRUE if the string passed as parameters matches the current string, ignoring character case. If one of the strings is NULL the method returns FALSE.
`getCharAt( pos INTEGER )` `  RETURNING STRING`	Returns the character at the byte position *pos* (starts at 1). Returns NULL if the position does not match a valid character-byte position in the current string or if the current string is null.
`getIndexOf( str STRING, spos INTEGER )` `  RETURNING INTEGER`	Returns the position of the sub-string *str* in the current string, starting from byte position *spos*. Returns zero if the

	sub-string was not found. Returns -1 if string is NULL.
`getLength( )` `  RETURNING INTEGER`	This method counts the number of bytes, including trailing blanks. The LENGTH() built-in function ignores trailing blanks.
`subString( spos INTEGER, epos INTEGER )` `  RETURNING STRING`	Returns the sub-string starting at byte position *spos* and ending at *epos*. Returns NULL if the positions do not delimit a valid sub-string in the current string, or if the current string is null.
`toLowerCase( )` `  RETURNING STRING`	Converts the current string to lowercase. Returns NULL if the string is null.
`toUpperCase( )` `  RETURNING STRING`	Converts the current string to uppercase. Returns NULL if the string is null.
`trim( )` `  RETURNING STRING`	Removes white space characters from the beginning and end of the current string. Returns NULL if the string is null.
`trimLeft( )` `  RETURNING STRING`	Removes white space characters from the beginning of the current string. Returns NULL if the string is null.
`trimRight( )` `  RETURNING STRING`	Removes white space characters from the end of the current string. Returns NULL if the string is null.

Example:

```
01 MAIN
02   DEFINE s STRING
03   LET s = "abcdef  "
04   DISPLAY s || ". (" || s.getLength() || ")"
05   IF s.trimRight() = "abcdef" THEN
06     DISPLAY s.toUpperCase()
07   END IF
08 END MAIN
```

INTEGER data type

Purpose:

The `INTEGER` data type is used for storing large whole numbers.

Syntax:

```
INT[EGER]
```

Notes:

1. `INT` and `INTEGER` are synonyms.

Usage:

The storage of `INTEGER` variables is based on 4 bytes of signed data (= 32 bits). The value range is from -2,147,483,647 to +2,147,483,647.

`INTEGER` variables are initialized to zero in functions, modules and globals.

Warning: The value -2,147,483,648 is reserved for the representation of NULL.

Example:

```
01 MAIN
02   DEFINE i INTEGER
03   LET i = 1234567
04   DISPLAY i
05 END MAIN
```

SMALLINT data type

Purpose:

The `SMALLINT` data type is used for storing small whole numbers.

Syntax:

```
SMALLINT
```

Notes:

1. Variables are initialized to zero in functions, modules and globals.
2. `SMALLINT` values can be converted to strings.

Usage:

The storage of `SMALLINT` variables is based on 2 bytes of signed data (= 16 bits). The value range is from -32,767 to +32,767.

`SMALLINT` variables are initialized to zero in functions, modules and globals.

Warning: The value -32,768 is reserved for the representation of NULL.

Example:

```
01 MAIN
02   DEFINE i SMALLINT
03   LET i = 1234
04   DISPLAY i
05 END MAIN
```

FLOAT data type

Purpose:

The FLOAT data type stores values as double-precision floating-point binary numbers with up to 16 significant digits.

Syntax:

```
{ FLOAT | DOUBLE PRECISION } [(precision)]
```

Notes:

1. FLOAT and DOUBLE PRECISION are synonyms.
2. The *precision* can be specified but it has no effect in programs.

Usage:

The storage of FLOAT variables is based on 8 bytes of signed data (= 64 bits), this type is equivalent to the **double** data type in C.

FLOAT variables are initialized to zero in functions, modules and globals.

FLOAT values can be converted to strings according to the DBMONEY environment variable (defines the decimal separator).

Tip: This data type it is not recommended for exact decimal storage; use the DECIMAL data type instead.

SMALLFLOAT data type

Purpose:

The SMALLFLOAT data type stores values as single-precision floating-point binary numbers with up to 8 significant digits.

Syntax:

```
{ SMALLFLOAT | REAL }
```

Notes:

1. SMALLFLOAT and REAL are synonyms.
2. SMALLFLOAT values can be converted to strings according to the DBMONEY environment variable (which defines the decimal separator).

Usage:

The storage of SMALLFLOAT variables is based on 4 bytes of signed data (= 32 bits), this type is equivalent to the **float** data type in C.

SMALLFLOAT variables are initialized to zero in functions, modules and globals.

SMALLFLOAT values can be converted to strings according to the DBMONEY environment variable (defines the decimal separator).

Tip: This data type it is not recommended for exact decimal storage; use the DECIMAL data type instead.

DECIMAL data type

Purpose:

The DECIMAL data type is provided to handle large numeric values with exact decimal storage.

Syntax:

```
{ DEC[IMAL] | NUMERIC } [ ( precision[,scale] ) ]
```

Notes:

1. DEC, DECIMAL and NUMERIC are synonyms.
2. *precision* defines the number of significant digits (limit is 32, default is 16).
3. *scale* defines the number of digits to the right of the decimal point.

4. When no *scale* is specified, the data type defines a floating point number.

Usage:

The `DECIMAL` data type must be used for storing number with fractional parts that must be calculated exactly.

`DECIMAL` variables are initialized to NULL in functions, modules and globals.

The largest absolute value that a `DECIMAL(p,s)` can store without errors is $10^{p-s} - 10^{s}$. The stored value can have up to 30 significant decimal digits in its fractional part, or up to 32 digits to the left of the decimal point.

When you specify both the *precision* and *scale*, you define a decimal with a fixed point arithmetic. If the data type declaration specifies a *precision* but no *scale*, it defines a floating-point number with *precision* significant digits. If the data type declaration specifies no *precision* and *scale*, the default is `DECIMAL(16)`, a floating-point number with a precision of 16 digits.

`DECIMAL` values can be converted to strings according to the `DBMONEY` environment variable (defines the decimal separator).

Warnings:

1. In ANSI-compliant databases, DECIMAL data types do not provide floating point numbers. When you define a database column as `DECIMAL(16)`, it is equivalent to a `DECIMAL(16,0)` declaration. You should always specify the scale to avoid mistakes.
2. When the default exception handler is used, if you try to assign a value larger than the Decimal definition (for example, 12345.45 into DECIMAL(4,2)), no out of range error occurs, and the variable is assigned with NULL. If WHENEVER ANY ERROR is used, it raises error -1226. If you do not use WHENEVER ANY ERROR, the STATUS variable is not set to -1226.

Example:

```
01 MAIN
02   DEFINE d1 DECIMAL(10,4)
03   DEFINE d2 DECIMAL(10,3)
04   LET d1 = 1234.4567
05   LET d2 = d1 / 3 -- Rounds decimals to 3 digits
06   DISPLAY d1, d2
07 END MAIN
```

MONEY data type

Purpose:

The `MONEY` data type is provided to store currency amounts with exact decimal storage.

Syntax:

```
MONEY [ (precision[,scale]) ]
```

Notes:

1. *precision* defines the number of significant digits (limit is 32, default is 16).
2. *scale* defines the number of digits to the right of the decimal point.
3. When no *scale* is specified, it defaults to 2.

Usage:

The `MONEY` data type is provided to store currency amounts. Its behavior is similar to the DECIMAL data type, with some important differences:

A `MONEY` variable is displayed with the currency symbol defined in the DBMONEY environment variable.

You cannot define floating-point numbers with `MONEY`: If you do not specific the *scale* in the data type declaration, it defaults to 2. If no *precision* / *scale* parameters are specified, MONEY is interpreted as a DECIMAL(16,2).

Warning: See the DECIMAL data type.

DATE data type

Purpose:

The `DATE` data type stores calendar dates with a Year/Month/Day representation.

Syntax:

```
DATE
```

Usage:

Storage of `DATE` variables is based on a 4 byte integer representing the number of days since 1899/12/31.

Because DATE values are stored as integers, you can use them in arithmetic expressions: the difference of two dates returns the number of days. This is possible (and portable) with language arithmetic operators, but should be avoided in SQL statements, because not all databases support integer-based date arithmetic.

Data conversions, input and display of `DATE` values are ruled by environment settings, such as the DBDATE and DBCENTURY environment variables.

Several built-in functions and constants are available such as MDY() and TODAY.

`DATE` variables are initialized to **zero (=1899/12/31)** in functions, modules and globals.

Tips:

1. Integers can represent dates as a number of days starting from 1899/12/31, and can be assigned to dates. It is not recommended that you directly assign integers to dates, however, for source code readability.
2. As Date-to-String conversion is based on an environment setting, it is not recommended that you hardcode strings representing Dates.

Example:

```
01 MAIN
02   DEFINE d DATE
03   LET d = TODAY
04   DISPLAY d, "  ", d+100
05 END MAIN
```

DATETIME data type

Purpose:

The `DATETIME` data type stores date and time data with time units from the year to fractions of a second.

Syntax:

```
DATETIME qual1 TO qual2

where qual1 can be one of:

  YEAR
  MONTH
  DAY
  HOUR
  MINUTE
  SECOND
  FRACTION
```

```
and qual2 can be one of:

  YEAR
  MONTH
  DAY
  HOUR
  MINUTE
  SECOND
  FRACTION
  FRACTION(1)
  FRACTION(2)
  FRACTION(3)
  FRACTION(4)
  FRACTION(5)
```

Notes:

1. *scale* defines the number of significant digits of the fractions of a second.
2. *qual1* and *qual2* qualifiers define the precision of the `DATETIME` variable.

Usage:

The `DATETIME` data type stores an instance in time, expressed as a calendar date and time-of-day.

The *qual1* and *qual2* qualifiers define the precision of the `DATETIME` variable. The precision can range from a year through a fraction of second.

`DATETIME` arithmetic is based on the INTERVAL data type, and can be combined with DATE values:

- Datetime +/- Datetime = Interval
- Datetime +/- Interval = Datetime
- Datetime +/- Date = Interval

The CURRENT operator provides current system date/time.

You can assign `DATETIME` variables with datetime literals with a specific notation.

`DATETIME` variables are initialized to NULL in functions, modules and globals.

`DATETIME` values can be converted to strings by the ISO format (YYYY-MM-DD hh:mm:ss.fffff).

Example:

```
01 MAIN
02   DEFINE d1, d2 DATETIME YEAR TO MINUTE
03   LET d1 = CURRENT YEAR TO MINUTE
04   LET d1 = "1998-01-23 12:34"
05   DISPLAY d1, d2
```

```
06 END MAIN
```

INTERVAL data type

Purpose:

The `INTERVAL` data type stores spans of time as Year/Month or Day/Hour/Minute/Second/Fraction units.

Syntax 1: *year-month* class interval

```
 INTERVAL YEAR[(precision)]  TO MONTH
|INTERVAL YEAR[(precision)]  TO YEAR
|INTERVAL MONTH[(precision)] TO MONTH
```

Syntax 2: *day-time* class interval

```
 INTERVAL DAY[(precision)]      TO FRACTION[(scale)]
|INTERVAL DAY[(precision)]      TO SECOND
|INTERVAL DAY[(precision)]      TO MINUTE
|INTERVAL DAY[(precision)]      TO HOUR
|INTERVAL DAY[(precision)]      TO DAY

|INTERVAL HOUR[(precision)]     TO FRACTION[(scale)]
|INTERVAL HOUR[(precision)]     TO SECOND
|INTERVAL HOUR[(precision)]     TO MINUTE
|INTERVAL HOUR[(precision)]     TO HOUR

|INTERVAL MINUTE[(precision)]   TO FRACTION[(scale)]
|INTERVAL MINUTE[(precision)]   TO SECOND
|INTERVAL MINUTE[(precision)]   TO MINUTE

|INTERVAL SECOND[(precision)]   TO FRACTION[(scale)]
|INTERVAL SECOND[(precision)]   TO SECOND

|INTERVAL FRACTION[(precision)] TO FRACTION[(scale)]
```

Notes:

1. *precision* defines the number of significant digits of the first qualifier, it must be an integer from 1 to 9.
   For YEAR, the default is 4. For all other time units, the default is 2.
   For example, YEAR(5) indicates that the INTERVAL can store a number of years with up to 5 digits.

Usage:

The INTERVAL data type stores a span of time, the difference between two points in time. It can also be used to store quantities that are measured in units of time, such as ages or times required for some activity.

The INTERVAL data type falls in two classes, which are mutually exclusive:

- *Year-time* intervals store a span of years, months or both.
- *Day-time* intervals store a span of days, hours, minutes, seconds and fraction of seconds, or a contiguous subset of those units.

INTERVAL values can be negative.

INTERVAL arithmetic is possible, and can involve DATETIME values:

- Interval +/- Interval = Interval
- Datetime +/- Datetime = Interval
- Datetime +/- Interval = Datetime

You can assign INTERVAL variable with interval literals with a specific notation.

INTERVAL values can be converted to strings by using the ISO format (YYYY-MM-DD hh:mm:ss.fffff).

Example:

```
01 MAIN
02   DEFINE i1 INTERVAL YEAR TO MONTH
03   DEFINE i2 INTERVAL DAY(5) TO MINUTE
04   LET i1 = "2342-4"
05   LET i2 = "23423 12:34"
06   DISPLAY i1, i2
07 END MAIN
```

BYTE data type

Purpose:

The BYTE data type stores any type of binary data, such as images or sounds.

Syntax:

```
BYTE
```

Usage:

A BYTE variable is actually a 'locator' for a large object stored in a file or in memory. The BYTE data type is a complex type that cannot be used like simple types such as INTEGER or CHAR: It is designed to handle a large amount of unstructured binary data. This type has a theoretical limit of 2^31 bytes, but the practical limit depends from the resources available to the process. You can use the BYTE data type to store, fetch or update the contents of a BYTE database column when using Informix, or the content of a BLOB column when using another type of database.

Warning: A BYTE variable must be initialized with the LOCATE instruction before usage. You might want to free resources allocated to the BYTE variable with the FREE instruction. Note that a FREE will remove the file if the LOB variable is located in a file.

The LOCATE instruction basically defines where the large data object has to be stored (in file or memory). This instruction will actually allow you to fetch a LOB into memory or into a file, or insert a LOB from memory or from a file into the database.

Warning: When you assign a BYTE variable to another BYTE variable, the LOB data is not duplicated, only the handler is copied.

Note that if you need to clone the large object, you can use the I/O built-in methods to read/write data from/to a specific file. The large object can be located in memory or in a file.

Methods:

Object Methods	
Name	**Description**
`readFile( fileName STRING )`	Reads data from a file and copies into memory or to the file used by the variables according to the LOCATE statement issued on the object.
`writeFile( fileName STRING )`	Writes data from the variable (memory or source file) to the destination file passed as parameter. The file is created if it does not exist.

Example:

```
01 MAIN
02   DEFINE b BYTE
03   DATABASE stock
04   LOCATE b IN MEMORY
05   SELECT bytecol INTO b FROM mytable
06 END MAIN
```

TEXT data type

Purpose:

The `TEXT` data type stores large text data.

Syntax:

```
TEXT
```

Usage:

A `TEXT` variable is actually a 'locator' for a large object stored in a file or in memory. The `TEXT` data type is a complex type that cannot be used like basic character string types such as `VARCHAR` or `CHAR`: It is designed to handle a large amount of text data. You can use this data type to store, fetch or update the contents of a TEXT database column when using Informix, or the content of a CLOB column when using another type of database.

Warning: A `TEXT` variable must be initialized with the LOCATE instruction before usage. You might want to free resources allocated to the `TEXT` variable with the FREE instruction. Note that a `FREE` will remove the file if the LOB variable is located in a file.

The `LOCATE` instruction basically defines where the large data object has to be stored (in file or memory). This instruction will actually allow you to fetch a LOB into memory or into a file, or insert a LOB from memory or from a file into the database.

Warning: When you assign a `TEXT` variable to another `TEXT` variable, the LOB data is not duplicated, only the handler is copied.

You can assign `TEXT` variables to/from `VARCHAR`, `CHAR` and `STRING` variables.

Note that if you need to clone the large object, you can use the I/O built-in methods to read/write data from/to a specific file. The large object can be located in memory or in a file.

Methods:

Object Methods	
Name	**Description**
`readFile( fileName STRING )`	Reads data from a file and copies into memory or to the file used by the variables according to the LOCATE statement issued on the object.
`writeFile( fileName STRING )`	Writes data from the variable (memory or source file) to the destination file passed as parameter.

The file is created if it does not exist.

Example:

```
01 MAIN
02   DEFINE t TEXT
03   DATABASE stock
04   LOCATE t IN FILE "/tmp/mytext.txt"
05   SELECT textcol INTO t FROM mytable
06 END MAIN
```

Literals

Summary:

- Integer Literals
- Decimal Literals
- String Literals
- Datetime Literals
- Interval Literals

See also: Variables, Data Types, Expressions.

INTEGER LITERALS

Purpose:

The language supports integer literals in base-10 notation, without blank spaces and commas and without a decimal point.

Syntax:

```
[+|-] digit[...]
```

Notes:

1. *digit* is a digit character from '0' to '9'.

Warnings:

1. Integer literals are limited to the ranges of an INTEGER value.

Example:

```
01 MAIN
02   DEFINE n INTEGER
03   LET n = 1234567
04 END MAIN
```

DECIMAL LITERALS

Purpose:

The language supports decimal literals as a base-10 representation of a real number, with an optional exponent notation.

Syntax:

```
[+|-] digit[...] dot digit[...] [ {e|E} [+|-] digit[...] ]
```

Notes:

1. *dot* is the decimal separator and is always a dot, independently from DBMONEY.
2. The E character is used to specify the exponent.

Example:

```
01 MAIN
02   DEFINE n DECIMAL(10,2)
03   LET n = 12345.67
04   LET n = -1.2356e-10
05 END MAIN
```

STRING LITERALS

Purpose:

The language supports string literals delimited by single quotes or double quotes.

Syntax 1 (using double quotes):

```
" alphanum [...] "
```

Syntax 2 (using single quotes):

```
' alphanum [...] '
```

Notes:

1. A string literal defines a character string constant, following the current character set.
2. A string literal can be written on multiple lines, the compiler merges lines by removing the new-line character.
3. The escape character is the back-slash character (\).
4. Quotes can be doubled to be included in strings.

Warnings:

1. An empty string ("") is equivalent to NULL.

Escape Sequences in string literals

A string literal can hold the following escape sequences:

1. `\\` is a backslash character.
2. `\"` is a double-quote character.
3. `\'` is a single-quote character.
4. `\n` is a new-line character.
5. `\r` is a carriage-return character.
6. `\0` is a null character.
7. `\f` is a form-feed character.
8. `\t` is a tab character.
9. `\xNN` is a character defined by the hexadecimal code **NN**.

Example:

```
01 MAIN
02   DISPLAY "Some text in double quotes"
03   DISPLAY 'Some text in single quotes'
04   DISPLAY "Escaped double quotes :  \"  ""  "
05   DISPLAY 'Escaped single quotes :  \'  ''  '
06   DISPLAY 'Insert a new-line character here: \n and continue with
text.'
07   DISPLAY "This is a text
08    on multiple
09    lines.\
10    You can insert a new-line with back-slash at the end of the
line."
11   IF "" IS NULL THEN DISPLAY 'Empty string is NULL' END IF
12 END MAIN
```

DATETIME LITERALS

Purpose:

The language supports datetime literals with the DATETIME () notation.

Syntax:

```
DATETIME ( dtrep ) qual1 TO qual2[(scale)]
```

Notes:

1. *dtrep* is the datetime value representation in normalized format (YYYY-MM-DD hh:mm:ss.fffff).
2. *qual1* and *qual2* are the datetime qualifiers as described in the Datetime data type.

Example:

```
01 MAIN
02   DEFINE d1 DATETIME YEAR TO SECOND
03   DEFINE d2 DATETIME HOUR TO FRACTION(5)
```

```
04   LET d1 = DATETIME ( 2002-12-24 23:55:56 ) YEAR TO SECOND
05   LET d2 = DATETIME ( 23:44:55.34532 ) HOUR TO FRACTION(5)
06 END MAIN
```

INTERVAL LITERALS

Purpose:

The language supports interval literals with the INTERVAL() notation.

Syntax:

```
INTERVAL ( inrep ) qual1[(precision)] TO qual2[(scale)]
```

Notes:

1. *inrep* is the interval value representation in normalized format (YYYY-MM or DD hh:mm:ss.fffff).
2. *qual1* and *qual2* are the interval qualifiers as described in the Interval data type..

Example:

```
01 MAIN
02   DEFINE i1 INTERVAL YEAR TO MONTH
03   DEFINE i2 INTERVAL HOUR(5) TO SECOND
04   LET i1 = INTERVAL ( 345-5 ) YEAR TO MONTH
05   LET i2 = INTERVAL ( 34562:12:33 ) HOUR(5) TO SECOND
06 END MAIN
```

Operators

Summary:

- Definition
- Operator List
- Order of Precedence List
- General Warnings

See also: Variables, Data Types, Expressions, Literals.

Definition

The operators listed in this section can appear in Expressions. Expressions with several operators are evaluated according to their precedence, from highest to lowest, as described in the Order of Precedence List. Use parentheses to instruct the runtime system to evaluate the expression in a different way than the default order of precedence.

List of Operators

- Parentheses (`( )`)
- Associative Operators
  - Membership (`.`)
- Assignment Operators
  - Assignment (`:=`)
- Comparison Operators
  - Is Null (`IS NULL`)
  - Like (`LIKE`)
  - Matches (`MATCHES`)
  - Equal (`==`)
  - Different (`!= or <>`)
  - Lower (`<`)
  - Lower or Equal (`<=`)
  - Greater (`>`)
  - Greater or Equal (`>=`)
- Logical Operators
  - Not (`NOT`)
  - And (`AND`)
  - Or (`OR`)
- State Operators
  - SQL State Code (`SQLSTATE`)
  - SQL Error Message (`SQLERRMESSAGE`)

Numeric Operators

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)
- Exponentiation (**)
- Modulus (MOD)

- String Operators
  - ASCII Char (ASCII)
  - Concatenate (||)
  - Append (,)
  - Substring ([x,y])
  - Formatting (USING)
  - Clipped (CLIPPED)
  - Spaces (SPACES)
  - Localized String (LSTR)
  - Replace (SFMT)
- Datetime Operators
  - Current Datetime (CURRENT x TO y)
  - Datetime Extensions (EXTEND(d, x TO y))
  - Date Conversion (DATE)
  - Time Conversion (TIME)
  - Current Date (TODAY)
  - Year of Date (YEAR(d))
  - Month of Date (MONTH(d))
  - Day of Date (DAY(d))
  - Weekday of Date (WEEKDAY(d))
  - Building a Date (MDY(m,d,y))
  - Interval Unit (UNITS)
- Form Field Operators
  - Field Buffer (GET_FLDBUF)
  - Current Field (INFIELD)
  - Field Modification (FIELD_TOUCHED)

Order of Precedence List

The following list describes the precedence order of operators. The **P** column defines the precedence, from highest(14) to lowest(1). The **A** column defines the direction of associativity (L=Left, R=Right, N=None). Some operators have the same precedence.

P	Operator	A	Description	Example
14	. (period)	L	Membership	myrecord.member1
14	variable[ ]	L	Array index or character	myarray[2,x,y]

			subscripts	
14	`function( )`	N	Function call	`1 + myfunc(10,"abc")`
13	`UNITS`	L	Single-qualifier interval	`(integer) UNITS DAY`
12	`+`	R	Unary plus	`+ number`
12	`-`	R	Unary minus	`- number`
11	`**`	L	Exponentiation	`x ** 5`
11	`MOD`	L	Modulus	`x MOD 2`
10	`*`	L	Multiplication	`x * y`
10	`/`	L	Division	`x / y`
9	`+`	L	Addition	`x + y`
9	`-`	L	Subtraction	`x - y`
8	`||`	L	Concatenation	`"Amount:" || amount`
7	`LIKE`	R	String comparison	`mystring LIKE "A%"`
7	`MATCHES`	R	String comparison	`mystring MATCHES "A*"`
6	`<`	L	Less than	`var < 100`
6	`<=`	L	Less then or equal to	`var <= 100`
6	`>`	L	Greater than	`var > 100`
6	`>=`	L	Greater than or equal to	`var >= 100`
6	`==`	L	Equals	`var == 100`
6	`<> or !=`	L	Not equal to	`var <> 100`
5	`IS NULL`	L	Test for NULL	`var IS NULL`
4	`NOT`	L	Logical inverse	`NOT ( a = b )`
3	`AND`	L	Logical intersection	`expr1 AND expr2`
2	`OR`	L	Logical union	`expr1 OR expr2`
1	`ASCII( )`	R	ASCII Character	`ASCII(32)`
1	`CLIPPED`	R	Delete trailing blanks	`DISPLAY string CLIPPED`
1	`COLUMN (reports)`	R	Begin line mode display	`PRINT COLUMN 32, "a"`
1	`(integer) SPACES`	R	Insert blank spaces	`DISPLAY "a" (5) SPACES`
1	`LSTR(string)`	R	Load localized string	`DISPLAY LSTR("str123")`
1	`SFMT(string [,p[...]])`	R	Parameter replacement	`DISPLAY SFMT("%1",123)`

1	SQLSTATE	R	SQL State Code	IF SQLSTATE="IX000"
1	SQLERRMESSAGE	R	SQL Error Message	DISPLAY SQLERRMESSAGE
1	USING	R	Format character string	TODAY USING "yy/mm/dd"
1	:=	L	Assignment	var := "abc"

General Warnings

Pure SQL Operators

The following operators are related to SQL syntax and not part of the language:

- BETWEEN *expr* AND *expr*
- IN (*expr* [, ..'])

Report Routine Operators

The following operators are only available in the FORMAT section of report routines:

- PAGENO
- WORDWRAP

See Report Definition for more details.

PARENTHESES

Purpose:

Parentheses are typically used to associate a set of values or expressions to override the default order of precedence.

Syntax:

(expr operator expr [...])

Notes:

1. Typically used to change the precedence of operators.

Example:

```
01 MAIN
02   DEFINE n INTEGER
03   LET n = ( ( 3 + 2 ) * 2 )
04   IF n=10 AND ( n<=0 OR n>=20 ) THEN
05     DISPLAY "OK"
06   END IF
07 END MAIN
```

MEMBERSHIP

Purpose:

The **period membership** operator specifies that its right-hand operand is a member of the set whose name is its left-hand operand.

Syntax:

```
setname.element
```

Notes:

1. Typically used for record members.

Example:

```
01 MAIN
02   DEFINE rec RECORD
03      n INTEGER,
04      c CHAR(10)
05   END RECORD
06   LET rec.n = 12345
06   LET rec.c = "abcdef"
07 END MAIN
```

ASSIGNMENT

Purpose:

The `:=` assignment operator sets a value to the left-hand operand, which must be a variable.

Syntax:

```
variable := value
```

Notes:

1. Do not confuse with the LET instruction.
2. The left-hand operand must be a variable.
3. The assignment operator can be used in expressions.
4. The assignment operator has the lowest precedence.

Example:

```
01 MAIN
02   DEFINE var1, var2 INTEGER
03   -- 1. Evaluates 2*5
04   -- 2. Sets var2 to 10
05   -- 3. Then affects var1 with 10
06   LET var1 = var2:=2*5
07 END MAIN
```

IS NULL

Purpose:

The `IS NULL` operator is provided to test whether a value is NULL.

Syntax:

```
IS NULL
```

Notes:

1. Applies to most Data Types, **except** complex types like BYTE and TEXT.

Example:

```
01 MAIN
02   DEFINE n INTEGER
03   LET n = 257
04   IF n IS NULL THEN
05      DISPLAY "Something is wrong here"
06   END IF
07 END MAIN
```

EQUAL

Purpose:

The `==` operator evaluates whether two expressions or two records are identical.

Syntax 1: Expression comparison

```
expr == expr
```

Syntax 2: Record comparison

```
record1.* == record2.*
```

Notes:

1. Syntax 1 applies to most Data Types, **except** complex types like BYTE and TEXT.
2. *expr* can be any expression supported by the language.
3. Syntax 2 allows you to compare all members of records having the same structure.
4. *record1* and *record2* are records with the same structure.
5. A single equal sign (=) can be used as an alias for this operator.

Usage:

When comparing expressions using the first syntax, the result of the operator is FALSE when one of the operands is NULL.

When comparing two records using the second syntax, the runtime system compares all corresponding members of the records. If a pair of members are different, the result of the operator is FALSE. When two corresponding members are NULL, they are considered as equal.

Example:

```
01 MAIN
02   IF 256==257 THEN
03      DISPLAY "Something is wrong here"
04   END IF
05 END MAIN
```

DIFFERENT

Purpose:

The != operator evaluates whether two expressions or two records are different.

Syntax 1: Expression comparison

```
expr != expr
```

Syntax 2: Record comparison

```
record1.* != record2.*
```

Notes:

1. Syntax 1 applies to most Data Types, **except** complex types like BYTE and TEXT.
2. *expr* can be any expression supported by the language.
3. Syntax 2 allows you to compare all members of records having the same structure.
4. *record1* and *record2* are records with the same structure.
5. An alias exists for this operator: <>

Usage:

When comparing expressions with the first syntax, the result of the operator is FALSE when one of the operands is NULL.

When comparing two records with the second syntax, the runtime system compares all corresponding members of the records. If one pair of members are different, the result of the operator is TRUE. When two corresponding members are NULL, they are considered as equal.

Example:

```
01 MAIN
02   IF 256 != 257 THEN
03      DISPLAY "This seems to be true"
04   END IF
05 END MAIN
```

LOWER

Purpose:

The < operator is provided to test whether a value or expression is lower than another.

Syntax:

```
expr < expr
```

Warnings:

1. Applies to most Data Types, **except** complex types like BYTE and TEXT.

LOWER OR EQUAL

Purpose:

The <= operator is provided to test whether a value or expression is lower than or equal to another.

Syntax:

```
expr <= expr
```

Warnings:

1. Applies to most Data Types, **except** complex types like BYTE and TEXT.

GREATER

Purpose:

The > operator is provided to test whether a value or expression is greater than another.

Syntax:

```
expr > expr
```

Warnings:

1. Applies to most Data Types, **except** complex types like BYTE and TEXT.

GREATER OR EQUAL

Purpose:

The >= operator is provided to test whether a value or expression is greater than or equal to another.

Syntax:

```
expr >= expr
```

Warnings:

1. Applies to most Data Types, **except** complex types like BYTE and TEXT.

NOT

Purpose:

The `NOT` operator is a typical logical NOT used to invert a Boolean expression.

Syntax:

```
NOT boolexpr
```

Example:

```
01 MAIN
02   IF NOT ( 256 != 257 ) THEN
03      DISPLAY "Something is wrong here"
04   END IF
05 END MAIN
```

AND

Purpose:

The `AND` operator is the logical intersection operator.

Syntax:

```
boolexpr AND boolexpr
```

Example:

```
01 MAIN
02   IF 256!=257 AND 256=257 THEN
03      DISPLAY "Sure?"
04   END IF
05 END MAIN
```

OR

Purpose:

The `OR` operator is the logical union operator.

Syntax:

```
boolexpr OR boolexpr
```

Example:

```
01 MAIN
02   IF TRUE OR FALSE THEN
03      DISPLAY "Must be true!"
04   END IF
05 END MAIN
```

SQLSTATE

Purpose:

The `SQLSTATE` operator returns the ANSI/ISO SQLSTATE code when an SQL error occurred.

Syntax:

```
SQLSTATE
```

Warnings:

1. The SQLSTATE error code is a standard ANSI specification, but not all database engines support this feature. Check the database server documentation for more details.

Example:

```
01 MAIN
02   DATABASE stores
03   WHENEVER ERROR CONTINUE
04   SELECT foo FROM bar
05   DISPLAY SQLSTATE
06 END MAIN
```

SQLERRMESSAGE

Purpose:

The `SQLERRMESSAGE` operator returns the error message if an SQL error occurred.

Syntax:

```
SQLERRMESSAGE
```

Example:

```
01 MAIN
02   DATABASE stores
03   WHENEVER ERROR CONTINUE
04   SELECT foo FROM bar
05   DISPLAY SQLERRMESSAGE
06 END MAIN
```

ASCII

Purpose:

The `ASCII` operator returns the character corresponding to the ASCII code passed as a parameter.

Syntax:

```
ASCII intexpr
```

Notes:

1. *intexpr* is an integer expression..
2. Typically used to generate a non-printable character such as NewLine or Escape.
3. In a default (U.S. English) locale, this is the logical inverse of the ORD() built-in function.

Warnings:

1. This is not a function, but a real operator (it can, for example, be used as a function parameter).

Tips:

1. Often used with parentheses (ASCII(n)), but these are not needed.

Example:

```
01 MAIN
02   DISPLAY ASCII 65, ASCII 66, ASCII 7
03 END MAIN
```

LIKE

Purpose:

The LIKE operator returns TRUE if a string matches a given mask.

Syntax:

```
expression [NOT] LIKE mask [ ESCAPE "char" ]
```

Warnings:

1. Do not confuse with the LIKE clause of the DEFINE instruction.
2. LIKE operators used in SQL statements are evaluated by the database server. This may have a different behavior than the LIKE operator of the language.

Notes:

1. *expression* is any character string expression.
2. *mask* is a character string expression defining the filter.
3. *char* is a single char specifying the escape symbol.

Usage:

The *mask* can be any combination of characters, including the % and _ wildcards:

- The % percent character matches any string of zero or more characters.
- The _ underscore character matches any single character.

The ESCAPE clause can be used to define an escape character different from the default backslash. It must be enclosed in single or double quotes.

A backslash (or the escape character specified by the ESCAPE clause) makes the operator treat the next character as a literal character, even if it is one of the special symbols in the above list. This allows you to search for %, _ or \ characters.

Example:

```
01 MAIN
02   IF "abcdef" LIKE "a%e_" THEN
03      DISPLAY "yes"
04   END IF
05 END MAIN
```

MATCHES

Purpose:

The `MATCHES` operator returns TRUE if a string matches a given mask.

Syntax:

```
expression [NOT] MATCHES mask [ ESCAPE "char" ]
```

Notes:

1. *expression* is any character string expression.
2. *mask* is a character string expression defining the filter.
3. *char* is a single char specifying the escape symbol.

Usage:

The *mask* can be any combination of characters, including the `*`, `?` and `[]` wildcards:

- The `*` star character matches any string of zero or more characters.
- The `?` question mark matches any single character.
- The `[ ]` brackets match any enclosed character. A hyphen (`-`) between characters means a range of characters. An initial caret (`^`) matches any character that is not listed.

The `ESCAPE` clause can be used to define an escape character different from the default backslash. It must be enclosed in single or double quotes.

A backslash (or the escape character specified by the `ESCAPE` clause) makes the operator treat the next character as a literal character, even if it is one of the special symbols in the above list. This allows you to search for `*`, `?` or `\` characters.

Example:

```
01 MAIN
02   IF "abcdef" NOT MATCHES "b*[a-z]" THEN
03      DISPLAY "yes"
04   END IF
05 END MAIN
```

CONCATENATE

Purpose:

The `||` operator is the concatenation operator that produces a string expression.

Syntax:

```
expr || expr
```

Notes:

1. *expr* can be a character, numeric or date time expression.
2. This operator has a high precedence; it can be used in parameters for function calls.
3. The precedence of this operator is higher than LIKE and MATCHES, but less than arithmetic operators. For example, `a || b + c` is equivalent to `(a||(b+c))`.

Warnings:

1. If any of the members of a concatenation expression is NULL, the result string will be NULL.

Example:

```
01 MAIN
02   DISPLAY "Length: " || length( "ab" || "cdef" )
03 END MAIN
```

APPEND

Purpose:

The `,` operator appends a value to a string.

Syntax:

```
charexpr , expr
```

Notes:

1. Can only be used in LET and DISPLAY instructions.
2. In earlier versions this was the only way to concatenate strings; use the `||` operator instead.

Example:

```
01 MAIN
02   DISPLAY "Today:", TODAY, " and a number: ", 12345.67
03 END MAIN
```

SUBSTRING

Purpose:

The `[]` operator is provided to extract a sub-string from a character variable.

Syntax:

```
charvar [ start [, end ] ]
```

Notes:

1. *start* defines the position of the first character of the sub-string to be extracted.
2. *end* defines the position of the last character of the sub-string to be extracted.
3. If *end* is not specified, only one character is extracted.

Warnings:

1. Sub-strings expressions in SQL statements are evaluated by the database server. This may have a different behavior than the sub-string operator of the language.

Example:

```
01 MAIN
02   DEFINE s CHAR(10)
03   LET s = "abcdef"
04   DISPLAY s[3,4]
05 END MAIN
```

USING

Purpose:

The `USING` operator converts datetime and numeric values into a string with a formatting mask.

Syntax:

```
expr USING "format"
```

Notes:

1. *format* defines the formatting mask to be used; see below for more details.

Warnings:

1. The formatting characters of USING are not identical to those that you can specify in the format strings of FORMAT and PICTURE form field attributes.

Formatting symbols for numbers:

Character	Description
*	Fills with asterisks any position that would otherwise be blank.
&	Fills with zeros any position that would otherwise be blank.
#	This does not change any blank positions in the display.
<	Causes left alignment.
, (comma)	Defines the position of the thousands separator. The thousands separator is not displayed if there are no digits to the left. By default, the thousands separator is a comma, but it can be another character as defined by DBFORMAT.
. (period)	Defines the position of the decimal separator. Only a single decimal separator may be specified. By default, the decimal separator is a period, however it can be another character as defined by DBMONEY or DBFORMAT.
-	Displays a minus sign for negative numbers.
+	Displays a plus sign for positive numbers.
$	This is the placeholder for the *front* specification of DBMONEY or DBFORMAT.
(	Displayed as left parentheses for negative numbers (accounting parentheses).
)	Displayed as right parentheses for negative numbers (accounting parentheses).

Formatting symbols for dates:

Character	Description
dd	Day of the month as a 2-digit integer.
ddd	Three-letter English-language abbreviation of the day of the week, for example, Mon, Tue.
mm	Month as a 2-digit integer.
mmm	Three-letter English-language abbreviation of the month, for example, Jan, Feb.
yy	Year, as a 2-digits integer representing the 2 trailing digits.
yyyy	Year as a 4-digit number.

Example:

```
01 MAIN
02   DEFINE d DECIMAL(12,2)
03   LET d = -12345678.91
04   DISPLAY d USING "$-##,###,##&.&&"
05   DISPLAY TODAY USING "yyyy-mm-dd"
06 END MAIN
```

CLIPPED

Purpose:

The `CLIPPED` operator removes trailing blanks of a string expression.

Syntax:

```
charexpr CLIPPED
```

Example:

```
01 MAIN
02   DISPLAY "Some text   " CLIPPED
03 END MAIN
```

SPACES

Purpose:

The `SPACES` operator returns a character string with blanks.

Syntax:

```
intexpr SPACES
```

Warnings:

1. *intexpr* is an integer expression.
2. "SPACE" is an alias for this operator.

Example:

```
01 MAIN
02   DISPLAY 20 SPACES || "xxx"
03 END MAIN
```

LSTR

Purpose:

The `LSTR` operator returns a Localized String corresponding to the identifier passed as parameter.

Syntax:

```
LSTR(strexpr)
```

Warnings:

1. *strexpr* is a string expression.

Example:

```
01 MAIN
02   DISPLAY LSTR ("str"||123)  -- loads string 'str123'
03 END MAIN
```

SFMT

Purpose:

The `SFMT` operator returns a string after replacing the parameter.

Syntax:

```
SFMT( strexpr [ , param [...] ] )
```

Warnings:

1. *strexpr* is a string expression.
2. *param* is any valid expression used to replace parameter place holders (%*n*).

Usage:

The `SFMT()` operator can be used with parameters that will be automatically set in the string at the position defined by parameter place holders. The parameters used with the `SFMT()` operator can be any valid expressions. Numeric and date/time expressions are evaluated to strings according to the current format settings (DBDATE, DBMONEY).

A place holder a is special marker in the string, that is defined by the percent character followed by the parameter number. For example, %4 represents the parameter #4. You are allowed to use the same parameter place holder several times in the string. If you want to use the percent sign in the string, you must escape it with %%.

Example:

```
01 MAIN
02   DEFINE n INTEGER
03   LET n = 234
04   DISPLAY SFMT("Order #%1 has been %2.",n,"deleted")
05 END MAIN
```

ADDITION

Purpose:

The + operator adds a number to another.

Syntax:

```
numexpr + numexpr
```

Example:

```
01 MAIN
02  DISPLAY 100 + 200
03 END MAIN
```

SUBTRACTION

Purpose:

The - operator subtracts a number from another.

Syntax:

```
numexpr - numexpr
```

Example:

```
01 MAIN
02  DISPLAY 100 - 200
03 END MAIN
```

MULTIPLICATION

Purpose:

The * operator multiplies a number with another.

Syntax:

```
numexpr * numexpr
```

Example:

```
01 MAIN
02  DISPLAY 100 * 200
03 END MAIN
```

DIVISION

Purpose:

The / operator divides a number by another.

Syntax:

```
numexpr / numexpr
```

Example:

```
01 MAIN
02  DISPLAY 100 / 200
03 END MAIN
```

EXPONENTIATION

Purpose:

The ** operator returns a value calculated by raising the left-hand operand to a power corresponding to the integer part of the right-hand operand.

Syntax:

```
numexpr ** intexpr
```

Example:

```
01 MAIN
02  DISPLAY 2 ** 8
03 END MAIN
```

MODULUS

Purpose:

The MOD operator returns the remainder, as an integer, from the division of the integer part of two numbers.

Syntax:

```
intexpr MOD intexpr
```

Example:

```
01 MAIN
02   DISPLAY 256 MOD 16
03   DISPLAY 26.51 MOD 2.7
04 END MAIN
```

CURRENT

Purpose:

The CURRENT operator returns the current date and time according to the qualifier.

Syntax:

```
CURRENT qual1 TO qual2[(scale)]
```

Notes:

1. *qual1, qual2* and *scale* define the date time qualifier; see the DATETIME data type for more details.

Example:

```
01 MAIN
02   DISPLAY CURRENT YEAR TO FRACTION(4)
03   DISPLAY CURRENT HOUR TO SECOND
04 END MAIN
```

EXTEND

Purpose:

The EXTEND operator adjusts a date time value according to the qualifier.

Syntax:

```
EXTEND ( dtexpr, qual1 TO qual2[(scale)] )
```

Notes:

1. This operator is used to convert a date time expression to a DATETIME value with a different precision.
2. *dtexpr* is a date or datetime expression. If it is a character string, it must consist of valid and unambiguous time-unit values and separators, but with these restrictions:
   - It cannot be a character string in date format, such as "12/12/99".
   - It cannot be an ambiguous numeric datetime value, such as "05:06" or "05".
   - It cannot be a time expression that returns an INTERVAL value.
3. *qual1, qual2* and *scale* define the date time qualifier, see the DATETIME data type for more details.
4. The default qualifier is YEAR TO DAY.

Example:

```
01 MAIN
02   DISPLAY EXTEND ( TODAY, YEAR TO FRACTION(4) )
03 END MAIN
```

DATE

Purpose:

The DATE operator converts a character expression, an integer or a datetime to a date value.

Syntax:

```
DATE [(dtexpr)]
```

Notes:

1. *dtexpr* is a character string, an integer or a datetime expression.

2. This operator is used to convert a character string, an integer or a date time value to a DATE value.
3. When *dtexpr* is a character string expression, it must properly formatted according to datetime format settings like DBDATE.
4. If *dtexpr* is an integer expression, it is used as the number of days since December 31, 1899.
5. If you supply no operand, it returns a character representation of the current date in the format "weekday month day year".

Example:

```
01 MAIN
02   DISPLAY DATE ( 34000 )
03   DISPLAY DATE ( "12/04/1978" )
04   DISPLAY DATE ( CURRENT )
05 END MAIN
```

TIME

Purpose:

The `TIME` operator converts the time-of-day portion of its datetime operand to a character string.

Syntax:

```
TIME [(dtexpr)]
```

Notes:

1. *dtexpr* is a datetime expression.
2. This operator converts a date time expression to a character string representing the time-of-day part of its operand.
3. The format of the returned string is always "hh:mm:ss".
4. If you supply no operand, it returns a character representation of the current time.

Example:

```
01 MAIN
02   DISPLAY TIME ( CURRENT )
03 END MAIN
```

TODAY

Purpose:

The TODAY operator returns the current calendar date.

Syntax:

```
TODAY
```

Notes:

1. Reads current system clock and returns a DATE value that represents the current calendar date.

Tips:

1. See also the CURRENT operator that returns current date and time.

Example:

```
01 MAIN
02   DISPLAY TODAY
03 END MAIN
```

YEAR

Purpose:

The YEAR operator extracts the year of a date time expression.

Syntax:

```
YEAR ( dtexpr )
```

Notes:

1. *dtexpr* is a date or datetime expression.
2. Returns an integer corresponding to the year portion of its operand.

Example:

```
01 MAIN
02   DISPLAY YEAR ( TODAY )
03   DISPLAY YEAR ( CURRENT )
04 END MAIN
```

MONTH

Purpose:

The `MONTH` operator extracts the month of a date time expression.

Syntax:

```
MONTH ( dtexpr )
```

Notes:

1. *dtexpr* is a date or datetime expression.
2. Returns a positive whole number between 1 and 12 corresponding to the month of its operand.

Example:

```
01 MAIN
02   DISPLAY MONTH ( TODAY )
03   DISPLAY MONTH ( CURRENT )
04 END MAIN
```

DAY

Purpose:

The `DAY` operator extracts the day of the month of a date time expression.

Syntax:

```
DAY ( dtexpr )
```

Notes:

1. *dtexpr* is a date or datetime expression.
2. Returns a positive whole number between 1 and 31 corresponding to the day of the month of its operand.

Example:

```
01 MAIN
02   DISPLAY DAY ( TODAY )
03   DISPLAY DAY ( CURRENT )
04 END MAIN
```

WEEKDAY

Purpose:

The `WEEKDAY` operator extracts the day of the week of a date time expression.

Syntax:

```
WEEKDAY ( dtexpr )
```

Notes:

1. *dtexpr* is a date or datetime expression.
2. Returns a positive whole number between 0 and 6 corresponding to the day of the week implied by its operand.
3. The integer 0 (Zero) represents Sunday.

Example:

```
01 MAIN
02   DISPLAY WEEKDAY ( TODAY )
03   DISPLAY WEEKDAY ( CURRENT )
04 END MAIN
```

MDY

Purpose:

The `MDY` operator builds a date value with 3 integers representing the month, day and year.

Syntax:

```
MDY ( intexpr1, intexpr2, intexpr3 )
```

Notes:

1. *intexpr1* is an integer representing the month (from 1 to 12).
2. *intexpr2* is an integer representing the day (from 1 to 28, 29, 30 or 31 depending on the month).
3. *intexpr3* is an integer representing the year (four digits).
4. The result is a DATE value.

Example:

```
01 MAIN
02   DISPLAY MDY ( 12, 3+2, 1998 )
03 END MAIN
```

UNITS

Purpose:

The UNITS operator converts an integer expression to an interval value.

Syntax:

```
intexpr UNITS qual[(scale)]
```

Notes:

1. *intexpr* is an integer expression.
2. *qual* is one of the unit specifiers of a DATETIME qualifier.
3. The result is a INTERVAL value.

Example:

```
01 MAIN
02   DEFINE d DATE
03   LET d = TODAY + 200
04   DISPLAY (d - TODAY) UNITS DAY
05 END MAIN
```

GET_FLDBUF

Purpose:

The GET_FLDBUF operator returns as character strings the current values of the specified fields.

Syntax:

```
GET_FLDBUF ( [group.]field [,...] )
```

Notes:

1. *group* can be a table name, a screen record, a screen array or 'formonly'.
2. *field* is the name of the screen field.

3. Typically used to get the value of a screen field before the input buffer is copied into the associated variable.
4. If multiple fields are specified between parentheses, you must use the RETURNING clause.
5. When used in a INPUT ARRAY instruction, the runtime system assumes that you are referring to the current row.

Warnings:

1. The values returned by this operator are context dependent; it must be used carefully. If possible, use the variable associated to the input field instead.

Example:

```
01 ...
02   LET v = GET_FLDBUF( customer.custname )
03   CALL GET_FLDBUF( customer.* ) RETURNING rec_customer.*
04 ...
```

INFIELD

Purpose:

The `INFIELD` operator returns TRUE if its operand is the identifier of the current screen field.

Syntax:

```
INFIELD ( [group.]field )
```

Notes:

1. *group* can be a table name, a screen record, a screen array or 'FORMONLY'.
2. *field* is the name of the field in the form.
3. Typically used to check for the current field in a CONSTRUCT, INPUT or INPUT ARRAY instruction.
4. When used in an INPUT ARRAY instruction, the runtime system assumes that you are referring to the current row.

Example:

```
01 ...
02 INPUT ...
03   IF INFIELD( customer.custname ) THEN
04      MESSAGE "The current field is customer's name."
05 ...
```

FIELD_TOUCHED

Purpose:

The FIELD_TOUCHED operator returns TRUE if the value of a screen field has changed since the beginning of the interactive instruction.

Syntax:

```
FIELD_TOUCHED ( [group.]field [,...] )
```

Notes:

1. *group* can be a table name, a screen record, a screen array or 'FORMONLY'.
2. *field* is the name of the field in the form.
3. Typically used to check if the value of a field was edited..
4. When used in an INPUT ARRAY instruction, the runtime system assumes that you are referring to the current row.

Warnings:

1. After a DISPLAY instruction, the modified field is marked as 'touched'.
2. Do not confuse with FGL_BUFFERTOUCHED; in that function, the flag is reset when entering a new field.

Usage:

For more details about the FIELD_TOUCHED operator usage and the understand the "touched flag" concept, see the Touched Flag section of the DIALOG instruction.

Example:

```
01 ...
02 AFTER INPUT
03   IF FIELD_TOUCHED( customer.custname ) THEN
04      MESSAGE "Customer name was changed."
05 ...
```

Expressions

Summary:

- Definition
- Boolean Expressions
- Integer Expressions
- Number Expressions
- String Expressions
- Date Expressions
- Datetime Expressions
- Interval Expressions

See also: Variables, Data Types, Literals, Constants.

Definition

What is an Expression?

An Expression is a sequence of operands, operators, and parentheses that the runtime system can evaluate as a single value.

Expressions can include the following components:

- Operators, as described in the Operators section.
- Parentheses, to overwrite precedence of operators.
- Operands, including the following:
  - Variables
  - Constants
  - Functions (returning a single value)
  - Literal values
  - Other expressions

Differences Between BDL and SQL Expressions

Expressions in SQL statements are evaluated by the database server, not by the runtime system. The set of operators that can appear in SQL expressions resembles the set of BDL operators, but they are not identical. A program can include SQL operators, but these are restricted to SQL statements. Similarly, most SQL operands are not valid in BDL expressions. The SQL identifiers of databases, tables, or columns can appear in a `LIKE` clause or field name in BDL statements, provided that these SQL identifiers comply with the naming rules of BDL. Here are some examples of SQL operands and operators that cannot appear in other BDL expressions:

- SQL identifiers, such as column names
- The SQL keywords `USER` and `ROWID`

- Built-in or aggregate SQL functions that are not part of BDL
- The `BETWEEN` and `IN` operators
- The `EXISTS`, `ALL`, `ANY`, or `SOME` keywords of SQL expressions

Conversely, you *cannot* include BDL specific operators in SQL expressions, as for example:

- Arithmetic operators for exponentiation (`**`) and modulus (`MOD`)
- String operators `ASCII`, `COLUMN`, `SPACE`, `SPACES`, and `WORDWRAP`
- Field operators `FIELD_TOUCHED( )`, `GET_FLDBUF( )`, and `INFIELD( )`
- The report operators `LINENO` and `PAGENO`

Parentheses in BDL Expressions

You can use parentheses as you would in algebra to override the default order of precedence of operators. In mathematics, this use of parentheses represents the "associative" operator. It is, however, a convention in computer languages to regard this use of parentheses as delimiters rather than as operators. (Do not confuse this use of parentheses to specify *operator precedence* with the use of parentheses to enclose arguments in function calls or to delimit other lists.)

In the following example, the variable **y** is assigned the value of 2:

```
LET y = 15 MOD 3 + 2
```

In the next example, however, **y** is assigned the value of 0 because the parentheses change the sequence of operations:

```
LET y = 15 MOD (3 + 2)
```

Boolean Expressions

A **Boolean expression** is one that evaluates to an INTEGER value that can be TRUE, FALSE and in some cases, NULL.

Notes:

1. Boolean expressions are a combination of Logical Operators and Boolean comparisons based on Comparison Operators.
2. Boolean expressions are based on the INTEGER data type for evaluation.
3. Any integer value different from zero is defined as true, while zero is defined as false.
4. Use an `INTEGER` variable to store the result of a Boolean expression.
5. If an expression that returns `NULL` is the operand of the `IS NULL` operator, the value of the Boolean expression is TRUE.

6. If you include a Boolean expression in a context where the runtime system expects a number, the expression is evaluated, and is then converted to an integer by the rules: `TRUE = 1` and `FALSE = 0`.
7. The Boolean expression evaluates to `TRUE` if the value is a non-zero real number *or* any of the following items:
   - Character string representing a non-zero number
   - Non-zero `INTERVAL`
   - Any `DATE` or `DATETIME` value
   - A `TRUE` value returned by a Boolean function like `INFIELD( )`
   - The built-in integer constant `TRUE`
8. If a Boolean expression includes an operand whose value is not an integer data type, the runtime system attempts to convert the value to an integer according to the data conversion rules.

Example:

```
01 MAIN
02   DEFINE r, c INTEGER
03   LET c = 4
03   LET r = TRUE!=FALSE AND ( c=2 OR c=4 )
04   IF ( r AND canReadFile("config.txt") ) THEN
05      DISPLAY "OK"
06   END IF
07 END MAIN
```

Warnings:

1. A Boolean expression evaluates to NULL if the value is `NULL` *and* the expression does not appear in any of the following contexts:
   - The `IS [NOT] NULL` test.
   - Boolean Comparisons.
   - Any conditional statement (`IF`, `CASE`, `WHILE`).
2. The syntax of Boolean expressions in BDL is not the same as *Boolean conditions* in SQL statements.
3. Boolean expressions in `CASE`, `IF`, or `WHILE` statements return `FALSE` if any element of the comparison is `NULL`, except for operands of the `IS NULL` and the `IS NOT NULL` operator. See Boolean Operators for more information about individual Boolean operators and Boolean expressions.

Integer Expressions

An **Integer expression** is one that evaluates to a whole number.

Notes:

1. The data type of the expression result can be SMALLINT or INTEGER.
2. The operands must be one of:
   - An integer literal

    - A variable or constant of type SMALLINT or INTEGER
    - A function returning a single integer value
    - A Boolean expression
    - The result of a DATE subtraction
3. If an integer expression includes an operand whose value is not an integer data type, the runtime system attempts to convert the value to an integer according to the data conversion rules.

Example:

```
01 MAIN
02   DEFINE r, c INTEGER
03   LET c = 4
04   LET r = c * ( 2 + c MOD 4 ) / getRowCount("customers")
05 END MAIN
```

Warnings:

1. If an element of an integer expression is NULL, the expression is evaluated to NULL.

Number Expressions

A **Number expression** is one that evaluates to a number data type.

Notes:

1. The data type of the expression result can be SMALLINT, INTEGER, DECIMAL, SMALLFLOAT or FLOAT.
2. The operands must be one of:
    - An integer literal
    - A decimal literal
    - A variable or constant of numeric data type
    - A function returning a single numeric value
    - A Boolean expression
    - The result of a DATE subtraction
3. If a number expression includes an operand whose value is not a numeric data type, the runtime system attempts to convert the value to a number according to the data conversion rules.

Example:

```
01 MAIN
02   DEFINE r, c DECIMAL(10,2)
03   LET c = 456.22
04   LET r = c * 2 + ( c / 4.55 )
05 END MAIN
```

Warnings:

1. If an element of a number expression is NULL, the expression is evaluated to `NULL`.

String Expressions

A **String expression** is one that includes at least one character string value and that evaluates to the STRING data type.

Notes:

1. The data type of the expression result is STRING.
2. At least one of the operands must be one of:
   - A string literal.
   - A variable or constant of type CHAR, VARCHAR or STRING.
   - A function returning a single character value
3. Other operands whose values are not character string data types are converted to strings according to the data conversion rules.

Example:

```
01 MAIN
02   DEFINE r, c VARCHAR(100)
03   LET c = "abcdef"
04   LET r = c[1,3] || ": " || TODAY USING "YYYY-MM-DD" || " " ||
length(c)
05 END MAIN
```

Warnings:

1. If an element of an integer expression is NULL, the expression is evaluated to NULL.
2. An empty string ("") is equivalent to NULL.

Date Expressions

A **Date expression** is one that evaluates to a DATE data type.

Notes:

1. The data type of the expression result is a DATE value.
2. The operands must be one of:
   - A string literal that can be evaluated to a Date according to DBDATE
   - A variable or constant of type DATE

    - A function returning a single Date value
    - A unary `+` or `-` associated to an Integer expression representing a number of days
    - The `TODAY` constant
    - A `CURRENT` expression with `YEAR TO DAY` qualifiers
    - An `EXTEND` expression with `YEAR TO DAY` qualifiers
3. If a date expression includes an operand whose value is not a date data type, the runtime system attempts to convert the value to a date value according to the data conversion rules.

Example:

```
01 MAIN
02   DEFINE r, c DATE
03   LET c = TODAY + 4
04   LET r = ( c - 2 )
05 END MAIN
```

Warnings:

1. If an element of an integer expression is NULL, the expression is evaluated to NULL.

Datetime Expressions

A **Datetime expression** is one that evaluates to a DATETIME data type.

Notes:

1. The data type of the expression result is a DATETIME value.
2. The operands must be one of:
    - A datetime literal
    - A string literal representing a Datetime with the ISO format `YYYY-MM-DD hh:mm:ss.fffff`
    - A variable or constant of DATETIME type
    - A function returning a single Datetime value
    - A unary `+` or `-` associated to an Interval expression
    - A `CURRENT` expression
    - An `EXTEND` expression
3. If a datetime expression includes an operand whose value is not a datetime data type, the runtime system attempts to convert the value to a datetime value according to the data conversion rules.

Example:

```
01 MAIN
02   DEFINE r, c DATETIME YEAR TO SECOND
03   LET c = CURRENT YEAR TO SECOND
```

```
04   LET r = c + INTERVAL( 234-02 ) YEAR TO MONTH
05 END MAIN
```

Warnings:

1. If an element of an integer expression is NULL, the expression is evaluated to NULL.

Interval Expressions

An **Interval expression** is one that evaluates to a INTERVAL data type.

Notes:

1. The data type of the expression result is a INTERVAL value.
2. The operands must be one of:
   - An interval literal
   - A string literal representing an Interval with the ISO format `YYYY-MM-DD hh:mm:ss.fffff`
   - An integer expression using the UNITS operator
   - A variable or constant of INTERVAL type
   - A function returning a single Interval value
   - The result of a DATETIME subtraction
3. If an interval expression includes an operand whose value is not an interval data type, the runtime system attempts to convert the value to an interval value according to the data conversion rules.

Example:

```
01 MAIN
02   DEFINE r, c INTERVAL HOUR TO MINUTE
03   LET c = "12:45"
04   LET r = c + ( DATETIME( 14-02 ) HOUR TO MINUTE - DATETIME( 10-43 )
HOUR TO MINUTE )
05 END MAIN
```

Warnings:

1. If an element of an integer expression is NULL, the expression is evaluated to NULL.

Exceptions

Summary:

- Exceptions handling
- Exception Actions
- Exception Types
- Exception Classes
- Exceptions handler (`WHENEVER`)
- Exception blocks (`TRY/CATCH`)
- Handling SQL Errors
- Tracing exceptions
- Examples
  - Example 1 - `WHENEVER ERROR CALL`
  - Example 2 - `WHENEVER ERROR CONTINUE / STOP`
  - Example 3 - `TRY / CATCH`
  - Example 4 - `WHENEVER + TRY CATCH`
  - Example 5 - `WHENEVER ERROR RAISE`

See also: Flow Control, Fgl Errors.

Exception handling

If an instructions executes abnormally, the runtime system throws exceptions that can be handled by the program. Actions can be taken based on the class of the exception. There is no way to raise exceptions explicitly; only the runtime system can throw exceptions. Runtime errors (i.e. exceptions) can be trapped by a WHENEVER exception handler or by a TRY / CATCH block.

Exception Actions

There are five actions that can be executed if an exception is raised:

`STOP`

> The program is immediately terminated. A message is displayed to the standard error with the location of the related statement, the error number, and the details of the exception.

`CONTINUE`

> The program continues normally (the exception is ignored).

`CALL` *name*

The function *name* is called by the runtime system. The function can be defined in any module, and must have zero parameters and zero return values. The STATUS variable will be set to the corresponding error number.

`GOTO` *name*

The program execution continues at the label identified by *name*.

`RAISE`

This statement instructs the DVM that an exception raised will not be handled by the local function, but by the calling function. If an exception is raised, the current function will return and the exception handling is left to the caller function.

Exception Types

There are four types of exceptions, defining the kind of errors that can occur:

Type	Reason	Examples
`ET_STATEMENT`	Error occurred in a statement.	DISPLAY AT invalid coordinates.
`ET_EXPRESSION`	Expression evaluation error.	Division by zero.
`ET_NOTFOUND`	An SQL statement returns status NOTFOUND.	FETCH when cursor is on last row.
`ET_WARNING`	An SQL statement sets sqlca.sqlawarn flag.	Fetched CHAR value has been truncated.

Exception Classes

The exception classes indirectly define the exception type:

Class	Related Exception Type (defines the error reason)	Default Action
`ERROR`	`ET_STATEMENT`	STOP
`ANY ERROR`	`ET_STATEMENT` and `ET_EXPRESSION`	CONTINUE
`NOT FOUND`	`ET_NOTFOUND`	CONTINUE
`WARNING`	`ET_WARNING`	CONTINUE

WHENEVER

The WHENEVER instruction defines exception handling in a program module, by associating an exception class with an exception action.

Syntax:

```
WHENEVER [ANY] ERROR { CONTINUE | STOP | CALL function | RAISE | GOTO label }
```

Notes:

1. *function* can be any function name defined in the program.
2. *label* can be any label defined in the current module.
3. Exception classes ERROR and SQLERROR are synonyms (compatibility issue).
4. Actions for classes ERROR, WARNING and NOT FOUND can be set independently.

Tips:

1. For SQL instructions that can potentially generate errors, it is recommended that you define an exception handler locally; errors in the rest of the program can be handled by the default exception handler. See example 2 for more details.

Warning: The scope of a WHENEVER instruction is similar to a C pre-processor macro. It is local to the module and valid until the end of the module, unless a new WHENEVER instruction is encountered by the compiler.

TRY - CATCH pseudo statement

Any FGL statement in the TRY block will be executed until an exception is thrown. After an exception the program execution continues in the CATCH block. If no CATCH block is provided, the execution continues after END TRY.

Without an exception the program continues after END TRY.

TRY can be compared with WHENEVER ERROR GOTO label.

The next two code fragments have similar behavior:

```
01 -- Exception handling using TRY CATCH
02 TRY
03     -- fgl-statements
04 CATCH
05     -- fgl-statements catching the error
06 END TRY
```

```
01 -- traditional fgl using WHENEVER ERROR GOTO
02 WHENEVER ERROR GOTO catch_error
03 -- fgl-statements
04 GOTO no_error
05 LABEL catch_error:
06 WHENEVER ERROR STOP
07 -- fgl-statements catching the error
08 LABEL no_error
```

The TRY statement can be nested in other TRY statements.

The TRY statement is a pseudo statement, because it does not instruct the compiler to generate code. It is not possible to set a debugger break point at TRY, CATCH or END TRY.

Handing SQL Errors

After executing an SQL statement, you can query STATUS, SQLSTATE, SQLERRMESSAGE and the SQLCA record to get the description of the error. When the statement has been executed with errors, STATUS and SQLCA.SQLCODE contain the **SQL Error Code**. If no error occurs, STATUS and SQLCA.SQLCODE are set to zero.

You control the result of an SQL statement execution by using the WHENEVER ERROR exception handler:

```
01 MAIN
02
03   DATABASE stores
04
05   WHENEVER ERROR CONTINUE
06   SELECT COUNT(*) FROM customer
07   IF sqlca.sqlcode THEN
08      ERROR "SQL Error occurred:", sqlca.sqlcode
09   END IF
10   WHENEVER ERROR STOP
11
12 END MAIN
```

The **SQL Error Codes** are not standard. For example, ORACLE returns 903 when a table name does not exist.

By convention, the STATUS and SQLCA.SQLCODE variables always use IBM Informix SQL Error Codes. When using IBM Informix, both STATUS and SQLCA.SQLCODE variables contain the native Informix error code. When using other database servers, the database interface automatically converts native SQL Error Codes to IBM Informix Error Codes. If no equivalent Informix Error Code can be found, the interface returns **-6372** in SQLCA.SQLCODE.

If an SQL error occurs when using IBM Informix, the SQLCA variable is filled with standard information as described in the Informix documentation. When using other

database servers, the native SQL Error Code is available in the SQLCA.SQLERRD[2] register. SQL Error Codes in SQLCA.SQLERRD[2] are always negative, even if the database server defines positives SQL Error Codes. Additionally, if the target database API supports ANSI SQL states, the SQLSTATE code is returned in SQLCA.SQLERRM.

The NOTFOUND (100) execution status is returned after a FETCH, when no rows are found.

See also: Connections.

Tracing exceptions

Exceptions will be automatically logged in a file by the runtime system if all the following conditions are true:

- The STARTLOG function has been previously called to specify the name of the exception logging file.
- The exception action is set to `CALL`, `GOTO` or `STOP`. Exceptions are not logged when the action is `CONTINUE`.
- The exception class is an `ERROR`, `ANY ERROR` or `WARNING`. `NOT FOUND` exceptions cannot be logged.

Each log entry contains:

- The system-time
- The location of the related FGL statement (source-file, line)
- The error-number
- The text of the error message, giving human-readable details for the exception

---

Examples

Example 1:

```
01 MAIN
02   WHENEVER ERROR CALL error_handler
03   DATABASE stores
04   SELECT dummy FROM systables WHERE tabid=1
05 END MAIN
06
07 FUNCTION error_handler()
08   DISPLAY "Error:", STATUS
09   EXIT PROGRAM 1
10 END FUNCTION
```

Example 2:

```
01 MAIN
02   DEFINE tabname VARCHAR(50)
03   DEFINE rowcount INTEGER
04
05   # In the DATABASE statement, no error should occur.
06   DATABASE stores
07
08   # But in the next procedure, user may enter a wrong table.
09   WHENEVER ERROR CONTINUE
10   PROMPT "Enter a table name:" FOR tabname
11   LET sqlstmt = "SELECT COUNT(*) FROM " || tabname
12   PREPARE s FROM sqlstmt
13   IF sqlca.sqlcode THEN
14      ERROR "SQL Error occurred:", sqlca.sqlcode
15   END IF
16   EXECUTE s INTO rowcount
17   IF sqlca.sqlcode THEN
18      ERROR "SQL Error occurred:", sqlca.sqlcode
19   END IF
20   WHENEVER ERROR STOP
21
22 END MAIN
```

Example 3:

```
01 MAIN
02     TRY
03         DATABASE invalid_database_name
04         DISPLAY "Will not be displayed"
05     CATCH
06         DISPLAY "Exception caught, status = ", status USING "----&"
07     END TRY
08 END MAIN
```

Example 4:

```
01 MAIN
02     DEFINE i INTEGER
03     WHENEVER ANY ERROR CALL foo
04     TRY
05         DISPLAY "Next exception should be handled by the CATCH
statement"
06         LET i = i / 0
07     CATCH
08         DISPLAY "Exception caught, status = ", status USING "----&"
09     END TRY
10     -- The previous error handler is restored after the TRY - CATCH
block
11     LET status = 0
12     DISPLAY "Next exception should be handled by the foo function"
13     LET i = i / 0
14 END MAIN
```

```
15
16 FUNCTION foo()
17     DISPLAY "foo called, status = ", status USING "----&"
18 END FUNCTION
```

Example 5:

```
01 MAIN
02     DEFINE i INTEGER
03     WHENEVER ANY ERROR CALL exception_handler
04     DISPLAY "Next function call will generate an exception"
05     DISPLAY "This exception should be handled by the function
exception_handler"
06     DISPLAY do_exception(100, 0)
07     WHENEVER ANY ERROR STOP
08 END MAIN
09
10 FUNCTION do_exception(a, b)
11     DEFINE a, b INTEGER
12     WHENEVER ANY ERROR RAISE
13     RETURN a / b
14 END FUNCTION
15
16 FUNCTION exception_handler()
17     DISPLAY "Exception caught, status = ", status USING "----&"
18 END FUNCTION
```

Variables

Summary:

- Definition
- Defining Variables (`DEFINE`)
- Declaration Context
- Structured Types
- Database Types
- User Types
- Default Values
- Variable Initialization (`INITIALIZE`)
- LOB Data Localization (`LOCATE`)
- LOB Data Release (`FREE`)
- Assigning Values (`LET`)
- Data Validation (`VALIDATE`)
- Examples

See also: Records, Arrays, Data Types, Constants, User Types.

Definition

A *variable* is a program element that can hold volatile data. The following list summarizes variables usage:

- You must DEFINE variables before use.
- After definition, variables have default values according to the data type.
- The scope of a variable can be global, local to a module, or local to a function.
- You can define structured variables with records and arrays.
- Variables can be initialized with the INITIALIZE instruction.
- Variables can be assigned with the LET instruction.
- Variables can be validated with the VALIDATE instruction.
- Variables can be used as parameters or fetch buffers in Static or Dynamic SQL statements.
- Variables can be used as input or display buffers in interactive instructions such as INPUT, INPUT ARRAY, DISPLAY ARRAY, CONSTRUCT.

DEFINE

Purpose:

A **variable** contains volatile information of a specific data type.

Syntax:

```
DEFINE identifier [,...] { type | LIKE [dbname:]tabname.colname }
```

Notes:

1. *identifier* is the name of the variable to be defined. See Identifiers for naming rules.
2. *type* can be any data type supported by the language, a record definition, an array definition, a user type, or a built-in class.
3. When using the `LIKE` clause, the data type is taken from the schema file. Columns defined as `SERIAL` are converted to INTEGER.
4. *dbname* identifies a specific database schema file.
5. *tabname.colname* can be any column reference defined in the database schema file.

Usage:

A *variable* is a named location in memory that can store a single value, or an ordered set of values. Variables can be global to the program, local to a module, or local to a function.

You cannot reference any program variable before it has been declared by the `DEFINE` statement.

Tips:

1. To write well-structured programs, it is recommended that you not use global variables. If you need persistent data storage during a program's execution, use variables local to the module and give access to them with functions.

Warnings:

1. When defining variables with the `LIKE` clause, the data types are taken from the schema file **during compilation**. Make sure that the schema file of the development database corresponds to the production database; otherwise the variables defined in the compiled version of your modules will not match the table structures of the production database.

Declaration Context

The `DEFINE` statement declares the identifier of one or more *variables*. There are two important things to know about these identifiers:

- Where in the program can they be used? The answer defines the scope of reference of the variable. A point in the program where an identifier can be used

is said to be *in* the scope of the identifier. A point where the identifier is not known is *outside* the scope of the identifier.
- When is storage for the variable allocated? Storage can be allocated either statically, when the program is loaded to run (at load time), or dynamically, while the program is executing (at runtime).

The context of its declaration in the source module determines where a variable can be referenced by other language statements, and when storage is allocated for the variable in memory. The DEFINE statement can appear in three contexts:

1. Within a `FUNCTION`, `MAIN`, or `REPORT` program block, `DEFINE` declares *local* variables, and causes memory to be allocated for them. These DEFINE declarations of local variables must precede any executable statements within the same program block.
    - The scope of reference of a local variable is restricted to the same program block. The variable is not visible elsewhere.
    - Storage for local variables is allocated when its `FUNCTION`, `REPORT`, or `MAIN` block is entered during execution. Functions can be called recursively, and each recursive entry creates its own set of local variables. The variable is unique to that invocation of its program block. Each time the block is entered, a new copy of the variable is created.
2. Outside any `FUNCTION`, `REPORT`, or `MAIN` program block, the `DEFINE` statement declares names and data types of *module* variables, and causes storage to be allocated for them. These declarations must appear before any program blocks.
    - Scope of reference is from the `DEFINE` statement to the end of the same module. The variable, however, is not visible within this scope in program blocks where a local variable has the same identifier.
    - Memory for variables of modules is allocated statically, when the program starts.
3. Inside a `GLOBALS` block, the `DEFINE` statement declares *global* variables.
    - Scope of reference is global to the whole program.
    - The memory for global variables is allocated statically, when the program starts.
    - Multiple `GLOBALS` blocks can be defined for a given module. Use one module to declare all global variables and reference that module within other modules by using the `GLOBALS "filename.4gl"` statement as the first statement in the module, outside any program block.

A compile-time error occurs if you declare the same name for two variables that have the same scope. You can, however, declare the same name for variables that differ in their scope. For example, you can use the same identifier to reference different local variables in different program blocks.

You can also declare the same name for two or more variables whose scopes of reference are different but overlapping. Within their intersection, the compiler interprets the identifier as referencing the variable whose scope is smaller, and therefore the variable whose scope is a superset of the other is not visible.

If a local variable has the same identifier as a global variable, then the local variable takes precedence inside the program block in which it is declared. Elsewhere in the program, the identifier references the global variable.

A module variable can have the same name as a global variable that is declared in a different module. Within the module where the module variable is declared, the module variable takes precedence over the global variable. Statements in that module cannot reference the global variable.

A module variable cannot have the same name as a global variable that is declared in the same module.

If a local variable has the same identifier as a module variable, then the local identifier takes precedence inside the program block in which it is declared. Elsewhere in the same source-code module, the name references the module variable.

Structured Types

You can use the `RECORD` or `ARRAY` keywords to declare a structured variable.

For example:

```
01 MAIN
02   DEFINE myarr ARRAY[100] OF RECORD
03      id INTEGER,
04      name VARCHAR(100)
05   END RECORD
06   LET myarr[2].id = 52
07 END MAIN
```

For more detail, refer to Arrays and Records.

Database Types

You can use the `LIKE` keyword to declare a variable that has the same data type as a specified column in a database schema.

For example:

```
01 SCHEMA stores
02 DEFINE cname LIKE customer.cust_name
03 MAIN
05   DEFINE cr RECORD LIKE customer.*
06 END MAIN
```

The following rules apply when using the `LIKE` keyword:

- A `SCHEMA` statement must define the database name identifying the database schema files to be used.
- The column data types are read from the schema file **during compilation**, not at runtime. Make sure that your schema files correspond exactly to the production database.
- The database schema files must exist and must be available as specified in the FGLDBPATH variable.
- The column data type defined by the database schema must be supported by the language. For more detail about supported types, refer to Data Types.
- When using database views, the column cannot be based on an aggregate function like `SUM()`.
- If `LIKE` references a `SERIAL` column, the new variable is of the INTEGER data type.
- The table qualifier must specify *owner* if *table.column* is not a unique column identifier within its database, or if the database is ANSI-compliant and any user of your application is not the owner of *tabl*e.

To understand how to generate database schema files with the schema extractor tool, refer to Database Schema Files

---

User Types

Variables can be defined with a user type:

```
01 TYPE custlist DYNAMIC ARRAY OF RECORD LIKE customer.*
02 MAIN
03   DEFINE cl custlist
04 END MAIN
```

The scope of a user type can be global, local to a module or local to a function. Variables can be defined with a user type defined in the same scope, or in a higher level of scope.

Default Values

After a variable is defined, it is automatically initialized by the runtime system to a default value based on the data type. The following table shows all possible default values that variables can take:

Data Type	Default Value
`CHAR`	NULL
`VARCHAR`	NULL
`STRING`	NULL
`INTEGER`	Zero

SMALLINT	Zero
FLOAT	Zero
SMALLFLOAT	Zero
DECIMAL	NULL
MONEY	NULL
DATE	1899-12-31 (= Zero in number of days)
DATETIME	NULL
INTERVAL	NULL
TEXT	NULL, See LOCATE
BYTE	NULL, See LOCATE

INITIALIZE

Purpose:

The INITIALIZE instruction assigns NULL or default values to variables.

Syntax:

```
INITIALIZE target [,...] { TO NULL | LIKE {table.*|table.column} }
```

Notes:

1. *target* is the name of the variable to be initialized.
2. *target* can be a simple variable, a record, a record member, an array or an array element.
3. If *target* is a record, you must use the star notation to reference all record members in the initialization.
4. *table.column* can be any column reference defined in the database schema files.

Usage:

The TO NULL clause initializes the variable to NULL.

When initializing a static or dynamic array TO NULL, all elements will be initialized to NULL. Note that dynamic arrays will keep the same number of elements (i.e. they are not cleared).

The LIKE clause initializes the variable to the default value defined in the database schema validation file. This clause works only by specifying the *table.column* schema entry corresponding to the variable.

To initialize a complete record, you can use the star to reference all members:

```
01 INITIALIZE record.* LIKE table.*
```

Warnings:

1. You cannot initialize variables defined with a complex data type (like TEXT or BYTE) to a non-NULL value.

Example:

```
01 DATABASE stores
02 MAIN
03   DEFINE cr RECORD LIKE customer.*
04   DEFINE a1 ARRAY[100] OF INTEGER
05   INITIALIZE cr.cust_name TO NULL
06   INITIALIZE cr.* LIKE customer.*
07   INITIALIZE a1 TO NULL
08   INITIALIZE a1[10] TO NULL
09 END MAIN
```

VALIDATE

Purpose:

The VALIDATE statement tests whether the value of a variable is within the range of values for a corresponding column in database schema files.

Syntax:

```
VALIDATE target [,...] LIKE {table.*|table.column}
```

Notes:

1. *target* is the name of the variable to be validated.
2. *target* can be a simple variable, a record, or an array element.
3. If target is a record, you can use the star to reference all record members in the validation.
4. Values are compared to the value defined in the database schema validation file.
5. *table.column* can be any column reference defined in the database schema file.

Errors:

1. If the value does not match any value defined in the INCLUDE attribute of the corresponding column, the runtime system raises an exception with error code **-1321**.

Warnings:

1. The `LIKE` clause requires the IBM Informix **upscol** utility to populate the **syscolval** table. See the database schema files for more details. **Informix only!**
2. You cannot initialize variables defined with a complex data type (like TEXT or BYTE) to a non-NULL value.

Example:

```
01 DATABASE stores
02 MAIN
03   DEFINE cname LIKE customer.cust_name
04   LET cname = "aaa"
05   VALIDATE cname LIKE customer.cust_name
06 END MAIN
```

LET

Purpose:

The `LET` statement assigns a value to a variable, or a set of values to a record.

Syntax:

```
LET target = expression
```

Notes:

1. *target* is the name of the variable to be assigned.
2. *target* can be a simple variable, a record, or an array element.
3. *expression* is any valid expression supported by the language
4. The runtime system applies data type conversion rules if the data type of *expression* does not correspond to the data type of *target*.
5. If *target* is a record, you can use the star to reference all record members in the validation, and expressions can also use this notation (`record.*`).

Warnings:

1. Variables defined with a complex data type (like TEXT or BYTE) can only be assigned to NULL.
2. For more detail, refer to the assignment operator.

Example:

```
01 DATABASE stores
02 MAIN
03   DEFINE c1, c2 RECORD LIKE customer.*
04   LET c1.* = c2.*
```

```
05 END MAIN
```

LOCATE

Purpose:

The `LOCATE` statement specifies where to store data of TEXT and BYTE variables.

Syntax:

```
LOCATE target [,...] IN { MEMORY | FILE filename }
```

Notes:

1. Defining the location of large object data is mandatory before usage.
2. *target* is the name of a TEXT or BYTE variable to be located.
3. *target* can be a simple variable, a record member, or an array element.
4. *filename* is a string expression defining the name of a file.
5. The `IN MEMORY` clause specifies that the large object data must be located in memory.
6. The `IN FILE` clause specifies that the large object data must be located in a file.
7. After defining the data storage, the variable can be used as a parameter or as a fetch buffer in SQL statements.
8. You can free the resources allocated to the large object variable with the FREE instruction.

Warnings:

1. You cannot use a large object variable if the data storage location is not defined.

Example:

```
01 MAIN
02   DEFINE ctext1, ctext2 TEXT
03   DATABASE stock
04   LOCATE ctext1 IN MEMORY
05   LOCATE ctext2 IN FILE "/tmp/data1.txt"
06   CREATE TABLE lobtab ( key INTEGER, col1 TEXT, col2 TEXT )
06   INSERT INTO lobtab VALUES ( 123, ctext1, ctext2 )
07 END MAIN
```

FREE

Purpose:

The FREE statement releases resources allocated to store the data of TEXT and BYTE variables.

Syntax:

```
FREE target
```

Notes:

1. *target* is the name of a TEXT or BYTE variable to be freed.
2. *target* can be a simple variable, a record member, or an array element.
3. If the variable was located in memory, the runtime system releases the memory.
4. If the variable was located in a file, the runtime system deletes the named file.
5. For variables declared in a local scope of reference, the resources are automatically freed by the runtime system when returning from the function or MAIN block.

Warnings:

1. After freeing a large object, you must LOCATE the variable again before usage.

Example:

```
01 MAIN
02   DEFINE ctext TEXT
03   DATABASE stock
03   LOCATE ctext IN FILE "/tmp/data1.txt"
04   SELECT col1 INTO ctext FROM lobtab WHERE key=123
05   FREE ctext
06 END MAIN
```

Examples

Example 1: Function variables

```
01 FUNCTION myfunc( )
02   DEFINE i INTEGER
03   FOR i=1 TO 10
04       DISPLAY i
05   END FOR
06 END FUNCTION
```

Example 2: Module variables

```
01 DEFINE s VARCHAR(100)
02
03 FUNCTION myfunc( )
04   DEFINE i INTEGER
05   FOR i=1 TO 10
06       LET s = "item #" || i
07   END FOR
08 END FUNCTION
```

Example 3: Global variables

File "myglobs.4gl":

```
01 GLOBALS
02   DEFINE userid CHAR(20)
03   DEFINE extime DATETIME YEAR TO SECOND
04 END GLOBALS
```

File "mylib.4gl":

```
01 GLOBALS "myglobs.4gl"
02
03 DEFINE s VARCHAR(100)
04
05 FUNCTION myfunc( )
06   DEFINE i INTEGER
07   DISPLAY "User Id = " || userid
08   FOR i=1 TO10
09       LET s = "item #" || i
10   END FOR
11 END FUNCTION
```

File "mymain.4gl":

```
01 GLOBALS "myglobs.4gl"
02
03 MAIN
04   LET userid = fgl_getenv("LOGNAME")
05   LET extime = CURRENT YEAR TO SECOND
06   CALL myfunc()
07 END MAIN
```

Constants

Summary:

- Definition
- Examples

See also: Variables, Records, Data Types, User Types.

Definition

Purpose:

A *constant* defines a read-only value identified by a name.

Syntax:

```
CONSTANT identifier [ datatype ] = value [,...]
```

Notes:

1. *identifier* is the name of the constant to be defined.
2. *datatype* can be any data type except complex types like TEXT or BYTE.
3. *value* can be an integer literal, a decimal literal, or a string literal. *value* cannot be NULL.

Usage:

You can declare a constant to define a static value that can be used in other instructions. Constants can be global, local to a module, or local to a function.

When declaring a constant, the data type specification can be omitted. The literal value automatically defines the data type:

```
01 CONSTANT c1 = "Drink" -- Declares a STRING constant
02 CONSTANT c2 = 4711    -- Declares an INTEGER constant
```

However, in some cases, you may need to specify the data type:

```
01 CONSTANT c1 SMALLINT = 12000 -- Would be an INTEGER by default
```

Constants can be used in variable, records, and array definitions:

```
01 CONSTANT n = 10
02 DEFINE a ARRAY[n] OF INTEGER
```

Constants can be used at any place in the language where you normally use literals:

```
01 CONSTANT n = 10
02 FOR i=1 TO n
```

Constants can be passed as function parameters, and returned from functions.

Warnings:

1. A constant cannot be used in the `ORDER BY` clause of a static `SELECT` statement, because the compiler considers identifiers after `ORDER BY` as part of the SQL statement (i.e. column names), not as constants.
   ```
   CONSTANT position = 3
   SELECT * FROM table ORDER BY position
   ```
2. Automatic date type conversion can take place in some cases:
   ```
   CONSTANT c1 CHAR(10) = "123"
   CONSTANT c2 CHAR(10) = "abc"
   DEFINE i INTEGER
   FOR i=1 TO c1 # Constant "123" is be converted to 123
   FOR i=1 TO c2 # Constant "abc" is converted to zero!
   ```
3. Character constants defined with a string literal that is longer than the length of the datatype are truncated:
   ```
   CONSTANT s CHAR(3) = 'abcdef'
   DISPLAY s # Displays "abc"
   ```
4. The compiler throws an error when the symbol used as a constant is not defined:
   ```
   DEFINE s CHAR(c) # Error, c is not defined!
   ```
5. The compiler throws an error when the symbol used as a constant is a variable:
   ```
   DEFINE c INTEGER
   DEFINE s CHAR(c) # Error, c is a variable!
   ```
6. The compiler throws an error when you try to assign a value to a constant:
   ```
   CONSTANT c INTEGER = 123
   LET c = 345 # Error, c is a constant!
   ```
7. The compiler throws an error when the symbol used is not defined as an integer constant:
   ```
   CONSTANT c CHAR(10) = "123"
   DEFINE s CHAR(c) # Error, c is not an integer!
   ```

Tips:

1. Define common special characters with constants:
   ```
   CONSTANT C_ESC  = '\x1b'
   CONSTANT C_TAB  = '\t'
   CONSTANT C_CR   = '\r'
   CONSTANT C_LF   = '\n'
   CONSTANT C_CRLF = '\r\n'
   ```

Examples

Example 1:

```
01 CONSTANT c1 ="Drink",   # Declares a STRING constant
02          c2 = 4711,     # Declares an INTEGER constant
03          n = 10,        # Declares an INTEGER constant
04          x SMALLINT = 1 # Declares a SMALLINT constant
05 DEFINE a ARRAY[n] OF INTEGER
06 MAIN
07   CONSTANT c1 = "Hello"
08   DEFINE i INTEGER
09   FOR i=1 TO n
10       ...
11   END FOR
12   DISPLAY c1 || c2  # Displays "Hello4711"
13 END MAIN
```

Records

Summary:

- Definition
- Examples

See also: Variables, Arrays, Data Types, Database Schema File.

Definition

Purpose:

A *record* defines a structured variable.

Syntax 1:

```
RECORD
  member { type | LIKE [dbname:]tabname.colname }
  [,...]
END RECORD
```

Syntax 2:

```
RECORD LIKE [dbname:]tabname.*
```

Notes:

1. *member* is an identifier for a record member variable.
2. *type* can be any data type, a record definition, or an array definition.
3. *dbname* identifies a specific database schema file.
4. *tabname* identifies a database table defined in the database schema file.
5. *colname* identifies a database column defined in the database schema file.

Usage:

A record is an ordered set of variables (called members), where each member can be of any data type, a record, or an array. Records whose members correspond in number, order, and data type compatibility to a database table can be useful for transferring data from the database to the screen, to reports, or to functions. In the first form (Syntax 1), record members are defined explicitly. In the second form (Syntax 2), record members are created implicitly from the table definition in the database schema file. The columns defined as `SERIAL` are converted to INTEGER.

Warning: When using the `LIKE` clause, the data types are taken from the database schema file during compilation. Make sure that the database schema file of the development database corresponds to the production database, otherwise the

records defined in the compiled version of your programs will not match the table structures of the production database. Statements like `SELECT * INTO record.* FROM table` would fail.

In the rest of the program, record members are accessed by a dot notation (`record.member`). The notation `record.member` refers to an individual member of a record. The notation `record.*` refers to the entire list of record members. The notation `record.first THRU record.last` refers to a consecutive set of members. (`THROUGH` is a synonym for `THRU`).

Records can be used as function parameters, and can be returned from functions.

It is possible to compare records having the same structure with the equal operator: `record1.* = record2.*`

Examples

Example 1:

```
01 MAIN
02   DEFINE rec RECORD
03               id INTEGER,
04               name VARCHAR(100),
05               birth DATE
06             END RECORD
07   LET rec.id = 50
08   LET rec.name = 'Scott'
09   LET rec.birth = TODAY
10   DISPLAY rec.*
11 END MAIN
```

Example 2:

```
01 SCHEMA stores
02 DEFINE cust RECORD LIKE customer.*
03 MAIN
04   SELECT * INTO cust.* FROM customer WHERE customer_num=2
05   DISPLAY cust.*
06 END MAIN
```

Arrays

Summary:

- Syntax
- Usage
- Examples

See also: Variables, Records, Data Types.

Syntax

Purpose:

Arrays can store a one-, two- or three-dimensional array of elements.

Syntax 1: Static array definition

```
ARRAY [ size [,size [,size] ] ] OF datatype
```

Syntax 2: Dynamic array definition

```
DYNAMIC ARRAY [ WITH DIMENSION rank ] OF datatype
```

Notes:

1. *size* can be an integer literal or an integer constant. The upper limit is **65535**.
2. *rank* can be an integer literal of 1, 2, or 3. Default is 1.
3. *datatype* can be any data type or a record definition.

Methods:

Object Methods	
Name	**Description**
`appendElement( )`	Adds a new element at the end of a dynamic array. This method has no effect on a static array.
`clear( )`	Removes all elements in a dynamic array. Sets all elements to NULL in a static array.
`deleteElement( INTEGER )`	Removes an element at the given position. In a static or dynamic array, the elements after the given position are moved up. In a dynamic array, the number of elements is

	decremented by 1.
`getLength( )` `  RETURNING INTEGER`	Returns the length of a one-dimensional array.
`insertElement( INTEGER )`	Inserts a new element at the given position. In a static or dynamic array, the elements after the given position are moved down. In a dynamic array, the number of elements is incremented by 1.

Usage

Arrays can store a one-, two- or three-dimensional array of variables, all of the same type. These can be any of the supported data types or a record definition, but it cannot be another array (`ARRAY .. OF ARRAY`).

The first syntax (`ARRAY[i[,j[,k]]]`) defines traditional static arrays, which are defined with an explicit size for all dimensions. Static arrays have a size limit. The biggest static arrays size you can define is **65535**.

Warning: Because of backward compatibility with Informix 4gl, all elements of static arrays are initialized, even if the array is not used. Therefore, it is not recommended that you define huge static arrays, as they can use a lot of memory.

The second syntax (`DYNAMIC ARRAY`) defines arrays with a variable size. Dynamic arrays have no theoretical size limit. The elements of dynamic arrays are allocated automatically by the runtime system, according to the indexes used.

```
01 MAIN
02   DEFINE a1 ARRAY[100] OF INTEGER -- This is a static array
03   DEFINE a2 DYNAMIC ARRAY OF INTEGER -- This is a dynamic array
04   LET a1[50] = 12456
05   LET a2[5000] = 12456  -- Automatic allocation for element 5000
06   LET a1[5000] = 12456  -- Runtime error!
07 END MAIN
```

Warning: A dynamic array element is allocated before it is used. For example, when you assign array element with the `LET` instruction, if the element does not exist, it is created automatically. This is also true when using a dynamic array in a `FOREACH` loop.

The elements of an array variable can be of any data type except an array definition, but an element can be a record that contains an array member.

```
01 MAIN
02   DEFINE arr ARRAY[50] OF RECORD
03              key INTEGER,
```

```
04            name CHAR(10),
05            address VARCHAR(200),
06            contacts ARRAY[50] OF VARCHAR(20)
07        END RECORD
08   LET arr[1].key = 12456
09   LET arr[1].name = "Scott"
10   LET arr[1].contacts[1] = "Bryan COX"
11   LET arr[1].contacts[2] = "Courtney FLOW"
12 END MAIN
```

A single array element can be referenced by specifying its coordinates in each dimension of the array.

Warning: For Informix 4gl compatibility, the compiler allows the .* notation to assign a dynamic array with a record structure to another dynamic array with the same structure, but the behavior is not clearly specified. Unlike simple records, the array is actually copied by reference. We strongly discourage to use the .* notation with dynamic arrays.

You cannot specify a *static array* as an argument or as a returned value of a function. However, dynamic arrays can be used as function parameter and will be passed by reference (i.e. the dynamic array can be modified inside the called function, and the caller will see the modifications).

In the `DEFINE` section of a REPORT statement, formal arguments cannot be declared as arrays, nor as record variables that contain array members.

If you reference an array element in an r-value, with an index outside the allocated dimensions, you get a **-1326** runtime error:

```
01 MAIN
02   DEFINE a DYNAMIC ARRAY OF INTEGER
03   LET a[50] = 12456
04   DISPLAY a[100]  -- Runtime error
05 END MAIN
```

Arrays can be queried with the `getLength()` method, to get the number of allocated elements:

```
01 MAIN
02   DEFINE a DYNAMIC ARRAY OF INTEGER
03   LET a[5000] = 12456
04   DISPLAY a.getLength()
05 END MAIN
```

You can insert a new element at a given position with the `insertElement()` method. The new element will be initialized to NULL. All subsequent elements are moved down by an offset of +1. Dynamic arrays will grow by 1, while static arrays will lose the last element:

```
01 MAIN
02   DEFINE a DYNAMIC ARRAY OF INTEGER
```

```
03   LET a[10] = 11
04   CALL a.insertElement(10)
05   LET a[10] = 10
06   DISPLAY a.getLength() -- shows 11
07   DISPLAY a[10]  -- shows 10
08   DISPLAY a[11]  -- shows 11
09 END MAIN
```

You can append a new element at the end of a dynamic array with the `appendElement()` method. The new element will be initialized to NULL. Dynamic arrays will grow by 1, while static arrays will not be affected by this method:

```
01 MAIN
02   DEFINE a DYNAMIC ARRAY OF INTEGER
03   LET a[10] = 10
04   CALL a.appendElement()
05   LET a[a.getLength()] = a.getLength()
06   DISPLAY a.getLength() -- shows 11
07   DISPLAY a[10]  -- shows 10
08   DISPLAY a[11]  -- shows 11
09 END MAIN
```

The `deleteElement()` method can be used to remove elements from a static or dynamic array. Subsequent elements are moved up by an offset of -1. Dynamic arrays will shrink by 1, while static arrays will have NULLs in the last element.

```
01 MAIN
02   DEFINE a DYNAMIC ARRAY OF INTEGER
03   LET a[10] = 9
04   CALL a.deleteElement(5)
06   DISPLAY a.getLength() -- shows 9
07   DISPLAY a[9]  -- shows 9
08 END MAIN
```

You can clear an array with the `clear()` method. When used on a static array, this method sets all elements to NULL. When used on a dynamic array, it removes all elements:

```
01 MAIN
02   DEFINE a DYNAMIC ARRAY OF INTEGER
03   LET a[10] = 11
04   DISPLAY a.getLength() -- shows 10
05   CALL a.clear()
06   DISPLAY a.getLength() -- shows 0
07 END MAIN
```

When used as a function parameter, static arrays are passed by value, while dynamic arrays are passed by reference:

```
01 MAIN
02   DEFINE a DYNAMIC ARRAY OF INTEGER
03   CALL fill(a)
04   DISPLAY a.getLength() -- shows 2
```

```
05 END MAIN
06 FUNCTION fill(x)
07   DEFINE x DYNAMIC ARRAY OF INTEGER
08   CALL x.appendElement()
09   CALL x.appendElement()
10 END FUNCTION
```

Array methods can be used on two- and three-dimensional arrays with the brackets notation:

```
01 MAIN
02   DEFINE a2 DYNAMIC ARRAY WITH DIMENSION 2 OF INTEGER
03   DEFINE a3 DYNAMIC ARRAY WITH DIMENSION 3 OF INTEGER
04   LET a2[50,100]  = 12456
05   LET a2[51,1000] = 12456
06   DISPLAY a2.getLength()         -- shows 51
07   DISPLAY a2[50].getLength()     -- shows 100
08   DISPLAY a2[51].getLength()     -- shows 1000
09   LET a3[50,100,100]  = 12456
10   LET a3[51,101,1000] = 12456
11   DISPLAY a3.getLength()         -- shows 51
12   DISPLAY a3[50].getLength()     -- shows 100
13   DISPLAY a3[51].getLength()     -- shows 101
14   DISPLAY a3[50,100].getLength() -- shows 100
15   DISPLAY a3[51,101].getLength() -- shows 1000
16   CALL a3[50].insertElement(10)  -- inserts at 50,10
17   CALL a3[50,10].insertElement(1)-- inserts at 50,10,1
18 END MAIN
```

Examples

Example 1: Using static and dynamic arrays.

```
01 MAIN
02   DEFINE a1 DYNAMIC ARRAY OF INTEGER
03   DEFINE a2 DYNAMIC ARRAY WITH DIMENSION 2 OF INTEGER
04   DEFINE a3 ARRAY[10,20] OF RECORD
05               id INTEGER,
06               name VARCHAR(100),
07               birth DATE
08             END RECORD
09   LET a1[5000] = 12456
10   LET a2[5000,300] = 12456
11   LET a3[5,1].id = a1[50]
12   LET a3[5,1].name = 'Scott'
13   LET a3[5,1].birth = TODAY
14 END MAIN
```

Example 2: Here is the recommended way to fetch rows into a dynamic array.

```
01 SCHEMA stores
02 MAIN
```

```
03   DEFINE a DYNAMIC ARRAY OF RECORD LIKE customer.*
04   DEFINE r RECORD LIKE customer.*
05   DATABASE stores
06   DECLARE c CURSOR FOR SELECT * FROM customer
07   FOREACH c INTO r.*
08     CALL a.appendElement()
09     LET a[a.getLength()].* = r.*
10   END FOREACH
11   DISPLAY "Rows found: ", a.getLength()
12 END MAIN
```

User Types

Summary:

- Definition
- Examples

See also: Variables, Records, Data Types, Constants.

Definition

Purpose:

A *user type* is a data type based on built-in types, records or arrays.

Syntax:

```
TYPE identifier definition
```

Notes:

1. *identifier* is the name of the user type to be defined.
2. *definition* is any data type, record structure, or array definition supported by the language.

Usage:

You can define a user type as a synonym for an existing data type, or as a shortcut for records and array structures.

After declaring a user type, it can be used as a normal data type to define variables.

The scope of a user type is the same as for variables and constants. Types can be global, local to a module, or local to a function.

Examples

Example 1:

```
01 TYPE customer RECORD
02         cust_num INTEGER,
03         cust_name VARCHAR(50),
04         cust_addr VARCHAR(200)
05     END RECORD
```

```
06 DEFINE c customer
```

Example 2:

The following example defines the user type in a globals file and then uses the type in a report program:

```
01 -- typeglobals.4gl
02
03 GLOBALS
04  TYPE rpt_order RECORD
05        order_num  INTEGER,
06        store_num  INTEGER,
07        order_date DATE,
08        fac_code   CHAR(3)
09     END RECORD
10 END GLOBALS
```

```
01 -- report1.4gl
02
03 GLOBALS "typeglobals.4gl"
04
05 MAIN
06   DEFINE o rpt_order
07   CONNECT TO "custdemo"
08   DECLARE order_c CURSOR FOR
09     SELECT order_num,
10            store_num,
11            order_date,
12            fac_code
13       FROM orders
14   START REPORT order_list
15   FOREACH order_c INTO o.*
16     OUTPUT TO REPORT order_list(o.*)
17   END FOREACH
18   FINISH REPORT order_list
19   DISCONNECT CURRENT
20 END MAIN
21
22 REPORT order_list(ro)
23   DEFINE ro rpt_order
24
25   FORMAT
26     ON EVERY ROW
27      PRINT ro.order_num,
...
```

Data Conversions

Summary:

- Data Conversion
- Conversion table

See also: Data Types.

Data Conversion

The runtime system performs data conversion implicitly without objection, as long as the data conversion is valid. For example, when you assign a number expression to a character variable, the runtime system converts the resulting number to a literal string.

Conversion rules apply to variable assignment, function parameters, and returned values.

Data conversions from or to character string values involve environment and locale settings like DBDATE, DBFORMAT.

When using the default exception handler, if a conversion error occurs, STATUS is zero and the target is set to zero for `SMALLINT`, `INTEGER`, `SMALLFLOAT` and `FLOAT` data types or NULL for all other data types.

Warning: The global STATUS variable is not set when a conversion error occurs unless you have enabled `ANY ERROR` detection with the WHENEVER instruction.

Conversion Table

The following table shows which pairs of data types are compatible.

Conversion table:

Columns represent source data types and lines represent receiving data types.														
	char	**varchar**	**string**	**integer**	**smallint**	**float**	**smallfloat**	**decimal**	**money**	**date**	**datetime**	**interval**	**text**	**byte**
char	1	1	1	1	1	1	1	1	1, 8	1, 9	1	1	15	15
varchar	1	1	1	1	1	1	1	1	1, 8	1, 9	1	1	15	15
string	1	1	1	1	1	1	1	1	1, 8	1, 9	1	1	15	15
integer	2, 3	2, 3	2, 3			3, 4	3, 4	3, 4	3, 4	11	15	15	15	15
smallint	2, 3	2, 3	2, 3	3		3, 4	3, 4	3, 4	3, 4	3,	15	15	15	15

										11				
float	2, 3, 5	2, 3, 5	2, 3, 5	10	10			3	3	11	15	15	15	15
smallfloat	2, 3, 5	2, 3, 5	2, 3, 5	5, 10	10	5		3, 5	3, 5	5, 11	15	15	15	15
decimal	2, 3, 6	2, 3, 6	2, 3, 6	3	3	3, 6	3, 6	3, 6	3, 6	3, 11	15	15	15	15
money	2, 3, 6	2, 3, 6	2, 3, 6	3	3	3, 6	3, 6	3, 6	3, 6	3, 11	15	15	15	15
date	2	2	2	11	11	3,4,11	3,4,11	3,4,11	3,4,11		12, 14	15	15	15
datetime	2	2	2	15	15	15	15	15	15	13, 14	7, 14	15	15	15
interval	2	2	2	15	15	15	15	15	15	15	15	3, 7	15	15
text	1	1	1	15	15	15	15	15	15	15	15	15		15
byte	15	15	15	15	15	15	15	15	15	15	15	15	15	

Notes:

1. If the result of converting a value to a character string is longer than the receiving variable, the character string is truncated from the right.
2. Character string values must depict a literal of the receiving data type.
3. If the value exceeds the range of the receiving data type, an overflow error occurs.
4. Any fractional part of the value is truncated.
5. If the passed value contains more significant digits than the receiving data type supports, low-order digits are discarded.
6. If the passed value contains more fractional digits than the receiving data type supports, low-order digits are discarded.
7. Differences in qualifiers can cause truncation from the left or right.
8. DBMONEY and DBFORMAT control the format of the converted string.
9. DBFORMAT, DBDATE, or GL_DATE controls the format of the result.
10. Rounding errors can produce an assigned value with fractional parts.
11. An integer value corresponding to a count of days is assigned.
12. An implicit EXTEND(value, YEAR TO DAY) is performed.
13. The DATE becomes a DATETIME YEAR TO DAY literal before assignment.
14. If the passed value has less precision than the receiving variable, any missing time unit values are obtained from the system clock.
15. **Unsupported conversion.**

Built-in Classes

Summary:

- Purpose
- Syntax
- Usage
  - base Package classes
  - ui Package classes
  - om Package classes
  - utils Package classes
  - os Package classes
  - Class and Object Methods
  - Working with Objects

See also: Variables, Functions.

Purpose

Built-in classes, grouped into packages, are predefined object templates that are provided by the runtime system. Each class contains methods that interact with a specific program object, allowing you to change the appearance or behavior of the object. The classes provide the benefits of OOP programming in 4GL.

Syntax

Syntax 1: Defining an object

```
obj package.classname
```

Syntax 2: Using a class method

```
package.classname.method
```

Syntax 3: Using an object method

```
obj.method
```

Notes:

1. *obj* is the name of the variable defined for the object
2. *package* is the name of the package the class comes from.
3. *classname* is the name of the built-in class.
4. *method* is the name of the method

Usage

Package: base

Classes	Purpose
Application	Provides an interface to the application internals
Channel	Provides basic read/write functionality (files/communication)
StringBuffer	Allows manipulation of character strings
StringTokenizer	Allows parsing of strings to extract tokens
TypeInfo	Provides serialization of program variables

Package: ui

Classes	Purpose
Interface	Provided to manipulate the user interface
Window	Provides an interface to the Window objects
Form	Provides an interface to the forms used by the program
Dialog	Provides an interface to the interactive instructions
ComboBox	Provides an interface to the ComboBox formfield view

Package: om

Classes	Purpose
DomDocument	Provides methods to manipulate a DOM data tree
DomNode	Provides methods to manipulate a node of a DOM data tree
NodeList	Holds a list of selected DomNode objects
SaxAttributes	Provides methods to manipulate XML element attributes
SaxDocumentHandler	Provides methods to write an XML filter
XmlReader	Provides methods to read and process a file written in XML format
XmlWriter	Provides methods to write XML documents to different types of output

Package: util

Classes	Purpose
Math	Provides an interface for mathematical functions

The *util* package is a Dynamic C Extension library; part of the standard package. To use the Math class, you must import the library in your program:

Package: os

Classes	Purpose
Path	Provides functions to manipulate files and directories on the machine where the BDL program executes

The *os* package is a Dynamic C Extension library; part of the standard package. To use the Path class, you must import the library in your program:

Methods

There are two types of methods: *Class Methods* and *Object Methods*. Methods can be invoked like global functions, by passing parameters and/or returning values.

Class Methods - you call these methods using the ***class identifier*** (package name + class name) as the prefix, with a period as the separator.

```
01 CALL ui.Interface.refresh()
```

The method `refresh()` is a Class Method of the **Interface** class, which is part of the **ui** package.

Object Methods - To use these methods, the object must exist. After an object has been created, you can call the Object Methods in the class by using the **object variable** as a prefix, with a period as the separator:

```
01 LET b = n.getDocumentElement()
```

The method `getDocumentElement()` is an Object Method of the class to which the *n* object belongs.

Working with Objects

To handle an object in your program, you

- define an object variable using the class identifier.
- instantiate the object (create it) before using it. You usually instantiate objects with a Class Method.
- once the object exists, you can call the Object methods of the class.

```
01 DEFINE n om.DomDocument, b DomNode
02 LET n = om.DomDocument.create("Stock")
03 LET b = n.getDocumentElement()
```

The object *n* is instantiated using the `create()` Class Method of the DomDocument class. The object *b* is instantiated using the `getDocumentElement()` Object method of the DomDocument class. This method returns the DomNode object that is the root node of the DomDocument object *n*.

The object variable only contains the reference to the object. For example, when passed to a function, only the reference to the object is copied onto the stack.

You do not have to destroy objects. This is done automatically by the runtime system for you, based on a reference counter.

```
01 MAIN
02    DEFINE d om.DomDocument
03    LET d = om.DomDocument.create("Stock")  -- Reference counter = 1
05 END MAIN  -- d is removed, reference counter = 0 => object is
destroyed.
```

You can pass object variables to functions or return them from functions. Objects are passed ***by reference*** to functions. In the following example, the function creates the object and returns its reference on the stack:

```
01 FUNCTION createStockDomDocument( )
02    DEFINE d om.DomDocument
03    LET d = om.DomDocument.create("Stock")  -- Reference counter = 1
04    RETURN d
05 END FUNCTION  -- Reference counter is still 1 because d is on the
stack
```

Another part of the program can get the result of that function and pass it as a parameter to another function.

Example:

```
01 MAIN
02    DEFINE x om.DomDocument
03    LET x = createStockDomDocument( )
04    CALL writeStockDomDocument( x )
05 END MAIN
06
07 FUNCTION createStockDomDocument( )
08    DEFINE d om.DomDocument
09    LET d = om.DomDocument.create("Stock")
```

```
10    RETURN d
11 END FUNCTION
12
13 FUNCTION writeStockDomDocument( d )
14    DEFINE d om.DomDocument
15    DEFINE r om.DomNode
16    LET r = d.getDocumentElement()
17    CALL r.writeXml("Stock.xml")
18 END FUNCTION
```

Compiling Programs

Summary:

- Compiling Source Code
- Creating Libraries
- Linking Programs
- Using Makefiles
- Getting Build Information

See also: Tools, Form Files, Message Files, Localized Strings.

Compiling Source Code

Source code modules (**4gl**) must be compiled to p-code modules (**42m**) with the fglcomp tool. Compiled p-code modules are portable; you can compile a module on a Windows platform and install it on a Unix production machine.

The following lines show a compilation in a Unix shell session:

```
$ cat xx.4gl
main
  display "hello"
end main

$ fglcomp xx.4gl

$ ls -s xx.42m
   4 xx.42m
```

If an error occurs, the compiler writes an error file with the **.err** extension.

```
$ cat xx.4gl
main
  let x = "hello"
end main

$ fglcomp xx.4gl
Compilation was not successful.  Errors found: 1.
 The file xx.4gl has been written.

$ cat xx.err
main
  let x = "hello"
| The symbol 'x' does not represent a defined variable.
| See error number -4369.
end main
```

With the `-M` option, you can force the compiler to display an error message instead of generating a **.err** error file:

```
$ fglcomp xx.4gl
xx.4gl:2:8 error:(-4369) The symbol 'x' does not represent a defined
variable.
```

By default, the compiler does not raise any warnings. You can turn on warnings with the `-W` option:

```
$ cat xx.4gl
main
  database test1
  select count(*) from x, outer(y) where x.k = y.k
end main

$ fglcomp -W stdsql xx.4gl
xx.4gl:3: warning: SQL statement or language instruction with specific
SQL syntax.
```

When a warning is raised, you can use the `-W error` option to force the compiler to stop as if an error was found.

For more details about warning options, see the fglcomp tool.

Creating Libraries

Compiled **42m** modules can be grouped in a library file using the **42x** extension.

Library linking is done with the fglrun tool by using the `-l` option. You can also use the fgllink tool.

The following lines show a link procedure to create a library in a Unix shell session:

```
$ fglcomp fileutils.4gl
$ fglcomp userutils.4gl
$ fgllink -o libutils.42x fileutils.42m userutils.42m
```

When you create a library, all functions of the **42m** modules used in the link command are registered in the **42x** file.

Warning: The 42x library file does not contain the 42m files. When deploying your application, you must provide all p-code modules as well as 42f, 42r and 42x files.

The **42x** libraries are typically used to link the final **42r** programs:

```
$ fglcomp mymain.4gl
$ fgllink -o myprog.42r mymain.42m libutils.42x
```

Note that **42r** programs must be re-linked if the content of **42x** libraries have changed. In the above example, if a function of the **userutils.4gl** source file was removed, you must recompile **userutils.4gl**, re-link the **libutils42x** library and re-link the **myprog.42r** program.

If you are using C Extensions, you may need to use the **-e** option to specify the list of extension modules if the IMPORT keyword is not used:

```
$ fgllink -e extlib,extlib2,extlib3 -o libutils.42x fileutils.42m
userutils.42m
```

Linking Programs

Genero programs are created by linking several **42m** modules and/or **42x** libraries together, to produce a file with the **42r** extension.

Program linking is done with the fglrun tool by using the `-l` option. You can also use the fgllink tool.

Warning: The 42r program file does not contain the 42m files. When deploying your application, you must provide all p-code modules as well as 42f, 42r and 42x files.

The following lines show a link procedure to create a library in a Unix shell session:

```
$ fglcomp main.4gl
$ fglcomp store.4gl
$ fgllink -o stores.42r main.42m store.42m
```

By default, if you do not specify an absolute path for a file, the linker searches for **42m** modules and **42x** libraries in the current directory.

Additionally, you can specify a search path with the FGLLDPATH environment variable:

```
$ FGLLDPATH=/usr/dev/lib/maths:/usr/dev/lib/utils
$ export FGLLDPATH
$ ls /usr/dev/lib/maths
mathlib1.42x
mathlib2.42x
mathmodule11.42m
mathmodule12.42m
mathmodule22.42m
$ ls /usr/dev/lib/utils
fileutils.42m
userutils.42m
dbutils.42m
$ fgllink -o myprog.42r mymodule.42m mathlib1.42x fileutils.42m
```

In this example the linker will find the specified files in the `/usr/dev/lib/maths` and `/usr/dev/lib/utils` directories defined in FGLLDPATH.

If you are using C Extensions, you may need to use the -e option to specify the list of extension modules if the IMPORT keyword is not used:

```
$ fgllink -e extlib,extlib2,extlib3 -o stores.42r main.42m store.42m
```

Warning: If none of the functions of a module are used by a program, the complete module is excluded when the program is linked. This may cause undefined function errors at runtime, such as when a function is only used in a dynamic call (an initialization function, for example.)

The following case illustrates this behavior:

```
$ cat x1.4gl
function fx1A()
end function
function fx2A()
end function

$ cat x2.4gl
function fx2A()
end function
function fx2B()
end function

$ cat prog.4gl
main
  call fx1A()
end main

$ fglcomp x1.4gl
$ fglcomp x2.4gl
$ fglcomp prog.4gl

$ fgllink -o lib.42x x1.42m x2.42m

$ fgllink -o prog.42r prog.42m lib.42x
```

Here, module `x1.42m` will be included in the program, but module `x2.42m` will not. At runtime, any dynamic call to `fx2A()` or `fx2B()` will fail.

The link process searches *recursively* for the functions used by the program. For example, if the MAIN block calls function FA in module MA, and FA calls FB in module MB, all functions from module MA and MB will be included in the **42r** program definition.

Using Makefiles

Most UNIX platforms provide the **make** utility program to compile projects. The **make** program is an interpreter of *Makefiles*. These files contain directives to compile and link programs and/or generate other kind of files.

When developing on Microsoft Windows platforms, you may use the **NMAKE** utility provided with Visual C++, however this tool does not have the same behavior as the Unix make program. To have a compatible make on Windows, you can install a GNU make or third party Unix tools such as Cygwin.

For more details about the **make** utility, see the platform-specific documentation.

The follow example shows a typical *Makefile* for Genero applications:

```
#-----------------------------------------------------
# Generic makefile rules to be included in Makefiles

.SUFFIXES: .42s .42f .42m .42r .str .per .4gl .msg .hlp

FGLFORM=fglform -M
FGLCOMP=fglcomp -M
FGLLINK=fglrun -l
FGLMKMSG=fglmkmsg
FGLMKSTR=fglmkstr
FGLLIB=$$FGLDIR/lib/libfgl4js.42x

all::

.msg.hlp:
        $(FGLMKMSG) $*.msg $*.hlp

.str.42s:
        $(FGLMKSTR) $*.str $*.42s

.per.42f:
        $(FGLFORM) $*.per

.4gl.42m:
        $(FGLCOMP) $*.4gl

clean::
        rm -f *.hlp *.42? *.out


#----------------------------
# Makefile example

include Makeincl

FORMS=\
 customers.42f\
 orderlist.42f\
 itemlist.42f
```

```
MODULES=\
 customerInput.42m\
 zoomOrders.42m\
 zoomItems.42m

customer.42x: $(MODULES)
        $(FGLLINK) -o customer.42x $(MODULES)
all:: customer.42x $(FORMS)
```

Getting Build Information

The compiler version used to build the **42m** modules must be compatible to the runtime system used to execute the programs. The fglcomp compiler writes version information in the generated **42m** files. This can be useful on site, if you need to check the version of the compiler that was used to build the **42m** nodules.

To extract build information, run fglrun with the `-b` option:

```
$ fglrun -b mymodule.42m
2.11.01-1161.12 /home/devel/stores/mymodule.4gl 15
```

The output shows the following fields:

1. The product version and build number (`2.11.01-1161.12`).
2. The full path of the source file (/home/devel/stores/mymodule.4gl).
3. The internal identifier of the pcode version.

Tip: Since version 2.11, fglrun -b can read the header of pcode modules compiled with older versions of fglcomp and display version information for such old modules. If fglrun cannot recognize a pcode module, it returns with an execution status is different from zero.

When reading build information of a 42x or 42r file, fglrun scans all modules used to build the library or program. You will see different versions in the first column if the modules where compiled with different versions of fglcomp. Note however that it's not recommended to end up with mixed versions on a production site:

```
$ fglrun -b myprogram.42r
2.11.01-1161.12 /home/devel/stores/mymodule1.4gl 15
2.10.02-1148.36 /home/devel/stores/mymodule2.4gl 15
2.11.01-1161.12 /home/devel/stores/mymodule3.4gl 15
```

Warning: Before release 2.10, the 42m p-code files were also stamped with a compilation timestamp. This timestamp information is no longer written to p-code files, allowing 42m file comparison, checksum creation, or storage of 42m file in versioning tools; the same p-code data is now generated after successive compilations.

To check the version of the runtime system, run fglrun with the `-V` option.

Programs

Summary:

- Runtime Configuration
- The MAIN block
- Signal Handling (DEFER)
- The STATUS variable
- The INT_FLAG variable
- The QUIT_FLAG variable
- Importing modules (IMPORT)
- Program Options (OPTIONS)
  - OPTIONS form-element LINE
  - OPTIONS DISPLAY ATTRIBUTES
  - OPTIONS INPUT ATTRIBUTES
  - OPTIONS INPUT WRAP
  - OPTIONS ON TERMINATE
  - OPTIONS ON CLOSE APPLICATION
  - OPTIONS HELP FILE
  - OPTIONS FIELD ORDER
  - OPTIONS control keys
  - OPTIONS RUN IN
- Running Programs (RUN)
- Stop Program Execution (EXIT PROGRAM)
- Database Schema Specification (SCHEMA)
- The NULL Constant
- The TRUE Constant
- The FALSE Constant
- The NOTFOUND Constant
- The BREAKPOINT instruction
- Setting Key Labels
- Responding to CTRL_LOGOFF_EVENT

See also: Compiling Programs, Preprocessor, Database Schema Files, Flow Control, The Application class, Localized Strings.

Runtime Configuration

You can control the behavior of the runtime system with some FGLPROFILE configuration parameters.

Intermediate field trigger execution

```
Dialog.fieldOrder = {true|false}
```

When this parameter is set to **true**, intermediate triggers are executed. As the user moves to a new field with a mouse click, the runtime system executes the BEFORE

`FIELD` / `AFTER FIELD` triggers of the input fields between the source field and the destination field. When the parameter is set to **false**, intermediate triggers are not executed.

For new applications, it is recommended that you set the parameter to **false**. GUI applications allow users to jump from one field to any other field of the form by using the mouse. Therefore, it makes no sense to execute the `BEFORE FIELD` / `AFTER FIELD` triggers of intermediate fields.

Important note: The default setting for the runtime system is **false**; while the default setting in FGLPROFILE for `Dialog.fieldOrder` is **true**. As a result, the overall setting after installation is **true**. To modify the behavior of intermediate field trigger execution, change the setting of `Dialog.fieldOrder` in FGLPROFILE to **false**.

Warning: The `Dialog.fieldOrder` configuration parameter is ignored when the dialog uses the FIELD ORDER FORM option.

Make current row visible after sort in tables

```
Dialog.currentRowVisibleAfterSort = {true|false}
```

When this parameter is set to **true**, the offset of table page is automatically adapted to show the current row after a sort. By default, the offset is not changed and current row may not be visible after sorting rows of a table. Changing this parameter has no impact on existing code, it is just an indicator to force the dialog to shift to the page of rows having the current row, as if the end-user had scrollbar. You can use this parameter to get the same behavior as well known e-mail readers.

The MAIN block

Purpose:

The `MAIN` block is the starting point of the application. When the runtime system executes a program, after some initialization, it gives control to this program block.

Syntax:

```
MAIN
  [ define-statement | constant-statement ]
  { [defer-statement] | fgl-statement | sql-statement }
  [...]
END MAIN
```

Notes:

1. *define-statement* defines function arguments and local variables.
2. *constant-statement* can be used to declare local constants.

3. *defer-statement* defines how to handle signals in the program.
4. *fgl-statement* is any instruction supported by the language.
5. *sql-statement* is any static SQL instruction supported by the language.

Signal Handling

Purpose:

The `DEFER` instruction allows you to control the behavior of the program when an *interruption* or *quit* signal has been received.

Syntax:

```
DEFER { INTERRUPT | QUIT }
```

Warnings:

1. `DEFER INTERRUPT` and `DEFER QUIT` instructions can only appear in the MAIN block.
2. Once deferred, you cannot reset to the default behavior.

Usage:

`DEFER INTERRUPT` indicates that the program must continue when it receives an *interrupt* signal. By default, the program stops when it receives an *interrupt* signal.

When an *interrupt* signal is caught by the runtime system and `DEFER INTERRUPT` is used, the `INT_FLAG` global variable is set to TRUE by the runtime system.

Interrupt signals are raised on terminal consoles when the user presses a key like CTRL-C, depending on the stty configuration. When a BDL program is displayed through a front end, no terminal console is used; therefore, users cannot send interrupt signals with the CTRL-C key. To send an interruption request from the front end, you must define an 'interrupt' action view. For more details, refer to Interruption Handling in the Dynamic User Interface.

`DEFER QUIT` indicates that the program must continue when it receives a *quit* signal. By default, the program stops when it receives a *quit* signal.

When a *quit* signal is caught by the runtime system and `DEFER QUIT` is used, the QUIT_FLAG global variable is set to TRUE by the runtime system.

STATUS

Purpose:

`STATUS` is a predefined variable that contains the execution status of the last instruction.

Syntax:

```
STATUS
```

Definition:

```
DEFINE STATUS INTEGER
```

Notes:

1. The data type of `STATUS` is INTEGER.
2. `STATUS` is typically used with `WHENEVER ERROR CONTINUE` (or `CALL`).
3. `STATUS` is set by expression evaluation errors only when `WHENEVER ANY ERROR` is used.
4. After an SQL statement execution, `STATUS` contains the value of SQLCA.SQLCODE. Use `SQLCA.SQLCODE` for SQL error management, and `STATUS` for 4gl errors.

Warnings:

1. `STATUS` is updated after any instruction execution. A typical mistake is to test `STATUS` after a `DISPLAY STATUS` instruction, written after an SQL statement.
2. While `STATUS` can be modified by hand, it is not recommended, as `STATUS` may become read-only in a later release.

Example:

```
01 MAIN
02   DEFINE n INTEGER
03   WHENEVER ANY ERROR CONTINUE
04   LET n = 10/0
05   DISPLAY STATUS
06 END MAIN
```

INT_FLAG

Purpose:

`INT_FLAG` is a predefined variable that is automatically set to TRUE when the user presses the interruption key.

Syntax:

```
INT_FLAG
```

Definition:

```
DEFINE INT_FLAG INTEGER
```

Notes:

1. The data type of `INT_FLAG` is INTEGER.
2. `INT_FLAG` is typically used with DEFER INTERRUPT.
3. `INT_FLAG` is set to TRUE when an interruption event is detected by the runtime system. The interruption event is raised when the user presses the interruption key.
4. If `DEFER INTERRUPT` is enabled and the interruption event arrives during a procedural instruction (`FOR` loop), the runtime system sets `INT_FLAG` to `TRUE`; it is up to the programmer to manage the interruption event (stop or continue with the procedure).
5. If `DEFER INTERRUPT` is enabled and the interruption event arrives during an interactive instruction (`INPUT`, `CONSTRUCT`), the runtime system sets `INT_FLAG` to `TRUE` and exits from the instruction. It is recommended that you test `INT_FLAG` after an interactive instruction to check whether the input has been cancelled.

Warnings:

1. Once `INT_FLAG` is set to `TRUE`, it must be reset to `FALSE` to detect a new interruption event. It is the programmer's responsibility to reset the `INT_FLAG` to `FALSE`.

Example:

```
01 MAIN
02   DEFINE n INTEGER
03   DEFER INTERRUPT
04   LET INT_FLAG = FALSE
05   FOR n = 1 TO 1000
06      IF INT_FLAG THEN EXIT FOR END IF
07      ...
08   END FOR
09 END MAIN
```

QUIT_FLAG

Purpose:

`QUIT_FLAG` is a predefined variable that is automatically set to TRUE when a 'quit event' arrives.

Syntax:

```
QUIT_FLAG
```

Definition:

```
DEFINE QUIT_FLAG INTEGER
```

Notes:

1. The data type of `QUIT_FLAG` is INTEGER.
2. `QUIT_FLAG` is typically used with DEFER QUIT.
3. `QUIT_FLAG` is set to TRUE when a quit event is detected by the runtime system. The quit event is raised when the user presses the QUIT signal key (Control+Backslash).
4. If `DEFER QUIT` is enabled and the quit event arrives during a procedural instruction (`FOR` loop), the runtime system sets `QUIT_FLAG` to `TRUE`. It is the programmer's responsibility to manage the quit event (whether to stop or continue with the procedure).
5. If `DEFER QUIT` is enabled and the quit event arrives during an interactive instruction (`INPUT`, `CONSTRUCT`), the runtime system sets `QUIT_FLAG` to `TRUE` and exits from the instruction. It is recommended that you test `QUIT_FLAG` after an interactive instruction to check whether the input has been cancelled.

Warnings:

1. Once `QUIT_FLAG` is set to `TRUE`, it must be reset to `FALSE` to detect a new quit event. It is the programmer's responsibility to reset the `QUIT_FLAG` to `FALSE`.

Example:

```
01 MAIN
02   DEFER QUIT
03   LET QUIT_FLAG = FALSE
04   INPUT BY NAME ...
05   IF QUIT_FLAG THEN
06      ...
07   END IF
08 END MAIN
```

Importing modules

Purpose:

The `IMPORT` instruction declares a C extension to be used by the current module.

Syntax:

```
IMPORT filename [,...]
```

Notes:

1. *filename* is the identifier (without the file extension) of the module to be loaded.

Usage:

The `IMPORT` instruction must be used to declare a C extension implementing functions or variables used by the current module.

Modules declared with the `IMPORT` instruction do not have to be linked to create a program. The runtime system automatically loads dependent modules.

The FGLLDPATH environment variable specifies the directories to search for the modules.

By default, the runtime system tries to load a module with the name **userextension,** if it exists. This simplifies the migration of existing C extensions; you just need to create a shared library named userextension.so (or userextension.dll on Windows), and copy the file to one of the directories defined in FGLLDPATH.

Warnings:

1. The `IMPORT` instruction must be the first instruction in the current module. If you specify this instruction after `DEFINE`, `CONSTANT` or `GLOBALS`, you will get a syntax error.
2. The compiler converts the module name specified by `IMPORT` to lowercase letters (if you write `IMPORT MyModule`, the name is stored as "mymodule" in the pcode.) When the module is loaded at runtime, the filename must match the lowercase name. Therefore, you must always use lowercase filenames for modules. Using lowercase filenames also simplifies distribution on platforms like Windows, where filenames are not case-sensitive.

Example:

```
01 IMPORT mylib1, mylib2
02 DEFINE filename STRING
03 MAIN
04   CALL func1()  -- function defined in mylib1
05 END MAIN
```

See also: C Extensions.

Program Options

Purpose:

The OPTIONS instruction allows you to change default program options.

Syntax:

```
OPTIONS
{ INPUT [NO] WRAP
| HELP FILE help-filename
| INPUT ATTRIBUTE ( {FORM|WINDOW|input-attributes} )
| DISPLAY ATTRIBUTE ( {FORM|WINDOW|display-attributes} )
| SQL INTERRUPT {ON|OFF}
| FIELD ORDER {CONSTRAINED|UNCONSTRAINED|FORM}
| ON TERMINATE SIGNAL CALL user-function
| ON CLOSE APPLICATION {CALL user-function|STOP}
| RUN IN {FORM|LINE} MODE
| MESSAGE LINE line-value TUI Only!
| COMMENT LINE {OFF|line-value} TUI Only!
| PROMPT LINE line-value TUI Only!
| ERROR LINE line-value TUI Only!
| FORM LINE line-value TUI Only!
| INSERT KEY key-name TUI Only!
| DELETE KEY key-name TUI Only!
| NEXT KEY key-name TUI Only!
| PREVIOUS KEY key-name TUI Only!
| ACCEPT KEY key-name TUI Only!
| HELP KEY key-name TUI Only!
} [,...]
```

Notes:

1. The effect of the OPTIONS instruction is global for the entire program.
2. Most program options can be changed during the program execution.

Usage:

A program can include several OPTIONS statements. If these statements conflict in their specifications, the OPTIONS statement most recently encountered at runtime prevails. OPTIONS can specify the following features of other statements (such as CONSTRUCT, DISPLAY, DISPLAY ARRAY, DISPLAY FORM, ERROR, INPUT, INPUT ARRAY, MESSAGE, OPEN FORM, OPEN WINDOW, PROMPT and RUN):

- Positions of the reserved lines
- Input and display attributes
- Logical key assignments
- The name of the Help file
- SQL statement interruption
- Field traversal constraints

- Default screen display mode

Defining the default position of reserved lines TUI Only!

The following options define the positions of reserved lines in TUI mode. Is it not recommended that you use these options in GUI mode, as most have no effect on the display.

- `COMMENT LINE` specifies the position of the Comment line. The comment line displays messages defined with the `COMMENT` attribute in the form specification file. The default is (`LAST-1`) for the SCREEN, and `LAST` for all other windows. You can hide the comment line with `COMMENT LINE OFF`.
- `ERROR LINE` specifies the position on the screen of the Error line that displays the text of the ERROR statement. The default is the `LAST` line of the SCREEN.
- `FORM LINE` specifies the position of the first line of a form. The default is (`FIRST+2`), or line 3 of the current window.
- `MENU LINE` specifies the position of the Menu line. This displays the menu name and options, as defined by the MENU statement. The default is the `FIRST` line in the window.
- `MESSAGE LINE` specifies the position of the Message line. This reserved line displays the text of the MESSAGE statement. The default is (`FIRST+1`), or line 2 of the current window.
- `PROMPT LINE` specifies the position of the Prompt line where the text of PROMPT statements is displayed. The default value is the `FIRST` line in the window.

You can specify any of the following positions for each reserved line:

Expression	Description
`FIRST`	The first line of the screen or window.
`FIRST + integer`	A relative line position from the first line.
integer	An absolute line position in the screen or window.
`LAST - integer`	A relative line position from the last line.
`LAST`	The last line of the screen or window.

Defining the default display and the input attributes

Any attribute defined by the `OPTIONS` statement remains in effect until the runtime system encounters a statement that redefines the same attribute. This can be another OPTIONS statement, or an `ATTRIBUTE` clause in one of the following statements:

- CONSTRUCT, INPUT, DISPLAY, DIALOG, INPUT ARRAY or DISPLAY ARRAY
- OPEN WINDOW

The `ATTRIBUTE` clause in these statements only redefines the attributes temporarily. After the window closes (in the case of an OPEN WINDOW statement) or after the statement terminates (in the case of a CONSTRUCT, INPUT, DISPLAY, DIALOG,

INPUT ARRAY, or DISPLAY ARRAY statement), the runtime system restores the attributes from the most recent `OPTIONS` statement.

The `FORM` keyword in `INPUT ATTRIBUTE` or `DISPLAY ATTRIBUTE` clauses instructs the runtime system to use the input or display attributes of the current form. Similarly, you can use the `WINDOW` keyword of the same clauses to instruct the program to use the input or display attributes of the current window. You cannot combine the `FORM` or `WINDOW` attributes with any other attributes.

The following table shows the valid *input-attributes* and *display-attributes*:

Attribute	Description
`BLACK, BLUE, CYAN, GREEN, MAGENTA, RED, WHITE, YELLOW`	The color of the displayed text.
`BOLD, DIM, INVISIBLE, NORMAL`	The font attribute of the displayed text.
`REVERSE, BLINK, UNDERLINE`	The video attribute of the displayed text.

Defining the form input loop

The tab order in which the screen cursor visits fields of a form is that of the *field list* of currently executing CONSTRUCT, INPUT, and INPUT ARRAY statements, unless the tab order has been modified by a `NEXT FIELD` clause. By default, the interactive statement terminates if the user presses RETURN in the last field (or if the entered data fills the last field and that field has the AUTONEXT attribute).

The `INPUT WRAP` keywords change this behavior, causing the cursor to move from the last field to the first, repeating the sequence of fields until the user presses the Accept key. The `INPUT NO WRAP` option restores the default input loop behavior.

Application Termination

The `OPTIONS ON TERMINATE SIGNAL CALL` *`function`* defines the function that must be called when the application receives the SIGTERM signal. With this option, you can control program termination - for example, by using ROLLBACK WORK to cancel all pending SQL operations. If this statement is not called, the program is stopped with an exit value of SIGTERM (15).

On Microsoft Windows platforms, the function will be called in the following cases:

- The console window that the program was started from is closed.
- The current user session is terminated (i.e. the user logs off).
- The system is shut down.

Front-End Termination

The `OPTIONS ON CLOSE APPLICATION CALL` *`function`* can be used to execute specific code when the front-end stops. For example, when the client program is stopped, when the user session is ended, or when the workstation is shut down.

Before stopping, the front-end sends a internal event that is trapped by the runtime system. When a callback function is specified with the above program option command, the application code that was executing is canceled, and the callback function is executed before the program stops.

You typically do a ROLLBACK WORK, close all files, and release all resources in that function.

The default is `OPTIONS ON CLOSE APPLICATION STOP`. This instructs the runtime system to stop without any error message if the front end program is stopped.

Note that a front-end program crash or network failure is not detected and cannot be handled by this instruction.

Defining the message file

The `HELP FILE` clause specifies an expression that returns the filename of a help file. This filename can also include a pathname. Messages in this file can be referenced by number in form-related statements, and are displayed at runtime when the user presses the Help key.

By default, message files are searched in the current directory, then DBPATH environment variable is scanned to find the file.

See also Message Files.

Defining field tabbing order

In an INPUT, INPUT ARRAY or CONSTRUCT, by default, the tabbing order is defined by the list of fields used by the program instruction. This corresponds to `FIELD ORDER CONSTRAINED`.

When using `FIELD ORDER UNCONSTRAINED`, the UP ARROW and DOWN ARROW keys will move the cursor to the field above or below, respectively. Use the `FIELD ORDER CONSTRAINED` option to restore the default behavior of the UP ARROW and DOWN ARROW keys (moving the cursor to the previous or next field, respectively).

Warning: The `UNCONSTRAINED` option can only be supported in TUI mode, with a simple form layout. It is not recommended to use this option: it is supported for backward compatibility only.

When you specify `FIELD ORDER FORM`, the tabbing order is defined by the TABINDEX attributes of the current form fields. This allows you to define a tabbing order specific to the layout of the form, independent of the program instruction:

Form file:

```
01 LAYOUT
02 GRID
03 {
04   First name:  [f001           ]
05   Last name:   [f002           ]
06 }
07 END
08 END
09 ATTRIBUTES
10 EDIT f001 = FORMONLY.fname, TABINDEX = 1;
11 EDIT f002 = FORMONLY.lname, TABINDEX = 2;
12 END
```

Program file:

```
01 MAIN
02   DEFINE fname, lname CHAR(20)
03   OPTIONS FIELD ORDER UNCONSTRAINED
04   OPEN FORM f1 FROM "f1"
05   DISPLAY FORM f1
06   INPUT BY NAME fname, lname
07 END MAIN
```

Defining control keys TUI Only!

The `OPTIONS` instruction can specify physical keys to support logical key functions in the interactive instructions.

Warning: The physical key definition options are only provided for backward compatibility with the character mode. These as not supported in GUI mode. Use the Action Defaults to define accelerator keys for actions.
In TUI mode, action defaults accelerators are ignored.

Description of the keys:

- The `ACCEPT KEY` specifies the key that validates a CONSTRUCT, INPUT, DIALOG, INPUT ARRAY, or DISPLAY ARRAY statement.
  The default `ACCEPT KEY` is `ESCAPE`.
- The `DELETE KEY` specifies the key in INPUT ARRAY statements that deletes a screen record.
  The default `DELETE KEY` is `F2`.
- The `INSERT KEY` specifies the key that opens a screen record for data entry in INPUT ARRAY.
  The default `INSERT KEY` is `F1`.

- The NEXT KEY specifies the key that scrolls to the next page of a program array of records in an INPUT ARRAY or DISPLAY ARRAY statement.
  The default NEXT KEY is F3.
- The PREVIOUS KEY specifies the key that scrolls to the previous page of program records in an INPUT ARRAY or DISPLAY ARRAY statement.
  The default PREVIOUS KEY is F4.
- The HELP KEY specifies the key to display help messages.
  The default HELP KEY is CONTROL-W.

You can specify the following keywords for the physical key names:

Key Name	Description
ESC or ESCAPE	The ESC key (not recommended, use ACCEPT instead).
INTERRUPT	The interruption key (on UNIX, interruption signal).
TAB	The TAB key (not recommended).
CONTROL-*char*	A control key where *char* can be any character except A, D, H, I, J, K, L, M, R, or X
F1 through F255	A function key.
LEFT	The left arrow key.
RETURN or ENTER	The return key.
RIGHT	The right arrow key.
DOWN	The down arrow key.
UP	The up arrow key.
PREVIOUS or PREVPAGE	The previous page key.
NEXT or NEXTPAGE	The next page key.

You might not be able to use other keys that have special meaning to your version of the operating system. For example, CONTROL-C, CONTROL-Q, and CONTROL-S specify the Interrupt, XON, and XOFF signals on many UNIX systems.

Setting default screen modes

When using character terminals, BDL recognizes two screen display modes: line mode (IN LINE MODE) and formatted mode (IN FORM MODE). The OPTIONS and RUN statements can explicitly specify a screen mode. The OPTIONS statement can set separate defaults for these statements.

After IN LINE MODE is specified, the terminal is in the same state (in terms of stty options) as when the program began. This usually means that the terminal input is in cooked mode, with interruption enabled, and input not available until after a newline character has been typed.

The IN FORM MODE keywords specify raw mode, in which each character of input becomes available to the program as it is typed or read.

By default, a program operates in line mode, but so many statements take it into formatted mode (including `OPTIONS` statements that set keys, DISPLAY, OPEN WINDOW, DISPLAY FORM, and other screen interaction statements), that typical programs are actually in formatted mode most of the time.

When the `OPTIONS` statement specifies `RUN IN FORM MODE`, the program remains in formatted mode if it currently is in formatted mode, but it does not enter formatted mode if it is currently in line mode.

When the `OPTIONS` statement specifies `RUN IN LINE MODE`, the program remains in line mode if it is currently in line mode, and it switches to line mode if it is currently in formatted mode.

The RUN instruction

Purpose:

The `RUN` instruction creates a new process and executes the command passed as an argument.

Syntax:

```
RUN command
  [ IN {FORM|LINE} MODE ]
  [ RETURNING variable | WITHOUT WAITING ]
```

Notes:

1. *command* is a string expression containing the command to be executed.
2. *variable* is an integer variable receiving the execution status of the command.

Warnings:

1. The execution status in the `RETURNING` clause is system dependent. See below for more details.

Usage:

The `RUN` instruction executes an operating system command line; you can even run a second application as a secondary process. When the command terminates, the runtime system resumes execution.

Defining the command execution shell

In order to execute the command line, the `RUN` instruction uses the OS-specific shell defined in the environment of the current user. On UNIX, this is defined by the **SHELL** environment variable. On Windows, this is defined by **COMSPEC**. Note that on

Windows, the program defined by the **COMSPEC** variable must support the ***/c*** option as CMD.EXE.

Waiting for the sub-process

By default, the runtime system waits for the end of the execution of the command.

Unless you specify `WITHOUT WAITING`, the `RUN` instruction also does the following:

1. Causes execution of the current program to pause.
2. Displays any output from the specified command in a new window.
3. After that command completes execution, closes the new window and restores the previous display in the screen.

If you specify `WITHOUT WAITING`, the specified command line is executed as a background process, and generally does not affect the visual display. This clause is useful if you know that the command will take some time to execute, and your program does not need the result to continue. It is also used in GUI mode to start another Genero program. In TUI mode, you must not use this clause because two programs cannot run simultaneously on the same terminal.

Catching the execution status

The `RETURNING` clause saves the termination status code of the command that `RUN` executes in a program variable of type SMALLINT. You can then examine this variable in your program to determine the next action to take. A status code of zero usually indicates that the command has terminated normally. Non-zero exit status codes usually indicate that an error or a signal caused execution to terminate.

Warning: The execution status provided by the `RETURNING` clause is platform-dependent. On Unix systems, the value is composed of two bytes having different meanings. On Windows platforms, the execution status is usually zero for success, not zero if an error occurred.

On Unix systems, the lower byte (x mod 256) of the return status defines the termination status of the RUN command. The higher byte (x / 256) of the return status defines the execution status of the program. On Windows systems, the value of the return status defines the execution status of the program.

IN LINE MODE and IN FORM MODE

By default, programs operate in *LINE MODE*, but as many statements take it into *FORM MODE* (including OPTIONS statements that set keys, DISPLAY, OPEN WINDOW, DISPLAY FORM, and other screen interaction statements), typical programs are actually in *FORM MODE* most of the time.

According to the type of command to be executed, you may need to use the `IN {LINE|FORM} MODE` clause with the `RUN` instruction. It defines how the terminal or the graphical front-end behaves when running the child process.

Besides RUN, the OPTIONS, START REPORT, and REPORT statements can explicitly specify a screen mode. If no screen mode is specified in the RUN command, the current value from the OPTIONS statement is used. This is, by default, IN LINE MODE. The default screen mode for PIPE specifications in REPORT is IN FORM MODE.

When the RUN statement specifies IN FORM MODE, the program remains in *form mode* if it is currently in *form mode*, but it does not enter *form mode* if it is currently in *line mode*. When the prevailing RUN option specifies IN LINE MODE, the program remains in *line mode* if it is currently in *line mode*, and it switches to *line mode* if it is currently in *form mode*. This also applies to the PIPE option.

Typically, if you need to run another interactive program, you must use the IN LINE MODE clause:

- In a TUI mode, the terminal is in the same state (in terms if *tty* options) as when the program began. Usually the terminal input is in cooked mode, with interrupts enabled and input not becoming available until after a new-line character is typed.
- In a Graphical UI, if the WITHOUT WAITING clause in used, the front-end is warned before the child process is started (this causes a first network round-trip) After the child is started, the front-end is warned that the command was executed (second network round-trip). If the RUN command must wait for child termination (i.e. no WITHOUT WAITING clause is used), no particular action is taken.

However, if you want to execute a sub-process running silently (batch program without output), you must use the IN FORM MODE clause:

- In a TUI mode, the screen stays in *form mode* if it was in *form mode*, which saves a clear / redraw of the screen. The FORM mode specifies the terminal raw mode, in which each character of input becomes available to the program as it is typed or read.
- In a Graphical UI, no particular action is taken to warn the front-end (there is no need to warn the front-end for batch program execution).

Tip: To summarize, the FORM MODE must be used to optimize programs, if the child program does not do any output. If the child program uses interactive instructions, displays messages to the terminal, or if you don't known what it does, just use the RUN instruction in LINE MODE (which is the default).

It is recommended that you use functions to encapsulate child program and system command execution:

```
01 MAIN
02   DEFINE result SMALLINT
03   CALL runApplication("app2 -p xxx")
04   CALL runBatch("ls -l", FALSE) RETURNING result
05   CALL runBatch("ls -l > /tmp/files", TRUE) RETURNING result
06 END MAIN
07
08 FUNCTION runApplication(pname)
```

```
09    DEFINE pname, cmd STRING
10    LET cmd = "fglrun " || pname
11    IF fgl_getenv("FGLGUI") == 1 THEN
12       RUN cmd WITHOUT WAITING
13    ELSE
14       RUN cmd
15    END IF
16 END FUNCTION
17
18 FUNCTION runBatch(cmd, silent)
19    DEFINE cmd STRING
20    DEFINE silent STRING
21    DEFINE result SMALLINT
22    IF silent THEN
23       RUN cmd IN FORM MODE RETURNING result
24    ELSE
25       RUN cmd IN LINE MODE RETURNING result
26    END IF
27    IF fgl_getenv("OS") MATCHES "Win*" THEN
28       RETURN result
29    ELSE
30       RETURN ( result / 256 )
31    END IF
32 END FUNCTION
```

EXIT PROGRAM

Purpose:

The `EXIT PROGRAM` instruction terminates the execution of the program.

Syntax:

```
EXIT PROGRAM [ exit-code ]
```

Notes:

1. *exit-code* is a valid integer expression that can be read by the process which invoked the program.
2. Usually, *exit-code* will be zero by default for normal, errorless termination.

Warnings:

1. *exit-code* is converted into a positive integer between 0 and 255 (8 bits).

Example:

```
01 MAIN
02   DISPLAY "Emergency exit."
03   EXIT PROGRAM (-1)
```

```
04   DISPLAY "This will never be displayed !"
05 END MAIN
```

Database Schema Specification

Purpose:

Database Schema Specification identifies the database schema files to be used for compilation.

Syntax 1:

```
SCHEMA dbname
```

Syntax 2:

```
[DESCRIBE] DATABASE dbname
```

Notes:

1. The SCHEMA instruction defines the database schema files to be used for compilation.
2. *dbname* identifies the name of the database schema file to be used.
3. The database name must be expressed explicitly and not as a variable.
4. Use this instruction outside any program block, before a variable declaration with DEFINE LIKE instructions. It must precede any program block in each module that includes a DEFINE…LIKE declaration or INITIALIZE…LIKE and VALIDATE…LIKE statements. It must precede any GLOBALS…END GLOBALS block. It must also precede any DEFINE…LIKE declaration of module variables.
5. The [DESCRIBE] DATABASE instruction defines both the database schema files for compilation and the default database to connect to at runtime when the MAIN block is executed.

Warnings:

1. [DESCRIBE] DATABASE is supported for backward compatibility, but it is strongly recommended that you use SCHEMA instead. The SCHEMA instruction defines only the database schema for compilation, and not the default database to connect to at runtime, which can have a different name than the development database.

Example:

```
01 SCHEMA dbdevelopment -- Compilation database schema
02 DEFINE rec RECORD LIKE customer.*
03 MAIN
04    DATABASE dbproduction -- Runtime database specification
05    SELECT * INTO rec.* FROM customer WHERE custno=1
06 END MAIN
```

NULL Constant

Purpose:

The NULL constant is provided as "nil" value.

Syntax:

```
NULL
```

Notes:

1. When comparing variables to NULL, use the IS NULL operator, not the equal operator.
2. If an element of an expression is NULL, the expression is evaluated to NULL.

Warnings:

1. Variables are initialized to NULL or to zero according to their data type.
2. Empty character string literals ("") are equivalent to NULL.
3. NULL cannot be used with the = equal comparison operation, you must use IS NULL.

Example:

```
01 MAIN
02   DEFINE s CHAR(5)
03   LET s = NULL
04   DISPLAY "s IS NULL evaluates to:"
05   IF s IS NULL THEN
06      DISPLAY "TRUE"
07   ELSE
08      DISPLAY "FALSE"
09   END IF
10 END MAIN
```

TRUE Constant

Purpose:

The TRUE constant is a predefined boolean value that evaluates to 1.

Syntax:

```
TRUE
```

Example:

```
01 MAIN
02   IF FALSE = TRUE THEN
03      DISPLAY "Something wrong here"
04   END IF
05 END MAIN
```

FALSE Constant

Purpose:

The `FALSE` constant is a predefined boolean value that evaluates to 0.

Syntax:

```
FALSE
```

Example:

```
01 FUNCTION isodd( value )
02   DEFINE value INTEGER
03   IF value MOD 2 = 1 THEN
04      RETURN TRUE
05   ELSE
06      RETURN FALSE
07   END IF
08 END FUNCTION
```

NOTFOUND Constant

Purpose:

The `NOTFOUND` constant is a predefined integer value that evaluates to 100.

Syntax:

```
NOTFOUND
```

Notes:

1. This constant is used to test the execution status of an SQL statement returning a result set, to check whether rows have been found.

Example:

```
01 MAIN
02   DATABASE stores
03   SELECT tabid FROM systables WHERE tabid = 1
04   IF SQLCA.SQLCODE = NOTFOUND THEN
05      DISPLAY "No row was found"
06   END IF
07 END MAIN
```

BREAKPOINT

Purpose:

The BREAKPOINT instruction sets a program breakpoint when running in debug mode.

Syntax:

```
BREAKPOINT
```

Usage:

Normally, to set a breakpoint when you debug a program, you must use the break command of the debugger. But in some situations, you might need to set the breakpoint programmatically. Therefore, the BREAKPOINT instruction has been added to the language.

When you start fglrun in debug mode, if the program flow encounters a BREAKPOINT instruction, the program execution stops and the debug prompt is displayed, to let you enter a debugger command.

The BREAKPOINT instruction is ignored when not running in debug mode.

Example:

```
01 MAIN
02   DEFINE i INTEGER
03   LET i=123
04   BREAKPOINT
05   DISPLAY i
06 END MAIN
```

Setting Key Labels

Purpose:

This feature allows you to define the labels of keys, to show a specific text in the default action button created for the key.

Syntax 1: In FGLPROFILE

```
key.key-name.text = "label"
```

Syntax 2: At the program level

```
CALL FGL_SETKEYLABEL( "key-name", "label" )
```

Syntax 3: At the form level

```
KEYS
key-name = "label"
[...]
[END]
```

Syntax 4: At the dialog level

```
CALL FGL_DIALOG_SETKEYLABEL( "key-name", "label" )
```

Syntax 5: At the field level

```
KEY key-name = "label"
```

Notes:

1. *key-name* is the name of the key as defined below.

Warning:

1. This feature is provided for backward compatibility.

Usage:

Traditional 4GL applications use a lot of function keys and/or control keys to manage user actions. For example, in the following interactive dialog, the function key `F10` is used to show a detail window:

```
01 INPUT BY NAME myrecord.*
02    ON KEY (F10)
03       CALL ShowDetail()
04 END INPUT
```

For backward compatibility, the language allows you to specify a label to be displayed in a default action button created specifically for the key.

By default, if you do not specify a label, no action button is displayed for a function key or control key.

The following table shows the key names recognized by the runtime system:

Key Name	Description
`f1` to `f255`	Function keys.
`control-a` to `control-z`	Control keys.
`accept`	Validation key.
`interrupt`	Cancellation key.
`insert`	The insert key when in an INPUT ARRAY.
`delete`	The delete key when in an INPUT ARRAY.
`help`	The help key.

You can define key labels at different levels, from the default settings to a specific field, to show a specific label for the key when the focus is in that field. The order of precedence for key label definition is the following:

1. The label defined with the KEY attribute of the form field.
2. The label defined for the current dialog, using the FGL_DIALOG_SETKEYLABEL function.
3. The label defined in the KEYS section of the form specification file.
4. The label defined as default for a program, using the FGL_SETKEYLABEL function.
5. The label defined in the FGLPROFILE configuration file (`key.`*`key-name`*`.text` entries).

You can query the label defined at the program level with the FGL_GETKEYLABEL function and, for the current interactive instruction, with the FGL_DIALOG_GETKEYLABEL function.

Responding to CTRL_LOGOFF_EVENT

Purpose:

On Windows platforms, when the user disconnects, the system sends a CTRL_LOGOFF_EVENT event to all console applications. When the DVM receives this event, it stops immediately (a simple exit(0) system call is done).

On a Windows Terminal Server, if an Administrator user closes his session, a CTRL_LOGOFF_EVENT is sent to all console applications started by ANY user

connected to the machine (even if these applications were not started by the Administrator).

To prevent the DVM from stopping on a logoff event, you can use the `fglrun.ignoreLogoffEvent` entry in the FGLPROFILE configuration file. If this entry is set to `true`, the CTRL_LOGOFF_EVENT event is ignored by the DVM.

```
fglrun.ignoreLogoffEvent = true
```

As a result, when the Administrator user disconnects on a Windows Terminal Server, programs started by remote users would not stop.

Database Schema Files

Summary:

- What are Database Schema Files?
- Database Schema Extractor
- Schema Files
  - Column Definition File (.sch)
  - Column Validation File (.val)
  - Column Video Attributes File (.att)

See also: Forms, Programs, Variables, fgldbsch

Definition of Database Schema Files

Database Schema Files hold the definition of the database tables and columns. The schema files contain the column data types, validation rules, form item types, and display attributes for columns.

The schema files are typically used to centralize column data types to define program variables, as well as display attributes which are normally specified in the form specification file.

The database schema files are generated with the fgldbsch tool from the system tables of an existing database.

In program sources or form specification files, you must specify the database schema file with the SCHEMA instruction. The FGLDBPATH environment variable can be used to define a list of directories where the compiler can find database schema files.

Warning: The data types, display attributes, and validation rules are taken from the *Database Schema Files* during compilation. Make sure that the schema files of the development database correspond to the production database, otherwise the elements defined in the compiled version of your modules and forms will not match the table structures of the production database.

Program variables can be defined with the LIKE keyword to get the data type defined in the schema files:

```
01 SCHEMA stores
02 MAIN
03   DEFINE custrec RECORD LIKE customer.*
04   DEFINE name LIKE customer.cust_name
05   ...
06 END MAIN
```

Form fields defined with the FIELD item type can get the form item type from the schema files:

```
01 SCHEMA stores
02 LAYOUT
03 GRID
04 {
05   [f001         ]
06 }
07 TABLES
08   customer
09 END
10 ATTRIBUTES
11 FIELD f001 = customer.cust_name;
12 END
```

Note: For handling uppercase characters in the database name you must quote the name: `SCHEMA "myDatabase"`

Database Schema Extractor

See also: fgldbsch

The `fgldbsch` tool extracts the schema description for any database supported by the product. Schema information is extracted from the database specific system tables. The database type is automatically detected after connection; you do not have to specify any database server type.

```
fgldbsch -db test1 -un scott -up fourjs -v -ie
```

The database system must be available and the database client environment must be set properly in order to generate the schema files.

You must run `fgldbsch` with the `-db dbname` option to identify the database to which to connect. The `dbname` and related options could be present in the FGLPROFILE file. See Indirect database specification method in Database Connections. Otherwise, related options have to be provided with the `fgldbsch` command.

If the operating system user is not the database user, you can provide a database login and password with the `-un` and `-up` options.

The database driver can be specified with the `-dv dbdriver` option, if the default driver is not appropriate.

The BDL compiler expects FGL data types in the schema file. While most data types correspond to Informix SQL data types, non-Informix databases can have different data types. Therefore, data types are generated from the system catalog tables according to some conversion rules. You can control the conversion method with the `-cv` option.

Each character position of the string passed by this option corresponds to a line in the conversion table. You must give a conversion code for each data type (for example: `-cv AABAAAB`). Run the tool with the `-ct` option to see the conversion tables. When using `X` as conversion code, the columns using the corresponding data types will be ignored and not written to the .sch file. This is particularly useful in the case of auto-generated columns like SQL Server's *uniqueidentifier* data type, when using a `DEFAULT NEWID()` clause.

With some databases, the owner of tables is mandatory to extract a schema, otherwise you would get multiple definitions of the same table in the **.sch** schema file. To prevent such mistakes, you can specify the schema owner with the `-ow owner` option. If this option is not used, `fgldbsch` will use the login name passed with the `-un user` option.

By default `fgldbsch` does not generate system table definitions. Use the `-st` option to extract schema information of system tables.

Warning: The `fgldbsch` tool in BDL v1.3x provides the `-ns` option to generate without the database system tables. This option is no longer supported in the `fgldbsch` tool in BDL v2.xx and is replaced by the `-st` option to generate with the database system tables.

Use the `-tn tabname` option to extract schema information of a specific table. You may use the `-of name` option to generate files with a different name than the default name (the name of the database specified with the `-db` option).

By default, table and column names are converted to lower case letters to enforce compatibility with Informix. You can force lower case, upper case or case-sensitive generation by using the `-cl`, `-cu` or `-cc` options.

Warning: When using an Informix database, fgldbsch extracts synonyms. By default, only PUBLIC synonyms are extracted to avoid duplicates in the .sch file when the same name is used by several synonyms by different owners. If you want to extract PRIVATE synonyms, you must use the -ow option to specify the owner of the tables and synonyms.

Schema Files

Column Definition File (.sch)

The **.sch** file contains the data types of table columns.

Example:

```
01 customer^customer_num^258^4^1
02 customer^customer_name^256^50^2
```

```
03 customer^customer_address^0^100^3
04 order^order_num^258^4^1
05 order^order_custnum^258^4^2
06 order^order_date^263^4^3
07 order^order_total^261^1538^4
```

Description:

The data type of program variables or form fields used to hold data of a given database column must match the data type used in the database. BDL simplifies the definition of these elements by centralizing the information in external **.sch** files, which contain column data types.

In form files, you can directly specify the table and column name in the field definition in the ATTRIBUTES section of forms.

In programs, you can define variables with the data type of a database column by using the LIKE keyword.

Warnings:

1. As column data types are extracted from the database system tables, you may get different results with different database servers. For example, Informix provides the DATE data type to store simple dates in year, month, and day format (= BDL DATE), while Oracle stores DATEs as year to second (= BDL DATETIME YEAR TO SECOND).

The following table describes the fields you will find in a row of the **.sch** file:

Pos	Type	Description
1	STRING	Database table name.
2	STRING	Column name.
3	SMALLINT	Coded column data type. If the column is NOT NULL, you must add 256 to the value.
4	SMALLINT	Coded data type length.
5	SMALLINT	Ordinal position of the column in the table.
6	STRING	Default value of the database column. The value can be a simple numeric constant (1234.56) or a string delimited by single quotes ('abcdef').
7	STRING	Default form item type. The value can be one of the form item types (Edit, ButtonEdit, ComboBox, and so on).

Next table shows the data types that can be represented in the **.sch** schema file:

Data type name	Data type code	Data type length (field #4) This is a SMALLINT value encoding the length or composite length of the type.

	(field #3)	
CHAR	**0**	Maximum number of characters.
SMALLINT	**1**	Fixed length of 2
INTEGER	**2**	Fixed length of 4
FLOAT	**3**	Fixed length of 8
SMALLFLOAT	**4**	Fixed length of 4
DECIMAL	**5**	The length is computed using the following formula: **length = (precision * 256) + scale**
SERIAL	**6**	Fixed length of 4
DATE	**7**	Fixed length of 4
MONEY	**8**	Same as DECIMAL
Unused	**9**	
DATETIME	**10**	To code the qualifiers, the length is computed using the following formula: **length = (prec * 256) + (qual1 * 16) + qual2** where *prec* is the precision of the last qualifier and *qual1* / *qual2* identify qualifiers according to the following list: 0 = YEAR 2 = MONTH 4 = DAY 6 = HOUR 8 = MINUTE 10 = SECOND 11 = FRACTION(1) 12 = FRACTION(2) 13 = FRACTION(3) 14 = FRACTION(4) 15 = FRACTION(5)
BYTE	**11**	Length of descriptor
TEXT	**12**	Length of descriptor
VARCHAR	**13**	If length is positive: **length = (min_space * 256) + max_size** If length is negative: **length + 65536 = (min_space * 256) + max_size**
INTERVAL	**14**	Same as DATETIME
NCHAR	**15**	Same as CHAR
NVARCHAR	**16**	Same as VARCHAR
INT8	**17**	Fixed length of 8
SERIAL8	**18**	Fixed length of 8
SET (Unused)	**19**	
MULTISET (Unused)	**20**	
LIST (Unused)	**21**	
Unnamed ROW (Unused)	**22**	
Variable-length	**40**	

opaque type VARCHAR2 *Named ROW* *(Unused)*	**201** **4118**	Maximum number of characters.

Column Validation File (.val)

The **.val** file holds functional and display attributes of columns.

Example:

```
01 customer^customer_name^STYLE^"important"^
02 customer^customer_name^SHIFT^UP^
03 customer^customer_name^COMMENTS^"Name of the customer"^
04 order^order_date^DEFAULT^TODAY^
05 order^order_date^COMMENTS^"Creation date of the order"^
```

Description:

The **.val** file holds default attributes and validation rules for database columns.

In form files, the attributes are taken from the **.val** file as defaults if the corresponding attribute is not explicitly specified in the field definition of the ATTRIBUTES section.

In programs, you can validate variable values in accordance with the INCLUDE attribute by using the VALIDATE instruction.

The **.val** file can be generated by fgldbsch from the Informix-specific `syscolval` table, or can be edited by an external column attributes editor.

The following table describes the structure of the **.val** file:

Pos	Type	Description
1	STRING	Database table name.
2	STRING	Column name.
3	STRING	Column property name.
4	STRING	Column property value.

The supported attribute definitions are:

Attribute Name	Description
`ACTION`	Defines the ACTION attribute. Value must be an identifier.
`AUTONEXT`	Defines the AUTONEXT attribute. When this attribute is defined, value is `YES`.

AUTOSCALE	Defines the AUTOSCALE attribute. When this attribute is defined, value is YES.
CENTURY	Defines the CENTURY attribute. The value must be one of: R, C, F, or P.
COLOR	Defines the COLOR attribute. The value is a color identifier (RED, GREEN, BLUE, ...)
COMMENTS	Defines the COMMENTS attribute. The value is a quoted string or Localized String (%"xxx").
DEFAULT	Defines the DEFAULT attribute. Number, quoted string or identifier (TODAY).
FORMAT	Defines the FORMAT attribute. The value is a quoted string.
HEIGHT	Defines the HEIGHT attribute. The value is an integer followed by: { CHARACTERS, COLUMNS, LINES, POINTS, or PIXELS }
IMAGE	Defines the IMAGE attribute. The value is a quoted string.
INCLUDE	Defines an include list as the INCLUDE attribute. Value must be a list: (*value* [,...]), where *value* can be a number, quoted string or identifier (TODAY).
INITIALIZER	Defines the INITIALIZER attribute. Value must be an identifier.
INVISIBLE	Defines the INVISIBLE attribute. When this attribute is defined, value is YES.
ITEMS	Defines the VALUEUNCHECKED attribute. The value must be a list: (*item* [,...]), where *item* can be a number, a quoted string or (*value*,"label").
ITEMTYPE	Defines the Form Item Type to be used when the column is used as FIELD in forms. Value must be an identifier defining the item type (case sensitive!): Edit, ButtonEdit, Label, Image, DateEdit, TextEdit, ComboBox, RadioGroup, CheckBox, Slider, SpinEdit, TimeEdit, ProgressBar
JUSTIFY	Defines the JUSTIFY attribute. The value must be one of: LEFT, CENTER or RIGHT.
ORIENTATION	Defines the ORIENTATION attribute. The value must be one of: VERTICAL or HORIZONTAL.
PICTURE	Defines the PICTURE attribute. The value is a quoted string.
SAMPLE	Defines the SAMPLE attribute. The value is a quoted string.
SCROLL	Defines the SCROLL attribute. When this attribute is defined, value is YES.

SCROLLBARS	Defines the SCROLLBARS attribute. The value must be one of: X, Y or BOTH.
SHIFT	Corresponds to the UPSHIFT and DOWNSHIFT attributes. Values can be UP or DOWN.
SIZEPOLICY	Defines the SIZEPOLICY attribute. The value must be one of: INITIAL, DYNAMIC or FIXED.
STEP	Defines the STEP attribute. The value must be an integer.
STRECH	Defines the STRETCH attribute. The value must be one of: X, Y or BOTH.
STYLE	Defines the STYLE attribute. The value is a quoted string.
TAG	Defines the TAG attribute. The value is a quoted string.
TEXT	Defines the TEXT attribute. The value is a quoted string or Localized String (%"xxx").
TITLE	Defines the TITLE attribute. The value is a quoted string or Localized String (%"xxx").
VALUEMIN	Defines the VALUEMIN attribute. The value must be an integer.
VALUEMAX	Defines the VALUEMAX attribute. The value must be an integer.
VALUECHECKED	Defines the VALUECHECKED attribute. The value must be an number or a quoted string.
VALUEUNCHECKED	Defines the VALUEUNCHECKED attribute. The value must be an number or a quoted string.
VERIFY	Defines the VERIFY attribute. When this attribute is defined, value is YES.
WANTTABS	Defines the WANTTABS attribute. When this attribute is defined, value is YES.
WANTNORETURNS	Defines the WANTNORETURNS attribute. When this attribute is defined, value is YES.
WIDTH	Defines the WIDTH attribute. The value is an integer followed by: { CHARACTERS, COLUMNS, LINES, POINTS, or PIXELS }

Column Video Attributes File (.att)

The **.att** file contains the default video attributes of columns.

This file is generated by fgldbsch from the Informix-specific syscolatt table.

The following table describes the structure of the **.val** file:

Pos	Type	Description
1	STRING	Database table name.
2	STRING	Column name.
3	SMALLINT	Ordinal number of the attribute record.
4	STRING	COLOR attribute (coded).
5	CHAR(1)	INVERSE attribute (y/n).
6	CHAR(1)	UNDERLINE attribute (y/n).
7	CHAR(1)	BLINK attribute (y/n).
8	CHAR(1)	LEFT attribute (y/n).
9	STRING	FORMAT attribute.
10	STRING	Condition.

Warning: This feature is supported for compatibility with Informix 4GL only.

Globals

Summary:

- Definition
- Examples

See also: Variables, Arrays, Records, Constants, Programs

Definition

Purpose:

The GLOBALS instruction declares modular variables that can be exported to other program modules.

Syntax 1: Global block declaration

```
GLOBALS
  declaration-statement
  [,...]
END GLOBALS
```

Syntax 2: Importing global variables

```
GLOBALS "filename"
```

Notes:

1. In **Syntax 1**, *declaration-statement* is a variable or constant declaration.
2. In **Syntax 2**, *filename* is the name of a file containing the definition of global variables. Use this syntax to include a global declarations in the current module.

Warnings:

1. If you modify *filename*, you must recompile all the modules that include *filename.*
2. Do not declare a variable outside a GLOBALS…END GLOBALS block in a GLOBALS file.
3. Avoid confusing function names and global variables names
4. Avoid declaring the same global variable twice when including multiple GLOBALS files.

Usage:

In general, a program variable is in scope only in the same FUNCTION, MAIN, or REPORT program block in which it was declared.

To extend the visibility of one or more module variables beyond the source module in which they are declared, you must take the following steps:

1. Declare variables in `GLOBALS`…`END GLOBALS` declarations in files containing only `GLOBALS`, `DEFINE`, and `DATABASE` statements (but no executable statements).
2. Specify the files in `GLOBALS "filename"` statements in each additional source module that includes statements referencing the variables.

The *filename* must contain the **.4gl** suffix. It can be a a relative or an absolute path. To specify a path, the slash (/) directory separator can be used for Unix and Windows platforms.

If a local variable has the same name as another variable that you declare in the `GLOBALS` statement, only the local variable is visible within its scope of reference.

Each variable declared in a `GLOBALS ... END GLOBALS` block becomes a global variable.

You can declare several `GLOBALS` blocks in the same module.

A `GLOBALS` file must not contain any executable statement.

You do not compile the source file containing the `GLOBALS` block.

You can declare several `GLOBALS "filename"` in the same module.

Although you can include multiple `GLOBALS`…`END GLOBALS` statements in the same application, do not declare the same identifier as the name of a variable within the DEFINE statements of more than one `GLOBALS` declaration. Even if several declarations of a global variable defined in multiple places are identical, declaring any global variable more than once can result in compilation errors or unpredictable runtime behavior.

A `GLOBALS` block can hold `GLOBALS "filename"` instructions. In such case, the specified files will be included recursively.

Tips:

1. Use only a few global variables, too much globals makes the source code difficult to maintain and denies reusability.
2. There is no need to compile *filename,* but compiling *filename* might be useful to detect syntax errors.
3. To improve the readability of your source code, prefix global variables by "g_".
4. Global variables are often used as constants.
5. Global arrays allow a function to access the array modified by another function.

Examples

Example 1: Multiple GLOBALS file

labels.4gl : This module defines the text that should be displayed on the screen

```
01 GLOBALS
02   CONSTANT g_lbl_val = "Index:"
03   CONSTANT g_lbl_idx = "Value:"
04 END GLOBALS
```

globals.4gl : Declares a global array and a constant containing its size

```
01 GLOBALS "labels.4gl" -- this statement could be line 2 of main.4gl
02 GLOBALS
03   DEFINE g_idx ARRAY[100] OF CHAR(10)
04   CONSTANT g_idxsize = 100
05 END GLOBALS
```

database.4gl : This module could be dedicated to database access

```
01 GLOBALS "globals.4gl"
02 FUNCTION get_id()
03   DEFINE li INTEGER
04   FOR li = 1 TO g_idxsize -- this could be a FOREACH statement
05       LET g_idx[li] = g_idxsize - li
06   END FOR
07 END FUNCTION
```

main.4gl : Fill in the global array and display the result

```
01 GLOBALS "globals.4gl"
02 MAIN
03   DISPLAY "Initializing constant values for this application..."
05   DISPLAY "Filling the data from function get_idx in module
database.4gl..."
06   CALL get_id()
07   DISPLAY "Retrieving a few values from g_idx"
08   CALL display_data()
09 END MAIN
10 FUNCTION display_data()
11   DEFINE li INTEGER
12   LET li = 1
13   WHILE li <= 10 AND li <= g_idxsize
14       DISPLAY g_lbl_idx CLIPPED || li || " " || g_lbl_val CLIPPED ||
g_idx[li]
15       LET li = li + 1
16   END WHILE
17 END FUNCTION
```

Flow Control

Summary:

- Invoking a function (CALL)
- Returning from a function (RETURN)
- Conditional cases (CASE)
- Continuing a block (CONTINUE *instruction*)
- Leaving a block (EXIT *instruction*)
- Iterative loop (FOR)
- Labeled transfer (GOTO)
- Conditional block (IF)
- Statement label (LABEL)
- Suspending execution (SLEEP)
- Conditional loop (WHILE)

See also: Programs, Functions, Reports, Expressions

CALL

Purpose:

The CALL instruction invokes a specified function.

Syntax:

```
CALL function ( [ parameter [,...] ] ) [ RETURNING variable [,...] ]
```

Notes:

1. *function* is the name of a built-in function or the name of the function defined in one of the modules of the program.
2. *parameter* can be a variable, a literal, a constant or any valid expression.
3. *parameters* are separated by a comma ' , '.
4. The RETURNING clause assigns values returned by the function to variables in the calling routine.
5. *variable* is a variable receiving a value returned by the function.
6. The RETURNING clause is only needed when the function returns parameters.
7. A function returning a single parameter can be used in expressions.

Tips:

1. You can use a double-pipe operator ' || ' to pass the concatenation of character expressions as a parameter.

Warnings:

1. The value of a receiving *variable* may be different from the value returned by the function, following the data conversion rules.

Example 1: Function returning a single value

```
01 MAIN
02   DEFINE var1 CHAR(10)
03   DEFINE var2 CHAR(2)
04   LET var1 = foo()
05   DISPLAY "var1 = " || var1
06   CALL foo() RETURNING var2
07   DISPLAY "var2 = " || var2
08 END MAIN
09
10 FUNCTION foo()
11   RETURN "Hello"
12 END FUNCTION
```

Example 2: Function returning several values

```
01 MAIN
02   DEFINE var1 CHAR(15)
03   DEFINE var2 CHAR(15)
04   CALL foo() RETURNING var1, var2
05   DISPLAY var1, var2
06 END MAIN
07
08 FUNCTION foo()
09   DEFINE r1 CHAR(15)
10   DEFINE r2 CHAR(15)
11   LET r1 = "return value 1"
12   LET r2 = "return value 2"
13   RETURN r1, r2
14 END FUNCTION
```

Example 3: Function and records

```
01 MAIN
02   DEFINE r1 RECORD
03               id1     INTEGER,
04               id2     INTEGER,
05               name    CHAR(30)
06             END RECORD
07   CALL get_name( NULL, NULL, NULL ) RETURNING r1.*
08   CALL get_name( NULL, r1.id2, r1.name ) RETURNING r1.*
09   CALL get_name( r1.* ) RETURNING r1.*
10   DISPLAY r1.name
11   CALL get_name( 1, 2, "John" ) RETURNING r1.id2, r1.id1, r1.name
12   DISPLAY r1.name
13 END MAIN
14
15 FUNCTION get_name( code1, code2, name )
```

```
16   DEFINE code1   INTEGER
17   DEFINE code2   INTEGER
18   DEFINE name    CHAR(30)
19   IF code1 IS NULL THEN
20      LET name = "ERROR:code1 is NULL"
21      LET code2 = NULL
22   ELSE
23      IF code2 IS NULL THEN
24        LET name = "ERROR:code2 is NULL"
25        LET code1 = NULL
26      ELSE
27         IF name IS NULL THEN
28            LET name = "SMITH"
29         END IF
30      END IF
31   END IF
32   RETURN code1, code2, name
33 END FUNCTION
```

RETURN

Purpose:

The `RETURN` instruction transfers the control back from a function with optional return values.

Syntax:

```
RETURN [ value [,...] ]
```

Notes:

1. *value* can be a variable, a literal, a constant or any valid expression.
2. Record members can be returned with the `.*` or `THRU` notation. Each member is returned as an independent variable.
3. A function may have several `RETURN` points (not recommended in structured programming) but they must all return the same number of values.
4. The number of returned *values* must correspond to the number of variables listed in the `RETURNING` clause of the `CALL` statement invoking this function.

Warnings:

1. A function cannot return an array.

Example:

```
01 MAIN
02   DEFINE forname, surname CHAR(10)
03   CALL foo(NULL) RETURNING forname, surname
```

```
04   DISPLAY forname CLIPPED, " ", upshift(surname) CLIPPED
05   CALL foo(1) RETURNING forname, surname
06   DISPLAY forname CLIPPED, " ", upshift(surname) CLIPPED
07 END MAIN
08
09 FUNCTION foo(code)
10   DEFINE code   INTEGER
11   DEFINE person RECORD
12                  name1   CHAR(10),
13                  name2   CHAR(20)
14                END RECORD
15  IF code IS NULL THEN
16     RETURN NULL, NULL
17  ELSE
18     LET person.name1 = "John"
19     LET person.name2 = "Smith"
20     RETURN person.*
21  END IF
22 END FUNCTION
```

CASE

Purpose:

The CASE instruction specifies statement blocks that must be executed conditionally.

Syntax 1:

```
CASE expression-1
  WHEN expression-2
    { statement | EXIT CASE }
    [...]
  [ OTHERWISE
    { statement | EXIT CASE }
    [...]
  ]
END CASE
```

Syntax 2:

```
CASE
  WHEN boolean-expression
    { statement | EXIT CASE }
    [...]
  [ OTHERWISE
    { statement | EXIT CASE }
    [...]
  ]
END CASE
```

Notes:

1. *expression-1* is any expression supported by the language.
2. *expression-2* is an expression that is tested against *expression-1. expression-1* and *expression-2* should have the same data type.
3. *boolean-expression* is any boolean expression supported by the language.
4. *statement* is any instruction supported by the language.
5. In a `CASE` flow control block, the first matching `WHEN` block is executed. If there is no matching `WHEN` block, then the `OTHERWISE` block is executed.
6. If there is no matching `WHEN` block and no `OTHERWISE` block, then the program control jumps to the statement following the `END CASE` keyword.
7. The `EXIT CASE` statement transfers the program control to the statement following the `END CASE` keyword.
8. There is an implicit `EXIT CASE` statement at the end of each `WHEN` block and at the end of the `OTHERWISE` block.

Warnings:

1. A `NULL` expression is considered as `FALSE`: When doing a `CASE` *`expr`* `... WHEN [NOT] NULL` using the syntax 1, it always evaluates to `FALSE`. Use syntax 2 as `CASE ... WHEN` *`expr`* `IS NULL` to test if an expression is null.
2. Make sure that *expression-2* is not a boolean expression when using the first syntax. The compiler will not raise an error in this case, but you might get unexpected results at runtime.
3. If there is more than one *expression-2* matching *expression-1* (syntax 1), or if two boolean expressions (syntax 2) are true, only the first matching `WHEN` block will be executed.
4. The `OTHERWISE` block must be the last block of the `CASE` instruction.

Example 1: First syntax

```
01 MAIN
02    DEFINE v CHAR(10)
03    LET v = "C1"
04    CASE v
05      WHEN "C1"
06        DISPLAY "Value is C1"
07      WHEN "C2"
08        DISPLAY "Value is C2"
09      WHEN "C3"
10        DISPLAY "Value is C3"
11      OTHERWISE
12        DISPLAY "Unexpected value"
13    END CASE
14 END MAIN
```

Example 2: Second syntax

```
01 MAIN
02    DEFINE v CHAR(10)
03    LET v = "C1"
```

```
04    CASE
05      WHEN ( v="C1" OR v="C2" )
06        DISPLAY "Value is either C1 or C2"
06      WHEN ( v="C3" OR v="C4" )
07        DISPLAY "Value is either C3 or C4"
08      OTHERWISE
09        DISPLAY "Unexpected value"
10    END CASE
11 END MAIN
```

CONTINUE

Purpose:

The `CONTINUE` instruction transfers the program execution from a statement block to another location in the compound statement that is currently being executed.

Syntax:

```
CONTINUE { FOR | FOREACH | MENU | CONSTRUCT | INPUT | WHILE }
```

Notes:

1. `CONTINUE` *`instruction`* can only be used within the statement block specified by *`instruction`*. For example, `CONTINUE FOR` can only be used within a `FOR ... END FOR` statement block.
2. The `CONTINUE FOR`, `CONTINUE FOREACH`, or `CONTINUE WHILE` keywords cause the current `FOR`, `FOREACH`, or `WHILE` loop (respectively) to begin a new cycle immediately. If conditions do not permit a new cycle, however, the looping statement terminates.
3. The `CONTINUE CONSTRUCT` and `CONTINUE INPUT` statements cause the program to skip all subsequent statements in the current control block. The screen cursor returns to the most recently occupied field in the current form, giving the user another chance to enter data in that field.
4. The `CONTINUE MENU` statement causes the program to ignore the remaining statements in the current `MENU` control block and redisplay the menu. The user can then choose another menu option.

Tips:

1. `CONTINUE INPUT` is valid in `INPUT` and `INPUT ARRAY` statements.

Example:

```
01 MAIN
02   DEFINE i INTEGER
03   LET i = 0
04   WHILE i < 5
```

```
05     LET i = i + 1
06     DISPLAY "i=" || i
07     CONTINUE WHILE
08     DISPLAY "This will never be displayed !"
09   END WHILE
10 END MAIN
```

FOR

Purpose:

The FOR instruction executes a statement block a specified number of times.

Syntax:

```
FOR counter = start TO finish [ STEP value ]
   statement
   [...]
END FOR
```

Notes:

1. *counter* is a variable of type INTEGER or SMALLINT that serves as an index for the FOR statement block.
2. *start* is an integer expression used to set an initial counter value.
3. *finish* is any valid integer expression used to specify an upper limit for *counter*.
4. *value* is any valid integer expression whose value is added to *counter* after each iteration of the statement block.
5. When the STEP keyword is not given, *counter* is incremented by 1.
6. *statement* is any instruction supported by the language.
7. If *value* is less than 0, *counter* is decreased. In this case, *start* should be higher than *finish*.

Usage:

The FOR instruction block executes the statements up to the END FOR keyword a specified number of times, or until EXIT FOR terminates the FOR statement.

The runtime system maintains the counter, whose value changes on each pass through the statement block. On the first iteration through the loop, this counter is set to the initial expression at the left of the TO keyword. For all further iterations, the value of the increment expression in the STEP clause specification (1 by default) is added to the counter in each pass through the block of statements. When the sign of the difference between the values of counter and the finish expression at the right of the TO keyword changes, the runtime system exits from the FOR loop.

The FOR loop terminates after the iteration for which the left- and right-hand expressions are equal. Execution resumes at the statement following the END FOR keywords. If either

expression returns NULL, the loop cannot terminate, because the Boolean expression "left = right" cannot become TRUE.

Tips:

1. If the FOR loop includes one or more SQL statements that modify the database, then it is advisable that the entire FOR loop be within a transaction. You may also PREPARE the SQL statements before the loop to increase performance.

Warnings:

1. *counter* MUST be of type INTEGER or SMALLINT.
2. *value* = 0 causes an unending loop unless there is an adequate EXIT FOR statement.
3. NULL for *start*, *finish* or *value* is treated as 0. There is no way to catch this as an error.
4. If *statement* modifies the value of *counter*, you might get unexpected results at runtime. In this case, it is recommended that you use a WHILE loop instead.
5. It is highly recommended that you ensure that *statement* does not modify the values of *start*, *finish* and/or *value*.

Example:

```
01 MAIN
02   DEFINE i, i_min, i_max INTEGER
03   LET i_min = 1
04   LET i_max = 10
05   DISPLAY "Look how well I can count from " || i_min || " to " ||
i_max
06   DISPLAY "I can count forwards..."
07   FOR i = i_min TO i_max
08       DISPLAY i
09   END FOR
10   DISPLAY "... and backwards!"
11   FOR i = i_max TO i_min STEP -1
12       DISPLAY i
13   END FOR
14 END MAIN
```

GOTO

Purpose:

The GOTO instruction transfers program control to a labeled line within the same program block.

Syntax:

```
GOTO [ : ] label-id
```

Notes:

1. *label-id* is the name of the LABEL statement to jump to.
2. The label can be defined before or after the GOTO statement.

Tips:

1. GOTO statements can reduce the readability of your program source and result in infinite loops. It is recommended that you use FOR, WHILE, IF, CASE, CALL statements instead.
2. The GOTO statement can be used in a WHENEVER statement to handle exceptions.

Warnings:

1. The LABEL and GOTO statements must use the *label-id* within a single MAIN, FUNCTION, or REPORT program block.

Example:

```
01 MAIN
02   DEFINE exit_code INTEGER
03   DEFINE l_status  INTEGER
04   WHENEVER ANY ERROR GOTO _error
05   DISPLAY 1/0
06   GOTO _noerror
07   LABEL _error:
08     LET l_status = STATUS
09     DISPLAY "The error number ", l_status, " has occurred."
10     DISPLAY "Description : ", err_get(l_status)
11     LET exit_code = -1
12   GOTO :_exit
13   LABEL _noerror:
14     LET exit_code = 0
15   GOTO _exit
16   LABEL _exit:
17   EXIT PROGRAM (exit_code)
18 END MAIN
```

EXIT

Purpose:

The EXIT instruction transfers control out of a control structure (a block, a loop, a CASE statement, or an interface instruction).

Syntax:

```
EXIT { CASE | FOR | MENU | CONSTRUCT | FOREACH | REPORT | DISPLAY |
INPUT | WHILE }
```

Notes:

1. The `EXIT` *`instruction`* instruction must be used inside the control structure specified by *`instruction`*. For example, `EXIT FOR` can only appear inside a `FOR ... END FOR` program structure.
2. `EXIT DISPLAY` exits the `DISPLAY ARRAY` instruction and `EXIT INPUT` exits both `INPUT` and `INPUT ARRAY` blocks.

Tips:

1. To exit a function, use the `RETURN` instruction.
2. To exit a program, use the `EXIT PROGRAM` instruction.

Example:

```
01 MAIN
02   DEFINE i INTEGER
03   LET i = 0
04   WHILE TRUE
05     DISPLAY "This is an infinite loop. How would you get out of here
?"
06     LET i = i + 1
07     IF i = 100 THEN
08       EXIT WHILE
09     END IF
10   END WHILE
11   DISPLAY "Well done."
12 END MAIN
```

IF

Purpose:

The `IF` instruction executes a group of statements conditionally.

Syntax:

```
IF condition THEN
   statement
   [...]
[ ELSE
   statement
   [...]
```

```
]
END IF
```

Notes:

1. *condition* is any boolean expression supported by the language.
2. *statement* is any instruction supported by the language.

Usage:

If *condition* is `TRUE`, the runtime system executes the block of statements following the `THEN` keyword, until it reaches either the `ELSE` keyword or the `END IF` keywords and resumes execution after the `END IF` keywords.

If *condition* is `FALSE`, the runtime system executes the block of statements between the `ELSE` keyword and the `END IF` keywords. If `ELSE` is absent, it resumes execution after the `END IF` keywords.

Tips:

1. To test the equality of integer expressions, both " = " and " == " operators may be used. `IF 5 = 5 THEN ...` can be written `IF 5 == 5 THEN ...`

Warnings:

1. A `NULL` expression is considered as `FALSE`. Use the `IS NULL` keyword to test if an expression is null.

Example:

```
01 MAIN
02   DEFINE name CHAR(20)
03   LET name = "John Smith"
04   IF name MATCHES "John*" THEN
05      DISPLAY "The first name is too common to be displayed."
06      IF name MATCHES "*Smith" THEN
07         DISPLAY "Even the last name is too common to be displayed."
08      END IF
09   ELSE
10      DISPLAY "The name is " || name || "."
11   END IF
12 END MAIN
```

LABEL

Purpose:

The `LABEL` instruction declares a statement label, making the next statement one to which a GOTO statement can transfer program control.

Syntax:

```
LABEL label-id:
```

Notes:

1. *label-id* is a unique identifier in a MAIN, REPORT, or FUNCTION program block.
2. The *label-id* must be followed by a colon (`:`).

Example:

```
01 MAIN
02   DISPLAY "Line 2"
03   GOTO line5
04   DISPLAY "Line 4"
05   LABEL line5:
06   DISPLAY "Line 6"
07 END MAIN
```

SLEEP

Purpose:

The `SLEEP` instruction causes the program to pause for the specified number of seconds.

Syntax:

```
SLEEP seconds
```

Notes:

1. *seconds* is a valid integer expression.

Warnings:

1. If *seconds* < 0 or *second* `IS NULL`, the program does not stop.

Example:

```
01 MAIN
```

```
02   DISPLAY "Please wait 5 seconds..."
03   SLEEP 5
04   DISPLAY "Thank you."
05 END MAIN
```

WHILE

Purpose:

The `WHILE` statement executes a block of statements while a condition that you specify in a boolean expression is true.

Syntax:

```
WHILE b-expression
   statement
   [...]
END WHILE
```

Notes:

1. *b-expression* is any valid boolean expression.
2. *statement* is any instruction supported by the language.

Usage:

If *b-expression* is `TRUE`, the runtime system executes the statements that follow it, down to the `END WHILE` keyword. The runtime system again evaluates the *b-expression*, and if it is still `TRUE`, the runtime system executes the same statement block. The runtime system usually stops when *b-expression* becomes `FALSE` or *statement* is `EXIT WHILE`.

If *b-expression* is `FALSE`, the runtime system passes control to the statement that follows `END WHILE`.

Tips:

1. If *b-expression* is complex, it is much better to define a boolean [INTEGER or CHAR(1)] variable that takes the result of *b-expression* and use this variable for *b-expression*.
2. A `WHILE` loop can replace a `FOR` loop : `FOR i = 1 TO 5 ; ... ; END FOR` is equivalent to `LET i = 1 ; WHILE i <= 5 ; ... ; LET i = i + 1 ; END WHILE`
3. In order to avoid unending loops, make sure that either *statement* will cause *b-expression* to be `FALSE`, or that the `EXIT WHILE` statement will be executed.

Example:

```
01 MAIN
02   DEFINE lval INTEGER
03   DEFINE lmin INTEGER
04   DEFINE lmax INTEGER
05   DEFINE lnb  INTEGER
06   DEFINE lcnt INTEGER
07   DEFINE lguess INTEGER
08   DISPLAY "NumberGuess program"
09   LET lnb  = 20
10   LET lmin = 0
11   LET lmax = 1000
12   LET lval = 753 --random value between lmin and lmax
13   DISPLAY "Guess a number between " || lmin || " and " || lmax
14   LET lguess = (lmax - lmin) / 2
15   LET lcnt = 1
16   WHILE lguess <> lval AND lcnt < lnb
17     DISPLAY "\n  Attempt number " || lcnt
18     DISPLAY "  Your guess is " || lguess || ", hopefully between "
|| lmin || " and " || lmax
19     IF lval > lguess THEN
20        DISPLAY "  Try higher."
21        LET lmin = lguess
22     ELSE
23        DISPLAY "  Try lower."
24        LET lmax = lguess
25     END IF
26     LET lguess = lmin + (lmax - lmin) / 2
27     LET lcnt = lcnt + 1
28   END WHILE
29   IF lcnt >= lnb THEN
30      DISPLAY "Sorry, the maximum number of attempts has been
reached. The number was " || lval
31   ELSE
32      DISPLAY "Well done. You have found the number " || lval || " in
" || lcnt || " attempts."
33   END IF
34 END MAIN
```

Functions

Summary:

- Definition
- Usage
- Examples

See also: Variables, Data Types, Flow Control

Definition

Purpose:

The `FUNCTION` statement defines a named program block containing a set of statements to be executed when the function is invoked.

Syntax:

```
FUNCTION function-name ( [ argument [,...] ] )
  [ define-statement | constant-statement ]
  { fgl-statement | sql-statement | return-statement } [...]
END FUNCTION
```

where *return-statement* is:

```
RETURN expression [,...]
```

Notes:

1. *function-name* is the identifier that you declare for this function and must be unique among all the names of functions or reports in the same program.
2. *argument* is the name of a formal argument to this function. Its scope of reference is local to the function.
3. *define-statement* is used to define function arguments and local variables.
4. *constant-statement* can be used to declare local constants.
5. *fgl-statement* is any instruction supported by the language.
6. *sql-statement* is any static SQL instruction supported by the language.
7. *expression* is any expression supported by the language.

Usage

The FUNCTION block both declares and defines a function. The function declaration specifies the identifier of the function and the identifiers of its formal arguments (if any).

A FUNCTION block cannot appear within the MAIN block, in a REPORT block, or within another FUNCTION block.

The data type of each formal argument of the function must be specified by a DEFINE statement that immediately follows the argument list. The actual argument in a call to the function need not be of the declared data type of the formal argument. If data type conversion is not possible, a runtime error occurs.

Function arguments are passed by value (i.e. value is copied on the stack) for basic data types and records, while dynamic arrays and objects are passed by reference (i.e. a handle to the original data is copied on the stack and thus allows modification of the original data inside the function).

Local variables are not visible in other program blocks. The identifiers of local variables must be unique among the variables that are declared in the same FUNCTION definition. Any global or module variable that has the same identifier as a local variable, however, is not visible within the scope of the local variable.

A function that returns one or more values to the calling routine must include the *return-statement*. Values specified in `RETURN` must correspond in number and position, and must be of the same or of compatible data types, to the variables in the `RETURNING` clause of the CALL statement. If the function returns a single value, it can be invoked as an operand within a expression. Otherwise, you must invoke it with the CALL statement with a `RETURNING` clause. An error results if the list of returned values in the `RETURN` statement conflicts in number or in data type with the `RETURNING` clause of the CALL statement that invokes the function.

Any GOTO or WHENEVER ERROR GOTO statement in a function must reference a statement label within the same `FUNCTION` block.

A function can invoke itself recursively with a CALL statement.

Warnings:

1. If no argument is specified, an empty argument list must still be supplied, enclosed between the parentheses.
2. If the name is also the name of a built-in function, an error occurs at link time, even if the program does not reference the built-in function.

Examples

Example 1:

```
01 FUNCTION findCustomerNumber(name)
02   DEFINE name CHAR(50)
03   DEFINE num INTEGER
```

```
04   CONSTANT sqltxt = "SELECT cust_num FROM customer WHERE cust_name =
?"
05   PREPARE stmt FROM sqltxt
06   EXECUTE stmt INTO num USING name
07   IF SQLCA.SQLCODE = 100 THEN
08      LET num =-1
09   END IF
10   RETURN num
11 END FUNCTION
```

Reports

Summary:

- What are reports?
- Report Engine Configuration
- Report Driver Instructions
- Report Routine Structure
- Statements in Report Routine
- Report Routine Prototype
- Two-Pass Reports
- Report Instructions
    - EXIT REPORT
    - PRINT
    - PRINTX
    - NEED
    - PAUSE
    - SKIP
- Report Operators
    - COLUMN
    - LINENO
    - PAGENO
    - SPACES
    - WORDWRAP
    - USING
    - ASCII
- Report Aggregate Functions
    - COUNT(*)
    - PERCENT(*)
    - SUM()
    - AVG()
    - MIN()
    - MAX()

See also: Programs, Variables, Result set

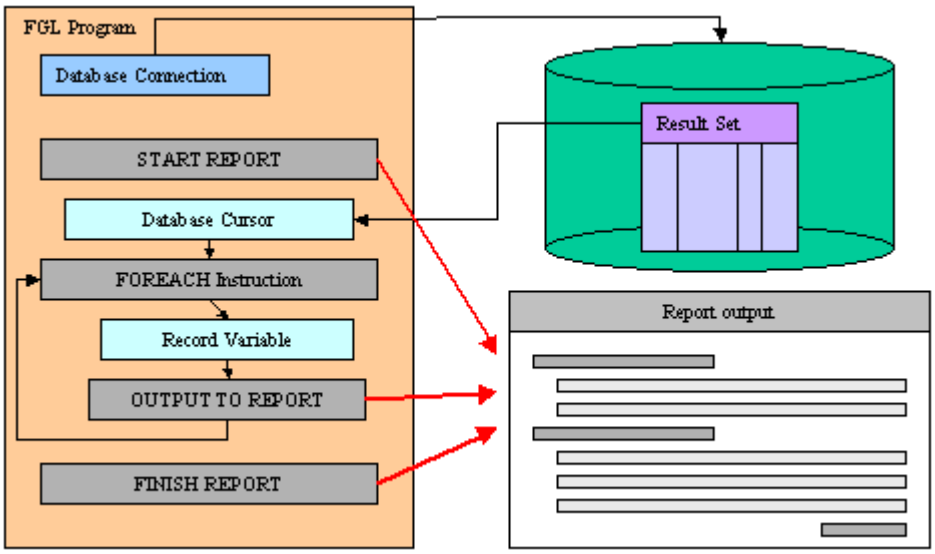
Definition

A report can arrange and format the data according to your instructions and display the output on the screen, send it to a printer, or store it as a file for future use.

To implement a report, a program must include two distinct components:

- The Report Driver specifies what data the report includes.
- The Report Routine formats the data for output.

The Report Driver retrieves the specified rows from a database, stores their values in program variables, and sends these - one input record at a time - to the Report Routine. After the last input record is received and formatted, the runtime system calculates any aggregate values based on all the data and sends the entire report to some output device.



By separating the two tasks of data retrieval and data formatting, the runtime system simplifies the production of recurrent reports and makes it easy to apply the same report format to different data sets.

The report engine supports the following features:

- The option to display report output to the screen for editing.
- Full control over page layout for your report, including first page header and generic page headers, page trailers, columnar presentation, and special formatting before groups and after groups sorted by value.
- Facilities for creating the report either from the rows returned by a cursor or from input records assembled from any other source, such as output from several different `SELECT` statements through the Report Driver.
- Control blocks to manipulate data from a database cursor on a row-by-row basis, either before or after the row is formatted by the report.
- Aggregate functions that can calculate and display frequencies, percentages, sums, averages, minimum, and maximum values.
- The `USING` operator and other built-in functions and operators for formatting and displaying information in output from the report.
- The `WORDWRAP` operator to format long character strings that occupy multiple lines of output from the report.
- The option to update the database or execute any sequence of SQL and other statements while writing a report, if the intermediate values calculated by the report meet specified criteria; for example, to write an alert message containing a second report.

- Stopping a report in the report definition code, with EXIT REPORT or TERMINATE REPORT.

The report engine supports one-pass reports and two-pass reports. The one-pass requires sorted data to be produced by the report driver in order to handle before/after groups properly. The two-pass record handles sort internally and does not need sorted data from the report driver. During the first pass, the report engine sorts the data and stores the sorted values in a temporary file in the database. During the second pass, it calculates any aggregate values and produces output from data in the temporary files.

Report Engine Configuration

By default, GROUP aggregate functions such as SUM() return a NULL value if all items values are NULL. You can force the report engine to return a zero decimal value with the following FGLPROFILE setting:

```
Report.aggregateZero = {true|false}
```

When this entry is set to true, aggregate functions return zero when all values are NULL.

Default value is : false (Aggregate functions evaluate to NULL if all items are NULL)

The Report Driver

The Report Driver invokes the report, retrieves data, and sends the data (as input records) to be formatted by the `REPORT` program block (or routine). A Report Driver can be part of the `MAIN` program block, or it can be in one or more functions. The report driver typically consists of a loop (such as WHILE, FOR, or FOREACH) with the following statements to process the report:

Instruction	Description
START REPORT	This statement is required to instantiate the report driver.
OUTPUT TO REPORT	Provide data for one row to the report driver.
FINISH REPORT	Normal termination of the report.
TERMINATE REPORT	Cancels the processing of the report.

Usage:

A report driver is started by the START REPORT instruction. Once started, data can be given to the report driver through the OUTPUT TO REPORT statement. To instruct the report engine to terminate output processing, use the FINISH REPORT instruction.

It is possible to manage several report drivers at the same time. It is even possible to invoke a report driver inside a REPORT program block, which is different from the current driver.

The programmer must make sure that the runtime system will always execute these instructions in the following order:

1. START REPORT
2. OUTPUT TO REPORT
3. FINISH REPORT

Example:

```
01 DATABASE stores7
02 MAIN
03   DEFINE rcust RECORD LIKE customer.*
04   DECLARE cu1 CURSOR FOR SELECT * FROM customer
05   START REPORT myrep
06   FOREACH cu1 INTO rcust.*
07      OUTPUT TO REPORT myrep(rcust.*)
08   END FOREACH
09   FINISH REPORT myrep
10 END MAIN
```

START REPORT

Syntax:

```
START REPORT report-name
  [
   TO
   {
      SCREEN
    | PRINTER
    | FILE filename
    | PIPE program { IN FORM MODE | IN LINE MODE }
    | OUTPUT
      {
         "SCREEN"
       | "PRINTER"
       | "FILE" DESTINATION filename
       | "PIPE { IN FORM MODE | IN LINE MODE }" DESTINATION program
       | variable [ DESTINATION { program | filename } ]
      }
```

```
    }
  ]
  [
   WITH
   {
     [ LEFT MARGIN = m-left [,] ]
     [ RIGHT MARGIN  = m-right [,] ]
     [ TOP MARGIN = m-top [,] ]
     [ BOTTOM MARGIN = m-bottom [,] ]
     [ PAGE LENGTH = m-length [,] ]
     [ TOP OF PAGE = c-top  [,] ]
   }
  ]
]
```

Notes:

1. The START REPORT statement supersedes any clause in the output section of the report definition.
2. *report-name* is a report that has been defined as a REPORT routine.
3. *filename* is a string expression specifying the file that receives output.
4. *program* is a string expression specifying a program, a shell script, or a command line to receive output.
5. *variable* is a variable of type STRING that specifies one of: SCREEN, PRINTER, FILE, PIPE, PIPE IN LINE MODE, PIPE IN FORM MODE. If PRINTER is specified, the DBPRINT environment variable specifies which printer.
6. The values corresponding to a margin and page length must be valid integer expressions.
7. The margins can be defined in any order, but a comma "," is required to separate two page dimensions statements.
8. The comma "," cannot appear before the first or after the last page dimensions statements.
9. *m-left* is the left margin in number of characters.
10. *m-right* is the right margin in number of characters.
11. *m-top* is the top margin in number of lines.
12. *m-bottom* is the bottom margin in number of lines.
13. *c-top* is a string that defines the page-eject character sequence.

Tips:

1. The START REPORT statement is handy to dynamically set up the destination and / or page setup of a report.

Warnings:

1. If a START REPORT statement references a report that is already running, the report is reinitialized; any output might be unpredictable.

OUTPUT TO REPORT

Syntax:

```
OUTPUT TO REPORT report-name ( parameters )
```

Notes:

1. *report-name* is the name of the report to which the *parameters* should be sent.
2. *parameters* is the data that needs to be sent to the report. As in a function call, *parameters* must match the DEFINE section of the report routine.

Warnings:

1. At compile time, the number of parameters is not checked against the DEFINE section of the report routine. This is a known behavior of the language.

FINISH REPORT

Syntax:

```
FINISH REPORT report-name
```

Notes:

1. *report-name* is the name of the report to be ended.
2. `FINISH REPORT` must be the last statement in the report driver.

Usage:

`FINISH REPORT` closes the report driver. Therefore, it must be the last statement in the report driver and must follow a `START REPORT` statement that specifies the name of the same report.

`FINISH REPORT` does the following:

1. Completes the second pass, if report is a two-pass report. These 'second pass' activities handle the calculation and output of any aggregate values that are based on all the input records in the report, such as `COUNT(*)` or `PERCENT(*)` with no `GROUP` qualifier.
2. Executes any `AFTER GROUP OF` control blocks.
3. Executes any `PAGE HEADER`, `ON LAST ROW`, and `PAGE TRAILER` control blocks to complete the report.
4. Copies data from the output buffers of the report to the destination.
5. Closes the Select cursor on any temporary table that was created to order the input records or to perform aggregate calculations.

TERMINATE REPORT

Syntax:

```
TERMINATE REPORT report-name
```

Notes:

1. *report-name* is the name of the report to be canceled.

Usage:

`TERMINATE REPORT` cancels the report processing. It is typically used when the program (or the user) becomes aware that a problem prevents the report from producing part of its intended output, or when the user interrupted the report processing.

`TERMINATE REPORT` has the following effects:

- Terminates the processing of the current report.
- Deletes any intermediate files or temporary tables that were created in processing the report.

The EXIT REPORT instruction has the same effect, except that it can be used inside the report definition.

Report Definition

Syntax:

```
REPORT report-name (argument-list)
    [ define-section  ]
    [ output-section  ]
    [ sort-section ]
    [ format-section  ]
END REPORT
```

where *define-section* is a function parameter definition using the DEFINE instruction. You usually define one or more record variables:

```
DEFINE variable RECORD
           member data-type
           [,...]
       END RECORD
```

where *output-section* is:

```
OUTPUT
[
  REPORT TO
  {
      SCREEN
    | PRINTER
    | [ FILE ] filename
    | PIPE [ IN FORM MODE | IN LINE MODE ] program
  }
]
[
  [ WITH ]
  [ LEFT MARGIN m-left ]
  [ RIGHT MARGIN m-right ]
  [ TOP MARGIN m-top ]
  [ BOTTOM MARGIN m-bottom ]
  [ PAGE LENGTH m-length ]
  [ TOP OF PAGE c-top ]
]
```

where *sort-section* is:

```
ORDER [ EXTERNAL ] BY variable-list
```

where *format-section* is:

```
FORMAT EVERY ROW
```

or:

```
FORMAT
{
 [ FIRST ] PAGE HEADER
 |  ON EVERY ROW
 | BEFORE GROUP OF variable
 | AFTER GROUP OF variable
 | PAGE TRAILER
 |  ON LAST ROW
}
      [ fgl-statement | sql-statement | report-statement ]
      [...]
   [...]
```

Notes:

1. The *define-section* declares the data types of local variables used within the report, and of any variables (the input records) that are passed as arguments to the report by the calling statement. Reports without arguments or local variables do not require a `DEFINE` section.
2. The *output-section* can set margin and page size values, and can also specify where to send the formatted output. Output from the report consists of successive pages, each containing a fixed number of lines whose margins and maximum number of characters are fixed.

3. The *sort-section* specifies how the rows have to be sorted. The specified sort order determines the order in which the runtime system processes any `GROUP OF` control blocks in the `FORMAT` section.
4. The *format-section* is required. It specifies the appearance of the report, including page headers, page trailers, and aggregate functions of the data. It can also contain control blocks that specify actions to take before or after specific groups of rows are processed. (Alternatively, it can produce a default report by only specifying `FORMAT EVERY ROW`).

Usage:

The report definition formats input records. Like the `FUNCTION` or `MAIN` statement, it is a program block that can be the scope of local variables. It is not, however, a function; it is not reentrant, and `CALL` cannot invoke it. The report definition receives data from its driver in sets called input records. These records can include program records, but other data types are also supported. Each input record is formatted and printed as specified by control blocks and statements within the report definition. Most statements and functions can be included in a report definition, and certain specialized statements and operators for formatting output can appear only in a report definition.

Like `MAIN` or `FUNCTION`, the report definition must appear outside any other program block. It must begin with the `REPORT` statement and must end with the `END REPORT` keywords.

Some statements are prohibited in a `REPORT` program control block.

The DEFINE Section

Syntax:

See the DEFINE statement.

Usage:

This section declares a data type for each formal argument in the `REPORT` prototype and for any additional local variables that can be referenced only within the `REPORT` program block. The `DEFINE` section is required if you pass arguments to the report or if you reference local variables in the report.

For declaring local variables, the same rules apply to the `DEFINE` section as to the DEFINE statement in `MAIN` and `FUNCTION` program blocks. Two exceptions, however, restrict the data types of formal arguments:

- Report arguments cannot be of type `ARRAY`.
- Report arguments cannot be records that include `ARRAY` members.

Data types of local variables that are not formal arguments are unrestricted. You must include arguments in the report prototype and declare them in the DEFINE section, if any of the following conditions is true:

- If you specify FORMAT EVERY ROW to create a default report, you must pass all the values for each record of the report.
- If an ORDER BY section is included, you must pass all the values that ORDER BY references for each input record of the report.
- If you use the AFTER GROUP OF control block, you must pass at least the arguments that are named in that control block.
- If an aggregate that depends on all records of the report appears anywhere except in the ON LAST ROW control block, you must pass each of the records of the report through the argument list.

Aggregates dependent on all records include:

- GROUP PERCENT(*) (anywhere in a report).
- Any aggregate without the GROUP keyword (anywhere outside the ON LAST ROW control block).

If your report calls an aggregate function, an error might result if any argument of an aggregate function is not also a format argument of the report. You can, however, use global or module variables as arguments of aggregates if the value of the variable does not change while the report is executing.

A report can reference variables of global or module scope that are not declared in the DEFINE section. Their values can be printed, but they can cause problems in aggregates and in BEFORE GROUP OF and AFTER GROUP OF clauses. Any references to non-local variables can produce unexpected results, however, if their values change while a two-pass report is executing.

The OUTPUT Section

Syntax:

```
OUTPUT
[
  REPORT TO
  {
      SCREEN
    | PRINTER
    | [ FILE ] filename
    | PIPE [ IN FORM MODE | IN LINE MODE ] program
  }
]
[
  [ WITH ]
  [ LEFT MARGIN m-left ]
```

```
  [ RIGHT MARGIN m-right ]
  [ TOP MARGIN m-top ]
  [ BOTTOM MARGIN m-bottom ]
  [ PAGE LENGTH m-length ]
  [ TOP OF PAGE c-top ]
]
```

Notes:

1. This section is superseded by any corresponding `START REPORT` specifications. Any output destination or page setup definition may be overridden by the report driver with the `START REPORT` instruction.
2. *program* is a string literal, global, or constant specifying the name of a program, shell script, command receiving the output.
3. *filename* is a string literal, global, or constant specifying the file which receives the output of the report.
4. *m-left* is the left margin in number of characters.
5. *m-right* is the right margin in number of characters.
6. *m-top* is the top margin in number of lines.
7. *m-bottom* is an integer the bottom margin in number of lines.
8. *c-top* is a string that defines the page-eject character sequence.

Usage:

The `OUTPUT` section can specify the destination and dimensions for output from the report and the page-eject sequence for the printer. If you omit the `OUTPUT` section, the report uses default values to format each page. This section is superseded by any corresponding `START REPORT` specifications.

The `OUTPUT` section can direct the output from the report to a printer, file, or pipe, and can initialize the page dimensions and margins of report output. If `PRINTER` is specified, the `DBPRINT` environment variable specifies which printer.

The `START REPORT` statement of the report driver can override all of these specifications by assigning another destination in its `TO` clause or by assigning other dimensions, margins, or another page-eject sequence in the `WITH` clause.

Because the size specifications for the dimensions and margins of a page of report output that the `OUTPUT` section can specify must be literal integers, you might prefer to reset these values in the `START REPORT` statement, where you can use variables to assign these values dynamically at runtime.

The ORDER BY Section

Purpose:

This section specifies how the variables of the input records are to be sorted. It is required if the report driver does not send sorted data to the report. The specified sort order determines the order in which the runtime system processes any `GROUP OF` control blocks in the FORMAT section.

Syntax:

```
ORDER [ EXTERNAL ] BY variable [ DESC | ASC ] [,...]
```

Notes:

1. The `EXTERNAL` keyword specifies that the data is sent to the report in a sorted order. Without the `EXTERNAL` keyword, the report driver sorts the data before sending it to the report program block.
2. *variable* identifies one of the variables passed to the report routine to be used for sorting rows. The variables must be separated by a comma.
3. The `DESC` or `ASC` options defines the sort order

Usage:

The `ORDER BY` section specifies a sort list for the input records. Include this section if values that the report definition receives from the report driver are significant in determining how `BEFORE GROUP OF` or `AFTER GROUP OF` control blocks will process the data in the formatted report output.

If you omit the `ORDER BY` section, the runtime system processes input records in the order received from the report driver and processes any `GROUP OF` control blocks in their order of appearance in the `FORMAT` section. If records are not sorted in the report driver, the `GROUP OF` control blocks might be executed at random intervals (that is, after any input record) because unsorted values tend to change from record to record.

If you specify only one variable in the `GROUP OF` control blocks, and the input records are already sorted in sequence on that variable by the `SELECT` statement, you do not need to include an `ORDER BY` section in the report.

Specify `ORDER EXTERNAL BY` if the input records have already been sorted by the `SELECT` statement. The list of variables after the keywords `ORDER EXTERNAL BY` control the execution order of `GROUP BY` control blocks.

Without the `EXTERNAL` keyword, the report is a *two-pass* report, meaning that the report engine processes the set of input records twice. During the first pass, the report engine sorts the data and stores the sorted values in a temporary file in the database. During the second pass, it calculates any aggregate values and produces output from data in the temporary files.

With the EXTERNAL keyword, the report engine only needs to make a single pass through the data: it does not need to build the temporary table in the database for sorting the data. Specifying EXTERNAL to instruct the report engine not to sort the records again might result in an improvement in performance.

The FORMAT Section

Purpose:

A report definition must contain a FORMAT section. The FORMAT section determines how the output from the report will look. It works with the values that are passed to the REPORT program block through the argument list or with global or module variables in each record of the report. In a source file, the FORMAT section begins with the FORMAT keyword and ends with the END REPORT keywords.

Syntax:

Default format:

```
FORMAT EVERY ROW
```

Custom format:

```
FORMAT
   control-block
      [ fgl-statement | sql-statement | report-statement ]
      [...]
    [...]
```

where *control-block* can be one of:

```
{
[ FIRST ] PAGE HEADER
|  ON EVERY ROW
|  BEFORE GROUP OF variable
|  AFTER GROUP OF variable
|  PAGE TRAILER
|  ON LAST ROW
}
```

Notes:

1. *fgl-statement* is any language instruction supported in the report routine.
2. *sql-statement* is any SQL statement supported by the language.
3. *report-statement* is any report-specific instruction.

Usage:

The `FORMAT` section is made up of the following *Control Blocks*:

- FIRST PAGE HEADER
- PAGE HEADER
- PAGE TRAILER
- BEFORE GROUP OF
- AFTER GROUP OF
- ON EVERY ROW
- ON LAST ROW

If you use the `FORMAT EVERY ROW`, no other statements or control blocks are valid. The `EVERY ROW` keywords specify a default output format, including every input record that is passed to the report.

Control blocks define the structure of a report by specifying one or more statements to be executed when specific parts of the report are processed.

If a report driver includes `START REPORT` and `FINISH REPORT` statements, but no data records are passed to the report, no control blocks are executed. That is, unless the report executes an `OUTPUT TO REPORT` statement that passes at least one input record to the report; then neither the `FIRST PAGE HEADER` control block nor any other control block is executed

Apart from `BEFORE GROUP OF` and `AFTER GROUP OF`, each control block must appear only one time.

More complex `FORMAT` sections can contain control blocks like `ON EVERY ROW` or `BEFORE GROUP OF`, which contain statements to execute while the report is being processed. Control blocks can contain report execution statements and other executable statements.

See also statements and report format section.

A control block may invoke most *fgl-statements* and *sql-statements*, except those listed in prohibited statements.

The `BEFORE/AFTER GROUP OF` control blocks can include aggregate functions to instruct the report engine to automatically compute such values.

A *report-statement* is a statement specially designed for the report format section. It cannot be used in any other part of the program.

The sequence in which the `BEFORE GROUP OF` and `AFTER GROUP OF` control blocks are executed depends on the sort list in the `ORDER BY` section, regardless of the physical sequence in which these control blocks appear within the `FORMAT` section.

FORMAT EVERY ROW

A report routine written with FORMAT EVERY ROW formats the report in a simple default format, containing only the values that are passed to the REPORT program block through its arguments, and the names of the arguments. You cannot modify the EVERY ROW statement with any of the statements listed in report execution statements, and neither can you include any control blocks in the FORMAT section.

The report engine uses as column headings the names of the variables that the report driver passes as arguments at runtime. If all fields of each input record can fit horizontally on a single line, the default report prints the names across the top of each page and the values beneath. Otherwise, it formats the report with the names down the left side of the page and the values to the right, as in the previous example. When a variable contains a null value, the default report prints only the name of the variable, with nothing for the value.

The following example is a brief report specification that uses FORMAT EVERY ROW. We assume here that the cursor that retrieved the input records for this report was declared with an ORDER BY clause, so that no ORDER BY section is needed in this report definition:

```
01 DATABASE stores7
02
03 REPORT simple( order_num, customer_num, order_date )
04
05   DEFINE order_num LIKE orders.order_num,
06          customer_num LIKE orders.customer_num,
07          order_date LIKE orders.order_date
08
09   FORMAT EVERY ROW
10
11 END REPORT
```

The above example would produce the following output:

```
order_num customer_num order_date
     1001         104  01/20/1993
     1002         101  06/01/1993
     1003         104  10/12/1993
     1004         106  04/12/1993
     1005         116  12/04/1993
     1006         112  09/19/1993
     1007         117  03/25/1993
     1008         110  11/17/1993
     1009         111  02/14/1993
     1010         115  05/29/1993
     1011         104  03/23/1993
     1012         117  06/05/1993
```

FIRST PAGE HEADER

This control block specifies the action that the runtime system takes before it begins processing the first input record. You can use it, for example, to specify what appears near the top of the first page of output from the report.

Because the runtime system executes the `FIRST PAGE HEADER` control block before generating any output, you can use this control block to initialize variables that you use in the `FORMAT` section.

If a report driver includes `START REPORT` and `FINISH REPORT` statements, but no data records are passed to the report, this control block is not executed. That is, unless the report executes an `OUTPUT TO REPORT` statement that passes at least one input record to the report, neither the `FIRST PAGE HEADER` control block nor any other control block is executed.

As its name implies, you can also use a `FIRST PAGE HEADER` control block to produce a title page as well as column headings. On the first page of a report, this control block overrides any `PAGE HEADER` control block. That is, if both a `FIRST PAGE HEADER` and a `PAGE HEADER` control block exist, output from the first appears at the beginning of the first page, and output from the second begins all subsequent pages.

The `TOP MARGIN` (set in the `OUTPUT` section) determines how close the header appears to the top of the page.

Warnings:

1. You cannot include a `SKIP` integer `LINES` statement inside a loop within this control block.
2. The `NEED` statement is not valid within this control block.
3. If you use an `IF…THEN…ELSE` statement within this control block, the number of lines displayed by any `PRINT` statements following the `THEN` keyword must be equal to the number of lines displayed by any `PRINT` statements following the `ELSE` keyword.
4. If you use a `CASE`, `FOR`, or `WHILE` statement that contains a `PRINT` statement within this control block, you must terminate the `PRINT` statement with a semicolon (;). The semicolon suppresses any `LINEFEED` characters in the loop, keeping the number of lines in the header constant from page to page.
5. You cannot use a `PRINT` filename statement to read and display text from a file within this control block

Corresponding restrictions also apply to `CASE`, `FOR`, `IF`, `NEED`, `SKIP`, `PRINT`, and `WHILE` statements in `PAGE HEADER` and `PAGE TRAILER` control blocks.

PAGE HEADER

This control block is executed whenever a new page is added to the report. The PAGE HEADER control block specifies the action that the runtime takes before it begins processing each page of the report. It can specify what information, if any, appears at the top of each new page of output from the report.

The TOP MARGIN specification (in the OUTPUT section) affects how many blank lines appear above the output produced by statements in the PAGE HEADER control block.

You can use the PAGENO operator in a PRINT statement within a PAGE HEADER control block to automatically display the current page number at the top of every page.

The FIRST PAGE HEADER control block overrides this control block on the first page of a report.

New group values can appear in the PAGE HEADER control block when this control block is executed after a simultaneous end-of-group and end-of-page situation.

The runtime system delays the processing of the PAGE HEADER control block until it encounters the first PRINT, SKIP, or NEED statement in the ON EVERY ROW, BEFORE GROUP OF, or AFTER GROUP OF control block. This order guarantees that any group columns printed in the PAGE HEADER control block have the same values as the columns printed in the ON EVERY ROW control block.

Warnings:

1. Warnings that apply to FIRST PAGE HEADER also apply to PAGE HEADER.

PAGE TRAILER

The PAGE TRAILER control block specifies what information, if any, appears at the bottom of each page of output from the report.

The runtime system executes the statements in the PAGE TRAILER control block before the PAGE HEADER control block when a new page is needed. New pages can be initiated by any of the following conditions:

- PRINT attempts to print on a page that is already full.
- SKIP TO TOP OF PAGE is executed.
- SKIP n LINES specifies more lines than are available on the current page.
- NEED specifies more lines than are available on the current page.

You can use the PAGENO operator in a PRINT statement within a PAGE TRAILER control block to automatically display the page number at the bottom of every page, as in the following example:

```
01 PAGE TRAILER
02    PRINT COLUMN 28, PAGENO USING "page <<<<"
```

The BOTTOM MARGIN specification (in the OUTPUT section) affects how close to the bottom of the page the output displays the page trailer.

Warnings:

1. Warnings that apply to FIRST PAGE HEADER also apply to PAGE TRAILER.

BEFORE/AFTER GROUP OF

The BEFORE/AFTER GROUP OF control blocks specify what action the runtime system takes respectively before or after it processes a group of input records. Group hierarchy is determined by the ORDER BY specification in the SELECT statement or in the report definition.

A group of records is all of the input records that contain the same value for the variable whose name follows the AFTER GROUP OF keywords. This group variable must be passed through the report arguments. A report can include no more than one AFTER GROUP OF control block for any group variable.

When the runtime system executes the statements in a BEFORE/AFTER GROUP OF control block, the report variables have the values from the first / last record of the new group. From this perspective, the BEFORE/AFTER GROUP OF control block could be thought of as the "on first / last record of group" control block.

Each BEFORE GROUP OF block is executed in order, from highest to lowest priority, at the start of a report (after any FIRST PAGE HEADER or PAGE HEADER control blocks, but before processing the first record) and on these occasions:

- Whenever the value of the group variable changes (after any AFTER GROUP OF block for the old value completes execution)
- Whenever the value of a higher-priority variable in the sort list changes (after any AFTER GROUP OF block for the old value completes execution)

The runtime system executes the AFTER GROUP OF control block on these occasions:

- Whenever the value of the group variable changes.
- Whenever the value of a higher-priority variable in the sort list changes.
- At the end of the report (after processing the last input record but before the runtime system executes any ON LAST ROW or PAGE TRAILER control blocks). In this case, each AFTER GROUP OF control block is executed in ascending priority.

How often the value of the group variable changes depends in part on whether the input records have been sorted by the SELECT statement:

- If records are already sorted, the BEFORE/AFTER GROUP OF block executes before the runtime system processes the first record of the group.
- If records are not sorted, the BEFORE GROUP OF block might be executed after any record because the value of the group variable can change with each record. If no ORDER BY section is specified, all BEFORE/AFTER GROUP OF control blocks are executed in the same order in which they appear in the FORMAT section. The BEFORE/AFTER GROUP OF control blocks are designed to work with sorted data.

You can sort the records by specifying a sort list in either of the following areas:

- An ORDER BY section in the report definition
- The ORDER BY clause of the SELECT statement in the report driver

To sort data in the report definition (with an ORDER BY section), make sure that the name of the group variable appears in both the ORDER BY section and in the BEFORE GROUP OF control block.

To sort data in the ORDER BY clause of a SELECT statement, perform the following tasks:

- Use the column name in the ORDER BY clause of the SELECT statement as the group variable in the BEFORE GROUP OF control block.
- If the report contains BEFORE or AFTER GROUP OF control blocks, make sure that you include an ORDER EXTERNAL BY section in the report to specify the precedence of variables in the sort list.

If you specify sort lists in both the report driver and the report definition, the sort list in the ORDER BY section of the REPORT takes precedence.
When the runtime system starts to generate a report, it first executes the BEFORE GROUP OF control blocks in descending order of priority before it executes the ON EVERY ROW control block. If the report is not already at the top of the page, the SKIP TO TOP OF PAGE statement in a BEFORE GROUP OF control block causes the output for each group to start at the top of a page.

If the sort list includes more than one variable, the runtime system sorts the records by values in the first variable (highest priority). Records that have the same value for the first variable are then ordered by the second variable and so on until records that have the same values for all other variables are ordered by the last variable (lowest priority) in the sort list.

The ORDER BY section determines the order in which the runtime system processes BEFORE GROUP OF and AFTER GROUP OF control blocks. If you omit the ORDER BY section, the runtime system processes any GROUP OF control blocks in the lexical order of their appearance within the FORMAT section.

If you include an ORDER BY section, and the FORMAT section contains more than one BEFORE GROUP OF or AFTER GROUP OF control block, the order in which these control blocks are executed is determined by the sort list in the ORDER BY section. In this case,

their order within the FORMAT section is not significant because the sort list overrides their lexical order.

The runtime system processes all the statements in a BEFORE GROUP OF or AFTER GROUP OF control block on these occasions:

- Each time the value of the current group variable changes.
- Each time the value of a higher-priority variable changes. How often the value of the group variable changes depends in part on whether the input records have been sorted. If the records are sorted, AFTER GROUP OF executes after the runtime system processes the last record of the group of records; BEFORE GROUP OF executes before the runtime system processes the first records with the same value for the group variable. If the records are not sorted, the BEFORE GROUP OF and AFTER GROUP OF control blocks might be executed before and after each record because the value of the group variable might change with each record. All the AFTER GROUP OF and BEFORE GROUP OF control blocks are executed in the same lexical order in which they appear in the FORMAT section.

In the AFTER GROUP OF control block, you can include the GROUP keyword to qualify aggregate report functions like AVG(), SUM(), MIN(), or MAX():

```
01  AFTER GROUP OF r.order_num
02    PRINT r.order_date, 7 SPACES,
03          r.order_num USING"###&", 8 SPACES,
04          r.ship_date, " ",
05          GROUP SUM(r.total_price) USING"$$$$,$$$,$$$.&&"
06  AFTER GROUP OF r.customer_num
07    PRINT 42 SPACES, "--------------------"
08    PRINT 42 SPACES, GROUP SUM(r.total_price) USING"$$$$,$$$,$$$.&&"
```

Using the GROUP keyword to qualify an aggregate function is only valid within the AFTER GROUP OF control block. It is not valid, for example, in the BEFORE GROUP OF control block.

After the last input record is processed, the runtime system executes the AFTER GROUP OF control blocks before it executes the ON LAST ROW control block.

ON EVERY ROW

The ON EVERY ROW control block specifies the action to be taken by the runtime system for every input record that is passed to the report definition.

The runtime system executes the statements within the ON EVERY ROW control block for each new input record that is passed to the report. The following example is from a report that lists all the customers, their addresses, and their telephone numbers across the page:

```
01  ON EVERY ROW
02    PRINT r.fname, " ", r.lname, " ",
03          r.address1, " ", r.cust_phone
```

The runtime system delays processing the `PAGE HEADER` control block (or the `FIRST PAGE HEADER` control block, if it exists) until it encounters the first `PRINT`, `SKIP`, or `NEED` statement in the `ON EVERY ROW` control block.

If a `BEFORE GROUP OF` control block is triggered by a change in the value of a variable, the runtime system executes all appropriate `BEFORE GROUP OF` control blocks (in the order of their priority) before it executes the `ON EVERY ROW` control block. Similarly, if execution of an `AFTER GROUP OF` control block is triggered by a change in the value of a variable, the runtime system executes all appropriate `AFTER GROUP OF` control blocks (in the reverse order of their priority) before it executes the `ON EVERY ROW` control block.

ON LAST ROW

The `ON LAST ROW` control block specifies the action that the runtime system is to take after it processes the last input record that was passed to the report definition and encounters the `FINISH REPORT` statement.

The statements in the `ON LAST ROW` control block are executed after the statements in the `ON EVERY ROW` and `AFTER GROUP OF` control blocks if these blocks are present.

When the runtime system processes the statements in an `ON LAST ROW` control block, the variables that the report is processing still have the values from the final record that the report processed. The `ON LAST ROW` control block can use aggregate functions to display report totals.

Statements in Report Definition Routine

Prohibited Statements

Language statements that have no meaning inside a report definition routine are prohibited. The following table shows some of the statements that are not valid within any control block of the `FORMAT` section of a `REPORT` program block:

`CONSTRUCT`	`FUNCTION`	`MENU`
`DEFER`	`INPUT`	`PROMPT`
`DEFINE`	`INPUT ARRAY`	`REPORT`
`DISPLAY ARRAY`	`MAIN`	`RETURN`

A compile-time error is issued if you attempt to include any of these statements in a control block of a report. You can call a function that includes some of these statements, but this is not recommended.

Report Control Statements

The following statements can appear only in control blocks of the `FORMAT` section of a report definition:

Statement	Effect
`EXIT REPORT`	Cancels processing of the report from within the report definition.
`NEED`	Forces a page break unless some specified number of lines is available on the current page of the report.
`PAUSE`	Allows the user to control scrolling of screen output (This statement has no effect if output is sent to any destination except the screen.)
`PRINT`	Appends a specified item to the output of the report.
`SKIP`	Inserts blank lines into a report or forces a page break.

The Report Prototype

When defining a report routine, the report name must immediately follow the `REPORT` keyword. The name must be unique among function and report names within the program. Its scope is the entire program.

The list of formal arguments of the report must be enclosed in parentheses and separated by commas. These are local variables that store values that the calling routine passes to the report. The compiler issues an error unless you declare their data types in the subsequent DEFINE section. You can include a program record in the formal argument list, but you cannot append the `.*` symbols to the name of the record. Arguments can be of any data type except ARRAY, or a record with an ARRAY member.

When you call a report, the formal arguments are assigned values from the argument list of the OUTPUT TO REPORT statement. These actual arguments that you pass must match, in number and position, the formal arguments of the `REPORT` routine. The data types must be compatible, but they need not be identical. The runtime system can perform some conversions between compatible data types.

The names of the actual arguments and the formal arguments do not have to match.

You must include the following items in the list of formal arguments:

- All the values for each row sent to the report in the following cases:
  - If you include an ORDER BY section or `GROUP PERCENT(*)` function

    - If you use a global aggregate function (one over all rows of the report) anywhere in the report, except in the ON LAST ROW control block
    - If you specify the FORMAT EVERY ROW default format
- Any variables referenced in the following group control blocks:
    - AFTER GROUP OF
    - BEFORE GROUP OF

Two-Pass Reports

The report engine supports one-pass reports and two-pass reports. The one-pass report requires sorted data to be produced by the report driver in order to handle before/after groups properly. The two-pass report handles sorts internally and does not need sorted data from the report driver. During the first pass, the report engine sorts the data and stores the sorted values in a temporary file in the database. During the second pass, it calculates any aggregate values and produces output from data in the temporary files.

A report is defined as a two-pass report if it includes any of the following items:

- An ORDER BY section without the `EXTERNAL` keyword.
- The `GROUP PERCENT(*)` aggregate function anywhere in the report.
- Any aggregate function that has no `GROUP` keyword in any control block other than ON LAST ROW.

Two-pass reports create temporary tables. The FINISH REPORT statement uses values from these tables to calculate any global aggregates, and then deletes the tables.

Warnings:

1. A two-pass report is one that creates a temporary table. Therefore, the report engine requires that the program be connected to a database when the report runs. Make sure that the database server and the database driver supports temporary table creation and indexes creation on temporary tables.

EXIT REPORT

Syntax:

```
EXIT REPORT
```

Usage:

`EXIT REPORT` cancels the report processing. It must appear in the FORMAT section of the report definition. It is useful after the program (or the user) becomes aware that a problem prevents the report from producing part of its intended output.

`EXIT REPORT` has the following effects:

- Terminates the processing of the current report.
- Deletes any intermediate files or temporary tables that were created in processing the report.

You cannot use the RETURN statement as a substitute for `EXIT REPORT`. An error is issued if `RETURN` is encountered within the definition of a report.

PRINT

Syntax:

```
PRINT
 {
    expression
  | COLUMN left-offset
  | PAGENO
  | LINENO
  | ns SPACES
  | [GROUP] COUNT(*) [ WHERE condition ]
  | [GROUP] PERCENT(*) [ WHERE condition ]
  | [GROUP] AVG( variable ) [ WHERE condition ]
  | [GROUP] SUM( variable ) [ WHERE condition ]
  | [GROUP] MIN( variable ) [ WHERE condition ]
  | [GROUP] MAX( variable ) [ WHERE condition ]
  | char-expression WORDWRAP [ RIGHT MARGIN rm ]
  | FILE "file-name"
 } [,...]
  [ ; ]
```

Notes:

1. *expression* is any legal language expression.
2. *left-offset* is described in COLUMN.
3. *ns* is described in SPACES.
4. *char-expression* is a string expression or a TEXT variable.
5. *file-name* is a string expression, or a quoted string, that specifies the name of a text file to include in the output from the report.

Warnings:

1. You cannot use `PRINT` to display a BYTE value. The string "<byte value>" is the only output from `PRINT` of any object that is not of the TEXT data type.

Usage:

This statement can include character data in the form of an ASCII file, a TEXT variable, or a comma-separated expression list of character expressions in the output of the

report. (For TEXT variable or filename, you cannot specify additional output in the same PRINT statement.) You cannot display a BYTE value. Unless its scope of reference is global or the current module, any program variable in expression list must be declared in the DEFINE section.

The PRINT FILE statement reads the contents of the specified filename into the report, beginning at the current character position. This statement permits you to insert a multiple-line character string into the output of a report. If *file-name* stores the value of a TEXT variable, the PRINT FILE *file-name* statement has the same effect as specifying PRINT *text-variable*. (But only PRINT variable can include the WORDWRAP operator)

PRINT statement output begins at the current character position, sometimes called simply the current position. On each page of a report, the initial default character position is the first character position in the first line. This position can be offset horizontally and vertically by margin and header specifications and by executing any of the following statements:

- The SKIP statement moves it down to the left margin of a new line.
- The NEED statement can conditionally move it to a new page.
- The PRINT statement moves it horizontally (and sometimes down).

Unless you use the keyword CLIPPED or USING, values are displayed with widths (including any sign) that depend on their declared data types.

Data Type	Default Print With
BYTE	N/A
CHAR	Length of character data type declaration.
DATE	DBDATE dependant, 10 if DBDATE = "MDY4/"
DATETIME	From 2 to 25, as implied in the data type declaration.
DECIMAL	(2 + p + s), where p is the precision and s is the scale from the data type declaration.
FLOAT	14
INTEGER	11
INTERVAL	From 3 to 25, as implied in the data type declaration.
MONEY	(2 + c + p + s), where c is the length of the currency defined by DBMONEY and p is the precision and s is the sacle from the data type declaration.
NCHAR	Length of character data type declaration.
NVARCHAR	Length current value in the variable.
SMALLFLOAT	14
SMALLINT	6
STRING	Length current value in the variable.
TEXT	Length current value in the variable.
VARCHAR	Length current value in the variable.

Unless you specify the FILE or WORDWRAP option, each PRINT statement displays output on a single line. For example, this fragment displays output on two lines:

```
01  PRINT fname, lname
02  PRINT city, ", ", state, " ", zipcode
```

If you terminate a PRINT statement with a semicolon, however, you suppress the implicit LINEFEED character at the end of the line. The next example has the same effect as the PRINT statements in the previous example:

```
01  PRINT fname;
02  PRINT lname
03  PRINT city, ", ", state, " ", zipcode
```

The expression list of a PRINT statement returns one or more values that can be displayed as printable characters. The expression list can contain report variables, built-in functions and operators. Some of these can appear only in a REPORT program block such as PAGENO, LINENO, PERCENT.

If the expression list applies the USING operator to format a DATE or MONEY value, the format string of the USING operator takes precedence over the DBDATE, DBMONEY, and DBFORMAT environment variables.

Aggregate report functions summarize data from several records in a report. The syntax and effects of aggregates in a report resemble those of SQL aggregate functions but are not identical.

The expression (in parentheses) that SUM( ), AVG( ), MIN( ), or MAX( ) takes as an argument is typically of a number or INTERVAL data type; ARRAY, BYTE, RECORD, and TEXT are not valid. The SUM( ), AVG( ), MIN( ), and MAX( ) aggregates ignore input records for which their arguments have null values, but each returns NULL if every record has a null value for the argument.

The GROUP keyword is an optional keyword that causes the aggregate function to include data only for a group of records that have the same value for a variable that you specify in an AFTER GROUP OF control block. An aggregate function can only include the GROUP keyword within an AFTER GROUP OF control block.

The optional WHERE clause allows you to select among records passed to the report, so that only records for which the Boolean expression is TRUE are included.

Example:

The following example is from the FORMAT section of a report definition that displays both quoted strings and values from rows of the customer table:

```
01  FIRST PAGE HEADER
02     PRINT COLUMN 30, "CUSTOMER LIST"
03     SKIP 2 LINES
04     PRINT "Listings for the State of ", thisstate
```

```
05  SKIP 2 LINES
06     PRINT "NUMBER", COLUMN 12, "NAME", COLUMN 35, "LOCATION",
07           COLUMN 57, "ZIP", COLUMN 65, "PHONE"
08     SKIP 1 LINE
09  PAGE HEADER
10     PRINT "NUMBER", COLUMN 12, "NAME", COLUMN 35, "LOCATION",
11            COLUMN 57, "ZIP", COLUMN 65, "PHONE"
12     SKIP 1 LINE
13  ON EVERY ROW
14     PRINT customer_num USING "###&", COLUMN 12, fname CLIPPED,
15          1 SPACE, lname CLIPPED, COLUMN 35, city CLIPPED, ", ",
16          state, COLUMN 57, zipcode, COLUMN 65, phone
```

PRINTX

Syntax:

```
PRINTX [NAME = identifier] expression
```

Notes:

1. *identifier* is the name to be used in the XML node.
2. *expression* is any legal language expression.

Usage:

The `PRINTX` statement is similar to PRINT, except that it prints data in XML format.

You typically write a complete report with `PRINTX` statements, to generate an XML output.

You can redirect the report output into a SAX document handler by calling the fgl_report_set_document_handler().

Note that unlike normal PRINT instructions, the PRINTX outputs both TEXT and BYTE data. The BYTE data is encoded to **Base64** before output.

NEED

Syntax:

```
NEED n LINE[S]
```

Notes:

1. *n* is the number of lines.

Usage:

This statement has the effect of a conditional SKIP TO TOP OF PAGE statement, the condition being that the number to which the integer expression evaluates is greater than the number of lines that remain on the current page.

The NEED statement can prevent the report from dividing parts of the output that you want to keep together on a single page. In the following example, the NEED statement causes the PRINT statement to send output to the next page unless at least six lines remain on the current page:

```
01  AFTER GROUP OF r.order_num
02    NEED 6 LINES
03    PRINT " ", r.order_date, "  ", GROUP SUM(r.total_price)
```

The LINES value specifies how many lines must remain between the line above the current character position and the bottom margin for the next PRINT statement to produce output on the current page. If fewer than LINES remain on the page, the report engine prints both the PAGE TRAILER and the PAGE HEADER.

The NEED statement does not include the BOTTOM MARGIN value when it compares LINES to the number of lines remaining on the current page. NEED is not valid in FIRST PAGE HEADER, PAGE HEADER, or PAGE TRAILER blocks.

PAUSE Console Only!

Syntax:

```
PAUSE [ "comment" ]
```

Notes:

1. *comment* is an optional comment to be displayed.

Usage:

Output is sent by default to the screen unless the START REPORT statement or the OUTPUT section specifies a destination for report output.

The PAUSE statement can be executed only if the report sends its output to the screen. It has no effect if you include a TO clause in either of these contexts:

- In the OUTPUT section of the report definition.
- In the START REPORT statement of the report driver.

Include the PAUSE statement in the PAGE HEADER or PAGE TRAILER block of the report. For example, the following code causes the runtime system to skip a line and pause at the end of each page of report output displayed on the screen:

```
01  PAGE TRAILER
02    SKIP 1 LINE
03    PAUSE "Press return to continue"
```

SKIP

Syntax:

```
SKIP { n LINE[S] | TO TOP OF PAGE }
```

Notes:

1. *n* is the number of lines.
2. The LINE and LINES keywords are synonyms in the SKIP statement.

Warnings:

1. The SKIP n LINES statement cannot appear within a CASE statement, a FOR loop, or a WHILE loop.
2. The SKIP TO TOP OF PAGE statement cannot appear in a FIRST PAGE HEADER, PAGE HEADER or PAGE TRAILER control block.

Usage:

The SKIP statement allows you to insert blank lines into report output or to skip to the top of the next page as if you had included an equivalent number of PRINT statements without specifying any expression list.

Output from any PAGE HEADER or PAGE TRAILER control block appears in its usual location.

Example:

```
01  FIRST PAGE HEADER
02    PRINT "Customer List"
03    SKIP 2 LINES
04    PRINT "Number        Name               Location"
05    SKIP 1 LINE
06  PAGE HEADER
07    PRINT "Number        Name               Location"
08    SKIP 1 LINE
09  ON EVERY ROW
10    PRINT r.customer_num, r.fname, r.city
```

COLUMN

Syntax:

```
COLUMN p
```

Notes:

1. *p* is the column position (starts at 1).

Usage:

COLUMN specifies the position in the current line of a report where output of the next value in a PRINT statement begins.

The COLUMN operator can appear in PRINT statements to move the character position forward within the current line.

The operand must be a non-negative integer that specifies a character position offset (from the left margin) no greater than the line width (that is, no greater than the difference (right margin - left margin). This designation moves the character position to a left-offset, where 1 is the first position after the left margin. If current position is greater than the operand, the COLUMN specification is ignored.

Example:

```
01  FIRST PAGE HEADER
02    PRINT "Customer List"
03    PRINT "Number", COLUMN 12,"Name", COLUMN 35,"Location"
04  PAGE HEADER
05    PRINT "Number", COLUMN 12,"Name", COLUMN 35,"Location"
06  ON EVERY ROW
07    PRINT customer_num, COLUMN 12,fname, COLUMN 35,city
```

LINENO

Syntax:

```
LINENO
```

Usage:

This operator takes no operand but returns the value of the line number of the report line that is currently printing. The report engine calculates the line number by calculating the number of lines from the top of the current page, including the TOP MARGIN.

Example:

In the following example, a PRINT statement instructs the report to calculate and display the current line number, beginning in the tenth character position after the left margin:

```
01  ON EVERY ROW
02    IF LINENO > 9 THEN
03      PRINT COLUMN 10, "Line:", LINENO USING "<<<"
04    END IF
```

PAGENO

Syntax:

```
PAGENO
```

Usage:

This operator takes no operand but returns the number of the page the report engine is currently printing.

You can use PAGENO in the PAGE HEADER or PAGE TRAILER block, or in other control blocks to number the pages of a report sequentially.

Example:

If you use the SQL aggregate COUNT(*) in the SELECT statement to find how many records are returned by the query, and if the number of records that appear on each page of output is both fixed and known, you can calculate the total number of pages, as in the following example:

```
01  FIRST PAGE HEADER
02    SELECT COUNT(*) INTO cnt FROM customer
03    LET y = cnt/50 -- Assumes 50 records per page
04  ON EVERY ROW
05    PRINT COLUMN 10, r.customer_num, ...
06  PAGE TRAILER
07    PRINT PAGE PAGENO USING "<<" OF cnt USING "<<"
```

If the calculated number of pages was 20, the first page trailer would be:

```
Page 1 of 20
```

PAGENO is incremented with each page, so the last page trailer would be:

```
Page 20 of 20
```

SPACES

Syntax:

```
n SPACES
```

Notes:

1. *n* is the number of spaces.

Usage:

This operator returns a string of blanks, equivalent to a quoted string containing the specified number of blanks.

In a PRINT statement, these blanks are inserted at the current character position.

Its operand must be an integer expression that returns a positive number, specifying an offset (from the current character position) no greater than the difference (right margin - current position). After `PRINT SPACES` has executed, the new current character position has moved to the right by the specified number of characters.

Outside PRINT statements, `SPACES` and its operand must appear within parentheses: `(n SPACES)`.

Example:

```
01 ON EVERY ROW
02    LET s = (6 SPACES), "=ZIP"
03    PRINT r.fname, 2 SPACES, r.lname, s
```

WORDWRAP

Syntax:

```
WORDWRAP [ RIGHT MARGIN tm ]
```

Notes:

1. *tm* defines the temporary right margin.

Usage:

The `WORDWRAP` operator automatically wraps successive segments of long character strings onto successive lines of report output. Any string value that is too long to fit

between the current position and the right margin is divided into segments and displayed between temporary margins:

- The current character position becomes the temporary left margin.
- Unless you specify `RIGHT MARGIN`, the right margin defaults to 132, or to the size value from the `RIGHT MARGIN` clause of the OUTPUT section or START REPORT instruction.

Specify `WORDWRAP RIGHT MARGIN` *`tm`* to set a temporary right margin, counting from the left edge of the page. This value cannot be smaller than the current character position or greater than right margin defined for the report. The current character position becomes the temporary left margin. These temporary values override the specified or default left and right margins of the report.

After the PRINT statement has executed, any explicit or default margins defined in the `RIGHT MARGIN` clause of the OUTPUT section or START REPORT instruction are restored.

The following PRINT statement specifies a temporary left margin in column 10 and a temporary right margin in column 70 to display the character string that is stored in the variable called *mynovel*:

```
01  PRINT COLUMN 10, mynovel WORDWRAP RIGHT MARGIN 70
```

The data string can include printable ASCII characters. It can also include the TAB (ASCII 9), LINEFEED (ASCII 10), and ENTER (ASCII 13) characters to partition the string into words that consist of sub-strings of other printable characters. Other nonprintable characters might cause runtime errors. If the data string cannot fit between the margins of the current line, the report engine breaks the line at a word division, and pads the line with blanks at the right.

From left to right, the report engine expands any TAB character to enough blank spaces to reach the next tab stop. By default, tab stops are in every eighth column, beginning at the left-hand edge of the page. If the next tab stop or a string of blank characters extends beyond the right margin, the report engine takes these actions:

1. Prints blank characters only to the right margin.
2. Discards any remaining blanks from the blank string or tab.
3. Starts a new line at the temporary left margin.
4. Processes the next word.

The report engine starts a new line when a word plus the next blank space cannot fit on the current line. If all words are separated by a single space, this action creates an even left margin. The following rules are applied (in descending order of precedence) to the portion of the data string within the right margin:

- Break at any LINEFEED, or ENTER, or LINEFEED, ENTER pair.
- Break at the last blank (ASCII 32) or TAB character before the right margin.
- Break at the right margin, if no character farther to the left is a blank, ENTER, TAB, or LINEFEED character.

The report engine maintains page discipline under the WORDWRAP option. If the string is too long for the current page, the report engine executes the statements in any page trailer and header control blocks before continuing output onto a new page.

For Japanese locales, a suitable break can also be made between the Japanese characters. However, certain characters must not begin a new line, and some characters must not end a line. This convention creates the need for KINSOKU processing, whose purpose is to format the line properly, without any prohibited word at the beginning or ending of a line.

Reports use the wrap-down method for WORDWRAP and KINSOKU processing. The wrap-down method forces down to the next line characters that are prohibited from ending a line. A character that precedes another that is prohibited from beginning a line can also wrap down to the next line. Characters that are prohibited from beginning or ending a line must be listed in the locale. 4GL tests for prohibited characters at the beginning and ending of a line, testing the first and last visible characters. The KINSOKU processing only happens once for each line. That is, no further KINSOKU processing occurs, even if prohibited characters are still on the same line after the first KINSOKU processing.

COUNT

Syntax:

```
[GROUP] COUNT(*) [ WHERE condition ]
```

Usage:

This aggregate returns the total number of records qualified by the optional WHERE condition.

Warnings:

1. You must include the (*) symbol.

Example:

The following fragment of a report definition uses the AFTER GROUP OF control block and GROUP keyword to form sets of records according to how many items are in each order. The last PRINT statement calculates the total price of each order, adds a shipping charge, and prints the result. Because no WHERE clause is specified here, GROUP SUM( ) combines the *total_price* of every item in the group included in the order.

```
01  AFTER GROUP OF number
02    SKIP 1 LINE
03    PRINT 4 SPACES, "Shipping charges for the order: ",
04          ship_charge USING "$$$$.&&"
```

```
05    PRINT 4 SPACES, "Count of small orders: ",
06         GROUP COUNT(*) WHERE total_price < 200.00 USING "##,###"
07    SKIP 1 LINE
08    PRINT 5 SPACES, "Total amount for the order: ",
09          ship_charge + GROUP SUM(total_price) USING "$$,$$$,$$$.&&"
```

PERCENT

Syntax:

```
[GROUP] PERCENT(*) [ WHERE condition ]
```

Usage:

This aggregate returns the percentage of the total number of records qualified by the optional `WHERE` condition.

Warnings:

1. You must include the `(*)` symbol.

SUM

Syntax:

```
[GROUP] SUM( expression ) [ WHERE condition ]
```

Usage:

This aggregate evaluates as the total of expression among all records or among records qualified by the optional `WHERE` clause and any `GROUP` specification.

Warnings:

1. If one of the values is NULL, it is ignored.
2. By default, if all values are NULL, the result of the aggregate is NULL. *See also:* Report Engine Configuration.

AVG

Syntax:

```
[GROUP] AVG( expression ) [ WHERE condition ]
```

Usage:

This aggregate evaluates as the average (that is, the arithmetic mean value) of expression among all records or among records qualified by the optional `WHERE` clause and any `GROUP` specification.

Warnings:

1. If one of the values is NULL, it is ignored.
2. By default, if all values are NULL, the result of the aggregate is NULL. *See also:* Report Engine Configuration.

MIN

Syntax:

```
[GROUP] MIN( expression ) [ WHERE condition ]
```

Usage:

For number, currency, and interval values, `MIN(expression)` returns the minimum value for *expression* among all records or among records qualified by the `WHERE` clause and any `GROUP` specification. For DATETIME or DATE data values, greater than means later and less than means earlier in time. Character strings are sorted according to their first character. If your program is executed in the default (U.S. English) locale, for character data types, greater than means after in the ASCII collating sequence, where a> A> 1, and less than means before in the ASCII sequence, where 1< A< a.

MAX

Syntax:

```
[GROUP] MAX( expression ) [ WHERE condition ]
```

Usage:

For number, currency, and interval values, `MAX(expression)` returns the maximum value for *expression* among all records or among records qualified by the `WHERE` clause

and any GROUP specification. For DATETIME or DATE data values, greater than means later and less than means earlier in time. Character strings are sorted according to their first character. If your program is executed in the default (U.S. English) locale, for character data types, greater than means after in the ASCII collating sequence, where a> A> 1, and less than means before in the ASCII sequence, where 1< A< a.

Localization

Summary:

- Localization Support
- Writing Programs
- Runtime System Settings
  - Language Settings
  - Numeric and Currency Settings
  - Date and Time Settings
- Database Client Settings
- Front-end Settings
- Runtime System Messages
- Troubleshooting
  - Locale settings (LANG) corrupted on Microsoft platforms
  - A form is displayed with invalid characters
  - Checking the locale configuration on UNIX platforms
  - Verifying if the locale is properly supported by the runtime system
  - How to retrieve the list of available locales on the system
  - How to retrieve the list of available codesets on the system
  - Using the charmap.alias file when client has different codeset names

See also: Localized Strings

Localization Support

Localization Support allows you to write BDL programs that follow a specific language and cultural rules. This includes single and multi-byte character set support, language-specific messages, as well as lexical/numeric/currency conventions.

Localization Support is based on the POSIX system libraries handling the locale. A locale is a set of language and cultural rules.

A BDL program needs to be able to determine its locale and act accordingly to be portable to different cultures.

Writing Programs

Runtime character set must match development character set

When writing a form or program source file, you use a specific character set. This character set depends upon the text editor or operating system settings you are using on the development platform. For example, when writing a string constant in a 4gl module, containing Arabic characters, you probably use the ISO-8859-6 character set. The

character set used used at runtime (during program execution) must match the character set used to write programs.

At runtime, a Genero program can only work in a specific character set. However, by using Localized Strings, you can start multiple instances of the same compiled program using different locales. For a given program instance the character set used by the strings resource files must correspond to the locale. Make sure the string identifiers use ASCII only.

Byte length semantics vs Character length semantics

Genero BDL uses byte length semantics: When defining a character data type like CHAR(n) or VARCHAR(n), n represents as a number of bytes, not a number of characters. In a single-byte character set like ISO-8859-1, any character is encoded on a unique byte, so the number of bytes equals the number of characters. But in a multi-byte character set, encoding requires more that one byte, so the number of bytes to store a multi-byte string is bigger as the number of characters. For example, in a BIG5 encoding, one Chinese character needs 2 bytes, so if you want to hold a BIG5 string with a maximum of 10 Chinese characters, you must define a CHAR(20). When using a variable-length encoding like UTF-8, characters can take one, two or more bytes, so you need to choose the right average to define CHAR or VARCHAR variables.

The definition of database columns using CHAR, VARCHAR, NCHAR and NVARCHAR types varies from one database vendor to another. Some use byte length semantics, other use character length semantics, and other provide both ways. For example, Informix uses bytes only; Oracle supports byte "CHAR(10 BYTE)" or character "CHAR(10 CHAR)" length semantics. SQL Server uses a single-byte character set for CHAR/VARCHAR and uses a 2-length Unicode character set (UCS-2) for NCHAR and NVARCHAR.

Other SQL elements like functions and operators are affected by the length semantic. For example, Informix LENGTH() function always returns a number of bytes, while Oracle's LENGTH() function returns a number of characters (use LENGTHB() to get the number of bytes with Oracle).

It is important to understand properly how the database servers handle multi-byte character sets. Check your database server reference manual: In most documentations you will find a "Localization" chapter which describes those concepts in detail.

For portability, we recommend to use byte length semantic based character data types in databases, because this corresponds to the length semantics used by Genero BDL (this is important when declaring variables by using DEFINE LIKE, which is based on database schemas).

Runtime System Settings

This section describes the settings defining the locale, changing the behavior of the runtime system.

Language Settings

The **LANG** environment variable defines the global settings for the language used by the application. This variable changes the behavior of the character handling functions, such as UPSHIFT, DOWNSHIFT. It also changes the handling of multi-byte characters. Invalid settings of **LANG** will cause compilation errors if a source file contains multi-byte characters.

With the **LANG** environment variable, you define the *language*, the *territory* (country) and the *codeset* (character set) to be used. The format of the value is normalized as follows, but may be specific on some operating systems:

```
language[_territory[.codeset]]
```

Warning: Most operating system vendors define specific set of values for the *language*, *territory* and *codeset*. For example, on a UNIX platform, you typically set "en_US.ISO8859-1" for a US English locale, while Microsoft Windows supports "English_USA.1252", or "en_us.1252". For more details about supported locales, please refer to the operating system documentation (search for the 'setlocale' function).

See also Troubleshooting to learn how to check if a locale is properly set, and list the locales installed on your system.

Numeric and Currency Settings

To perform decimal to/from string conversions, the runtime system uses the DBMONEY or DBFORMAT environment variables. These variables define hundreds / decimal separators and currency symbols for MONEY data types.

The **LC_MONETARY** and **LC_NUMERIC** standard environment variables, defining numeric and monetary rules, are ignored.

Date and Time Settings

To perform date to/from string conversions, the runtime system uses by default the DBDATE environment variable. When assigning a string to a date variable, the standard environment variable **LC_TIME** is ignored.

When using the FORMAT field attribute or the USING operator to format dates with abbreviated day and month names - by using **ddd** / **mmm** markers - the system uses English-language based texts for the conversion. This means, day (ddd) and month (mmm) abbreviations are not localized according to the locale settings, they will always be in English.

Database Client Settings

This section describes the settings defining the locale for the database client.

Each database vendor has its own locale settings.

Warning: You must properly configure the database client locale in order to send/receive data to the database server, according to the locale used by your application. Both database client locale and application locale settings must match (you cannot have a database client locale in Japanese and a runtime locale in Chinese).

Here is the list of environment variables defining the locale used by the application, for each supported database client:

Database Client	Settings
Genero DB	The character set used by the client is defined by the **characterset** ODBC DSN configuration parameter. If this parameter is not set, it defaults to ASCII. Before version 3.80, the character set was defined by the **ANTS_CHARSET** environment variable.
Oracle	The client locale settings can be set with environment variables like **NLS_LANG**, or after connection, with the ALTER SESSION instruction. By default, the client locale is set from the database server locale.
Informix	The client locale is defined by the **CLIENT_LOCALE** environment variable. For backward compatibility, if CLIENT_LOCALE is not defined, other settings are used if defined (DBDATE / DBTIME / GL_DATE / GL_DATETIME, as well as standard LC_* variables).
IBM DB2	The client locale is defined by the **DB2CODEPAGE** profile variable. You must set this variable with the db2set command. If DB2CODEPAGE is not set, DB2 uses the operating system code page on Windows and the LANG environment variable on Unix.
Microsoft SQL Server	The client locale is defined by the Window operating **system locale** where the database client is installed.
PostgreSQL	The client locale can be set with the **PGCLIENTENCODING** environment variable, with the *client_encoding* configuration parameter in postgresql.conf, or after connection, with the SET CLIENT_ENCODING instruction. Check the *pg_conversion* system table for available character set conversions.

MySQL	The client locale is defined by the ***default-character-set*** option in the configuration file, or after connection, with the SET NAMES and SET CHARACTER SET instructions.
Sybase ASA	The client locale is defined by the operating **system locale** where the database client is installed.

See database vendor documentation for more details.

Front-End Settings

The front-end workstation must support the character set used on the runtime system side. You can refer to each front-end documentation to check the list of supported character sets. The host operating system must also be able to handle the character set. For instance, a Western-European Windows is not configured to handle Arabic applications. If you start an Arabic application, some graphical problems may occur (for instance the title bar won't display Arabic characters, but unwanted characters instead).

Runtime System Messages

Predefined runtime system error messages are stored in the **.iem** system message files. The system message files use the same technique as user defined message files (See Message Files). The default message files are located in the **FGLDIR/msg/en_US** directory (**.msg** sources are provided).

For backward compatibility with Informix 4gl, some of these system error messages are used by the runtime system to report a "normal" error during a dialog instruction. For example, end users may get the error -1309 "There are no more rows in the direction you are going" when scrolling an a DISPLAY ARRAY list.

Here are some examples of system messages that can appear during a dialog:

Number	Description
-1204	Invalid year in date.
-1304	Error in field.
-1305	This field requires an entered value.
-1306	Please type again for verification.
-1307	Cannot insert another row - the input array is full.
-1309	There are no more rows in the direction you are going.
and more...	

While it is recommended to use Localized Strings to internationalize application messages, you might need to translated the default system messages to a specific locale and language, or you might just want to customize the English messages.

With this technique, you can deploy multiple message files in different languages and locales in the same FGLDIR/msg directory.

To use your own customized system messages, do the following:

1. Create a new directory under $FGLDIR/msg, using the same name as your current locale.
   For example, if `LANG=fr_FR.ISO8859-1`, you must create `$FGLDIR/msg/fr_FR.ISO8859-1`.
2. Copy the original system message source files (.msg) from `$FGLDIR/msg/en_US` to the locale-specific directory.
   For example: `$FGLDIR/msg/$LANG`.
3. Modify the source files with the **.msg** suffix.
4. Re-compile the message files with the fglmkmsg tool to produce **.iem** files.
5. Run a program to check if the new messages are used.

Warnings:

1. **The locale can be set with different environment variables (see setlocale manual pages for more details). To identify the locale name, the runtime system first looks for the LC_ALL value, then LC_CTYPE and finally LANG.**
2. **Pay attention to locale settings when editing message files: You must use the same locale as the one used at runtime.**

Troubleshooting

Locale settings (LANG) corrupted on Microsoft platforms

On Microsoft Windows XP / 2000 platforms, some system updates (Services Pack 2) or Office versions do set the LANG environment variable with a value for Microsoft applications (for example 1033).

Such value is not recognized by Genero as a valid locale specification. Make sure that the LANG environment variable is properly set in the context of Genero applications.

A form is displayed with invalid characters

You may have different codesets on the client workstation and the application server. The typical mistake that can happen is the following: You have edited a form-file with the encoding CP1253; you compile this form-file on a UNIX-server (encoding ISO-8859-7). When displaying the form, invalid characters will appear. This is usually the case when you write your source file under a Windows system (that uses Microsoft Code Page encodings), and use a Linux server (that uses ISO codepages).

Warning: All source files must be created/edited in the encoding of the server (where fglcomp and fglrun will be executed).

Checking the locale configuration on Unix platforms

On Unix systems, the **locale** command without parameters outputs information about the current locale environment.

Once the **LANG** environment variable is set, check that the locale environment is correct:

```
$ export LANG=en_US.ISO8859-1
$ locale
LANG=en_US.ISO8859-1
LC_CTYPE="en_US.ISO8859-1"
LC_NUMERIC="en_US.ISO8859-1"
LC_TIME="en_US.ISO8859-1"
LC_COLLATE="en_US.ISO8859-1"
LC_MONETARY="en_US.ISO8859-1"
LC_MESSAGES="en_US.ISO8859-1"
LC_PAPER="en_US.ISO8859-1"
LC_NAME="en_US.ISO8859-1"
LC_ADDRESS="en_US.ISO8859-1"
LC_TELEPHONE="en_US.ISO8859-1"
LC_MEASUREMENT="en_US.ISO8859-1"
LC_IDENTIFICATION="en_US.ISO8859-1"
LC_ALL=
```

If the locale environment is not correct, then you should check the value of the following environment variables : LC_ALL, LC_CTYPE, LC_NUMERIC, LC_TIME, LC_COLLATE, ... value.

The following examples show the effect of LC_ALL and LC_CTYPE on locale configuration. The LC_ALL variable overrides all other LC_.... variables values.

```
$ export LANG=en_US.ISO8859-1
$ export LC_ALL=POSIX
$ export LC_CTYPE=fr_FR.ISO8859-15
$ locale
LANG=en_US.ISO8859-1
LC_CTYPE="POSIX"
LC_NUMERIC="POSIX"
LC_TIME="POSIX"
LC_COLLATE="POSIX"
LC_MONETARY="POSIX"
LC_MESSAGES="POSIX"
LC_PAPER="POSIX"
LC_NAME="POSIX"
LC_ADDRESS="POSIX"
LC_TELEPHONE="POSIX"
LC_MEASUREMENT="POSIX"
LC_IDENTIFICATION="POSIX"
LC_ALL=POSIX
$ fglrun -i mbcs
```

```
LANG honored : yes
Charmap      : ANSI_X3.4-1968
Multibyte    : no
Stateless    : yes
```

The charset used is the ASCII charset. Clearing the LC_ALL environment variable produces the following output:

```
$ unset LC_ALL
$ locale
LANG=en_US.ISO8859-1
LC_CTYPE=fr_FR.ISO8859-15
LC_NUMERIC="en_US.ISO8859-1"
LC_TIME="en_US.ISO8859-1"
LC_COLLATE="en_US.ISO8859-1"
LC_MONETARY="en_US.ISO8859-1"
LC_MESSAGES="en_US.ISO8859-1"
LC_PAPER="en_US.ISO8859-1"
LC_NAME="en_US.ISO8859-1"
LC_ADDRESS="en_US.ISO8859-1"
LC_TELEPHONE="en_US.ISO8859-1"
LC_MEASUREMENT="en_US.ISO8859-1"
LC_IDENTIFICATION="en_US.ISO8859-1"
LC_ALL=
$ fglrun -i mbcs
Error: locale not supported by C library, check LANG.
$ locale charmap
ANSI_X3.4-1968
```

After clearing the LC_ALL value, the value of the variable LC_CTYPE is used. It appears that it is not correct. After clearing this value we get the following output:

```
$ unset LC_CTYPE
$ locale
LANG=en_US.ISO8859-1
LC_CTYPE="en_US.ISO8859-1"
LC_NUMERIC="en_US.ISO8859-1"
LC_TIME="en_US.ISO8859-1"
LC_COLLATE="en_US.ISO8859-1"
LC_MONETARY="en_US.ISO8859-1"
LC_MESSAGES="en_US.ISO8859-1"
LC_PAPER="en_US.ISO8859-1"
LC_NAME="en_US.ISO8859-1"
LC_ADDRESS="en_US.ISO8859-1"
LC_TELEPHONE="en_US.ISO8859-1"
LC_MEASUREMENT="en_US.ISO8859-1"
LC_IDENTIFICATION="en_US.ISO8859-1"
LC_ALL=
$ locale charmap
ISO-8859-1
$ fglrun -i mbcs
LANG honored : yes
Charmap      : ISO-8859-1
Multibyte    : no
Stateless    : yes
```

Verifying if the locale is properly supported by the runtime system

You can check if the **LANG** locale is supported properly by using the **-i mbcs** option of the compilers and runner programs:

```
$ fglcomp -i mbcs
LANG honored : yes
Charmap      : ANSI_X3.4-1968
Multibyte    : no
Stateless    : yes
```

The lines printed with **-i info** option indicate if the locale can be supported by the operating system libraries. Here is a short description of each line:

Verification Parameter	Description
LANG Honored	This line indicates that the current locale configuration has been correctly set. *Check if the indicator shows 'yes'.*
Charmap	This is the name of the character set used by the runtime system.
Multibyte	This line indicates if the character set is multi-byte. *Can be 'yes' or 'no'.*
Stateless	A few character sets are using an internal state that can change during the character flow. Only stateless character sets can be supported by Genero. *Check if the indicator shows 'yes'.*

How to retrieve the list of available locales on the system

On Unix systems, the locale command with the parameter '-a' writes the names of available locales.

```
$ locale -a
...
en_US
en_US.iso885915
en_US.utf8
en_ZA
en_ZA.utf8
en_ZW
...
```

How to retrieve the list of available codesets on the system

On Unix systems, the locale command with the parameter '-m' writes the names of available codesets.

```
$ locale -m
...
```

```
ISO-8859-1
ISO-8859-10
ISO-8859-13
ISO-8859-14
ISO-8859-15
...
```

Using the charmap.alias file when client has different codeset names

The name of the codeset can be different from one system to another. The file $FGLDIR/etc/charmap.alias is used to provide the translation of the local name to a generic name. The generic name is the name sent to the front-end. It is the translated name that appears when the command 'fglrun -i mbcs' is used. The local codeset name is the value obtained using the system call 'nl_langinfo(CODESET)'. Note: This file might be incomplete.

An example of locale configuration on HP

```
$ export LANG=en_US.iso88591
$ locale
LANG=en_US.iso88591
LC_CTYPE="en_US.iso88591"
LC_COLLATE="en_US.iso88591"
LC_MONETARY="en_US.iso88591"
LC_NUMERIC="en_US.iso88591"
LC_TIME="en_US.iso88591"
LC_MESSAGES="en_US.iso88591"
LC_ALL=
$ locale charmap
"iso88591.cm"
```

The charmap.alias file contains the following line:

```
iso88591 ISO8859-1
```

The name sent to the client is ISO-8859-1 instead of iso88591.

The following C program should compile, and outputs the current codeset name.

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <langinfo.h>
int main()
{
  setlocale(LC_ALL, "");
  printf("%s\n", nl_langinfo(CODESET));
  exit(0);
}
```

With the previous example this program outputs:

```
iso88591
```

Localized Strings

Summary:

- What are Localized Strings?
- Syntax
- Source String Files
- Extracting Strings
- Compiling String Files
- Using String Files at Runtime
- Predefined Application Strings
- Example

See also: Programs, FGLPROFILE

Definition

Localized Strings provide a means of writing applications in which the text of strings can be customized on site. This feature can be used to implement internationalization in your application, or to use site-specific text (for example, when business terms change based on territory).

This string localization feature does not define language identification. It is a simple way to define external resource files which the runtime system can search, in order to assign text to strings in the BDL application. The text is replaced at runtime in the p-code modules (**42m**), in the compiled form specification files (**42f**), and in any XML resource files loaded in the Abstract User Interface tree (**4ad**, **4st**, **4tb**, and so on).

By using a simple notation, you can identify the Localized Strings in the source code:

```
01 MAIN
02   DISPLAY %"my text"
03 END MAIN
```

The fglcomp and fglform compilers have been extended to support a new option to extract the Localized Strings. This way, the Localized Strings can be extracted into Source String Files.

From the original Source String File, you can create other files containing different text (for example, one file for each language you want to support).

You must use the fglmkstr tool to compile the source string files into a binary version. By convention, compiled string resource files must have the **42s** extension.

By default, the compiled string files are loaded at runtime, according to the name of the program (**42r**). It is also possible to define global string files in the FGLPROFILE configuration file. *See also:* Using String Files at Runtime.

In **42m** p-code modules, the Localized Strings are coded in our specific binary format. But, for XML files such as compiled form files (**42f**), the localized strings must be identified with a specific node, following the XML standards.

To support localized strings in XML files, any file loaded into the Abstract User Interface tree is parsed to search for `<LStr>` nodes.

The `<LStr>` nodes define the same attributes as in the parent node with localized string identifiers, for example:

```
01 <Label text="Hello!" >
02   <LStr text="label01" />
03 </Label>
```

The runtime system automatically replaces corresponding attributes in the parent node (text="Hello!"), with the localized text found in the compiled string files, according to the string identifier (label01). After interpretation, the `<LStr>` nodes are removed from the XML data.

Syntax

Syntax 1: Static Localized String

```
%"sid"
```

Syntax 2: Dynamic Localized String

```
LSTR(eid)
```

Notes:

1. *sid* is a character string literal that defines both the string identifier and the default text.
2. *eid* is a character string expression used at runtime as string identifier to load the text.

Usage:

A Localized String can be used in the source code of program modules or form specification files to identify a text string that must be converted at runtime.

Static Localized Strings

A Localized String begins with a percent sign (%), followed by the name of the string which will be used to identify the text to be loaded. Since the name is a string, you can use any kind of characters in the name, but it is recommended that you use a proper naming convention. For example, you can specify a path by using several identifiers separated by a dot, without any special characters such as space or tab:

```
01 MAIN
02   DISPLAY %"common.helloworld"
03 END MAIN
```

The string after the percent sign defines both the localized string identifier and the default text to be used for extraction, or the default text when no string resource files are provided at runtime.

You can use this notation in form specification files as well, at any place where a string literal can be used.

```
01 LAYOUT
02   VBOX
03     GROUP g1 (TEXT=%"group01")
04 ...
```

Warning: It is not possible to specify a static localized string directly in the area of containers like GRID, TABLE or SCROLLGRID. You must use label fields to use localized strings in layout labels:

```
01 LAYOUT
02   GRID
03   {
04     [lab01  |f001             ]
05   {
06   END
07 END
08 ATTRIBUTES
09 LABEL lab01 : TEXT=%"myform.label01";
10 EDIT f001 = FORMONLY.field01;
11 END
```

Dynamic Localized Strings

The language provides a special operator to load a Localized String dynamically, using an expression as string identifier. The name of this operator is `LSTR()`, and the syntax is described above.

The following code example builds a Localized String identifier with an integer and loads the corresponding string with the `LSTR()` operator:

```
01 MAIN
02   DEFINE n INTEGER
03   LET n = 234
04   DISPLAY LSTR("str"||n)  -- loads string 'str234'
05 END MAIN
```

See also: The SFMT() operator

Source String Files

By convention, the source files of Localized Strings have the **.str** extension.

Defining a string:

You define a list of string identifiers, and the corresponding text, by using the following syntax:

```
"identifier" = "string"
```

Special characters:

The fglmkstr compiler accepts the backslash "\" as the escape character, to define non-printable characters:

```
\l   \n   \r   \t   \\
```

Example:

```
01 "id001" = "Some text"
02 "this.is.a.path.for.a.very.long.string.identifier" = "Customer List"
03 "special.characters.backslash" = "\\"
04 "special.characters.newline" = "\n"
```

Extracting Strings

In order to extract Localized String from Source String Files, use the fglcomp and fglform compilers with the `-m` option:

```
$ fglcomp -m mymodule.4gl
```

The compilers dumps all localized string to stdout. This output can be redirected to a file to generate the default Source String File with all the localized strings used in the 4gl file.

Compiling String Files

The Source String Files (**.str**) must be compiled to binary files (**.42s**) in order to be used at runtime.

To compile a Source String File, use the fglmkstr compiler:

```
$ fglmkstr filename.str
```

This tool generates a **.42s** file with the *filename* prefix.

Using String Files at Runtime

Distributing string resource files

The "`42s`" Compiled String Files must be distributed with the program files in a directory specified in the DBPATH environment variable.

How does the runtime system load the strings?

The string files are loaded in the following order:

1. the files defined in FGLPROFILE (see below),
2. a file having the same name prefix as the current "`42r`" program,
3. a file with the name "`default.42s`".

For each string file, the runtime system searches in following directories:

1. in the current directory,
2. in the path list defined in the DBPATH environment variable,
3. in `FGLDIR/lib`.

A string is loaded in memory only once (if the same string is defined in another file, it is ignored).

What happens if a string is not defined in a resource file?

If a localized string is not defined in a resource files, the runtime system uses the string identifier as default text.

What happens if a string is defined more that once?

When a localized string is defined in several resource files, the runtime system uses the first string found.

For example, if the string "hello" is defined in `program.42s` as "hello from program", and in `default.42s` as "hello from default", the runtime system will use the text "hello from program".

Defining a list of string resource files in FGLPROFILE

You can specify a list of Compiled String Files with entries in the FGLPROFILE configuration file. The file name must be specified without a file extension. The runtime system searches for a file with the "`42s`" extension in the current directory and in the path list defined in the DBPATH environment variable.

List of resource files

To define the list of resource files to be used, specify the total number of files with:

```
fglrun.localization.file.count = integer
```

And for each file, define the filename (without the **42s** extension), including an index number, with:

```
fglrun.localization.file.index.name = "filename"
```

Start *index* at 1.

Warning switches

If the text of a string is not found at runtime, the DVM can show a warning, for development purposes.

```
fglrun.localization.warnKeyNotFound = boolean
```

By default, this warning switch is disabled.

Predefined Application Strings

What is a Predefined Application String?

In some situations, the runtime system needs to display texts to the user. For example, the runtime system library includes a report viewer, which displays a form. By default the texts are in English, and you may need to localize the texts in another language. So the strings of this component must be 'localizable', as other application strings.

To customize the built-in strings, the runtime system uses the mechanism of localized strings.

All strings used by the runtime system are centralized in a unique file:

```
$FGLDIR/src/default.str
```

which is compiled into:

```
$FGLDIR/lib/default.4ls
```

This file is always loaded by the runtime system.

To overwrite the defaults, you can re-define these strings in your own localized string files. *See also:* Using String Files at Runtime.

Example

The Source String File "common.str" (a compiled version must be created):

```
01 "common.accept" = "OK"
02 "common.cancel" = "Cancel"
03 "common.quit" = "Quit"
```

The Source String File "actions.str" (a compiled version must be created):

```
01 "action.append" = "Append"
02 "action.modify" = "Modify"
03 "action.delete" = "Delete"
```

The Source String File "customer.str" (a compiled version must be created):

```
01 "customer.mainwindow.title" = "Customers"
02 "customer.listwindow.title" = "Customer List"
03 "customer.l_custnum" = "Number:"
04 "customer.l_custname" = "Name:"
05 "customer.c_custname" = "The customer name"
06 "customer.g_data" = "Customer data"
07 "customer.g_actions" = "Actions"
08 "customer.qdelete" = "Are you sure you want to delete this
customer?"
```

The FGLPROFILE configuration file parameters:

```
01 fglrun.localization.file.count = 2
02 fglrun.localization.file.1.name = "common"
03 fglrun.localization.file.2.name = "actions"
```

Remark: The 'customer' string file does not have to to listed in FGLPROFILE since it is loaded as it has the same name as the program.

The form specification file "f1.per":

```
01 LAYOUT (TEXT=%"customer.mainwindow.title")
02   GRID
03   {
04    <g g1                                    >
05     [lab1       ] [f01            ]
```

```
06     [lab2       ] [f02                    ]
07
08    <g g2                                   >
09     [b1               ] [b2                ]
10
11   }
12   END
13 END
14 ATTRIBUTES
15   LABEL  lab1 : TEXT=%"customer.l_custnum";
16   EDIT   f01  = FORMONLY.custnum;
17   LABEL  lab2 : TEXT=%"customer.l_custname";
18   EDIT   f02  = FORMONLY.custname, COMMENT=%"customer.c_custname";
19   BUTTON b1   : edit, TEXT=%"action.modify";
20   BUTTON b2   : quit, TEXT=%"common.quit";
21   GROUP  g1   : TEXT=%"customer.g_data";
22   GROUP  g2   : TEXT=%"customer.g_actions";
23 END
```

The program "customer.4gl" using the strings file:

```
01 MAIN
02   DEFINE rec RECORD
03              custnum INTEGER,
04              custname CHAR(20)
05            END RECORD
06   OPEN WINDOW w1 WITH FORM "f1"
07   MENU
08     ON ACTION edit
09        INPUT BY NAME rec.*
10     ON ACTION quit
11        EXIT MENU
12   END MENU
13 END MAIN
```

Built-in Functions

Summary:

- What is a built-in function?
- List of built-in functions
- List of de-supported functions
- The key code table

See also: Utility Functions, Variables, Functions, Operators, Built-in Classes.

What is a built-in function?

A built-in function is a predefined function that is included in the runtime system or provided as a library function automatically linked to your programs. You do not have to link with any additional BDL library to create your programs. The built-in functions are part of the language.

See also Utility Functions.

Warnings:

1. Do not confuse built-in functions with SQL aggregate functions like:
   - `AVG( )`
   - `MAX( )`
   - `MIN( )`
   - `SUM( )`
2. Do not confuse built-in functions with operators. Some operators have the same syntax as functions, but these are real language operators that have a specific order of precedence. Operators can be used in different contexts according to the BDL grammar. See for example:
   - YEAR(date)
   - MONTH(date)
   - DAY(date)
   - WEEKDAY(date)
   - MDY(integer,integer,integer)
   - GET_FLDBUF(field)
   - INFIELD(field)
   - FIELD_TOUCHED(field)

List of built-in functions

Function	Description
ARG_VAL()	Returns a command line argument by position.

ARR_COUNT()	Returns the number of records entered in a program array during or after execution of the INPUT ARRAY statement.
ARR_CURR()	Returns the current row in a DISPLAY ARRAY or INPUT ARRAY.
DOWNSHIFT()	Returns a string value in which all uppercase characters in its argument are converted to lowercase.
ERR_GET()	Returns the text corresponding to an error number.
ERR_PRINT()	Prints in the error line the text corresponding to an error number.
ERR_QUIT()	Prints in the error line the text corresponding to an error number and terminates the program.
ERRORLOG()	Copies the string passed as a parameter into the error log file.
FGL_BUFFERTOUCHED()	Returns TRUE if the current input buffer was modified since the field was selected.
FGL_DIALOG_GETBUFFER()	Returns the value of the current field as a string.
FGL_DIALOG_GETBUFFERLENGTH()	When using a paged display array, returns the number of rows to fill the array buffer.
FGL_DIALOG_GETBUFFERSTART()	When using a paged display array, returns the row offset to fill the array buffer.
FGL_DIALOG_GETCURSOR()	Returns the position of the edit cursor in the current field.
FGL_DIALOG_GETFIELDNAME()	Returns the name of the current input field.
FGL_DIALOG_GETKEYLABEL()	Returns the text associated to a key in the current interactive instruction.
FGL_DIALOG_INFIELD()	Returns TRUE if the field passed as a parameter is the current input field.
FGL_DIALOG_SETBUFFER()	Sets the value of the current

	field as a string.
FGL_DIALOG_SETCURRLINE()	Moves to a specific row in a record list.
FGL_DIALOG_SETCURSOR()	Sets the position of the input cursor within the current field.
FGL_DIALOG_SETFIELDORDER()	Enables or disables field order constraint.
FGL_DIALOG_SETKEYLABEL()	Sets the text associated to a key for the current interactive instruction.
FGL_DRAWBOX()	Draws a rectangle based on character terminal coordinates in the current open window.
FGL_DRAWLINE()	Draws a line based on character terminal coordinates in the current open window.
FGL_GETCURSOR()	Returns the position of the edit cursor in the current field.
FGL_GETENV()	Returns the value of the environment variable having the name you specify as argument.
FGL_GETFILE()	Transfers a file from the front-end to the application server machine.
FGL_GETHELP( )	Returns the help message according to a number.
FGL_GETKEYLABEL()	Returns the default label associated to a key.
FGL_GETPID( )	Returns the system process id.
FGL_GETRESOURCE( )	Returns the value of an FGLPROFILE entry.
FGL_GETVERSION( )	Returns the build number of the runtime system.
FGL_GETWIN_HEIGHT()	Returns the number of rows of the current window.
FGL_GETWIN_WIDTH()	Returns the width of the current window as a number of columns.
FGL_GETWIN_X()	Returns the horizontal position of the current window.
FGL_GETWIN_Y()	Returns the vertical position of the current window.
FGL_KEYVAL()	Returns the key code of a

	logical or physical key.
FGL_LASTKEY()	Returns the key code of the last key pressed by the user.
FGL_PUTFILE()	Transfers a file from from the application server machine to the front-end.
FGL_REPORT_PRINT_BINARY_FILE()	Prints a file containing binary data during a report.
FGL_REPORT_SET_DOCUMENT_HANDLER()	Defines the document handler to be used for a report.
FGL_SET_ARR_CURR()	Sets the current line in a record list.
FGL_SETENV()	Sets an environment variable
FGL_SETKEYLABEL()	Sets the default label associated to a key.
FGL_SETSIZE()	Sets the size of the main application window.
FGL_SETTITLE()	Sets the title of the main application window.
FGL_SYSTEM()	Starts a program in a UNIX terminal emulator when using a graphical front end.
FGL_WIDTH()	Returns the number of columns needed to represent the string.
FGL_WINDOW_GETOPTION()	Returns the attributes of the current window.
LENGTH()	Returns the number of characters of the string passed as a parameter.
NUM_ARGS()	Returns the number of program arguments.
ORD()	Returns the ASCII value of the first byte of a character expression.
SCR_LINE()	Returns the number of the current screen record in its screen array.
SET_COUNT()	Specifies the number of records that contain data in a static array.
SHOWHELP()	Displays a runtime help message, corresponding to its specified argument, from the

	current help file.
STARTLOG()	Initializes error logging and opens the error log file passed as a parameter.
UPSHIFT()	Returns a string value in which all lowercase characters in its argument are converted to uppercase.

List of de-supported built-in functions:

Function	Description
FGL_FORMFIELD_GETOPTION()	Returns attributes of a specified form field.
FGL_GETKEY()	Waits for a keystroke and returns the key code.
FGL_GETUITYPE()	Returns the type of the front end.
FGL_SCR_SIZE( )	Returns the number of rows of a screen array of the current form.
FGL_WINDOW_OPEN( )	Opens a new window with coordinates and size.
FGL_WINDOW_OPENWITHFORM( )	Opens a new window with coordinates and form.
FGL_WINDOW_CLEAR( )	Clears the window having the name that is passed as a parameter.
FGL_WINDOW_CLOSE( )	Closes the window having the name that is passed as a parameter.
FGL_WINDOW_CURRENT( )	Makes current the window having the name that is passed as a parameter.

ARG_VAL( )

Purpose:

This function returns a command line argument by position.

Context:

1. At any place in the program.

Syntax:

```
CALL ARG_VAL( position INTEGER ) RETURNING result STRING
```

Notes:

1. *position* is the argument position. 0 returns the name of the program, 1 returns the first argument.

Usage:

This function provides a mechanism for passing values to the program through the command line that invokes the program. You can design a program to expect or allow arguments after the name of the program in the command line.

Like all built-in functions, `ARG_VAL()` can be invoked from any program block. You can use it to pass values to MAIN, which cannot have formal arguments, but you are not restricted to calling `ARG_VAL()` from the MAIN statement. You can use the `ARG_VAL()` function to retrieve individual arguments during program execution. You can also use the NUM_ARGS() function to determine how many arguments follow the program name on the command line.

If *position* is greater than 0, `ARG_VAL(position)` returns the command-line argument used at a given position. The value of *position* must be between 0 and the value returned by NUM_ARGS(), the number of command-line arguments. The expression `ARG_VAL(0)` returns the name of the application program.

See also: NUM_ARGS().

NUM_ARGS( )

Purpose:

This function returns the number of program arguments.

Context:

1. At any place in the program.

Syntax:

```
CALL NUM_ARGS( ) RETURNING result INTEGER
```

Notes:

1. returns 0 if no arguments are passed to the program.

See also: ARG_VAL().

SCR_LINE( )

Purpose:

This function returns the number of the current screen record in its screen array.

Context:

1. During a DISPLAY ARRAY or INPUT ARRAY statement.

Syntax:

```
CALL SCR_LINE( ) RETURNING result INTEGER
```

Notes:

1. The current record is the line of a screen array that is highlighted at the beginning of a `BEFORE ROW` or `AFTER ROW` clause.

Warnings:

1. When using new graphical objects such as TABLEs, this function can return an invalid screen array line number, because the current row may not be visible if the user scrolls in the list with scrollbars.

See also: ARR_CURR().

SET_COUNT( )

Purpose:

This function specifies the number of records that contain data in a static array.

Context:

1. Before a DISPLAY ARRAY or INPUT ARRAY statement.

Syntax:

```
CALL SET_COUNT( nbrows INTEGER )
```

Notes:

1. *nbrows* defines the number of rows in the static array.
2. Using this function is equivalent to the `COUNT` attribute of INPUT ARRAY and DISPLAY ARRAY statements.

Usage:

When using a static array in an INPUT ARRAY (with `WITHOUT DEFAULTS` clause) or a DISPLAY ARRAY statement, you must specify the total number of records in the array. In typical applications, these records contain the values in the retrieved rows that a `SELECT` statement returned from a database cursor. You specify the number of rows with the `SET_COUNT()` function or with the `COUNT` attribute of INPUT ARRAY and DISPLAY ARRAY statements.

Warning: You do not have to specify the number of rows when using a dynamic array. When using a dynamic array, the number of rows is defined by the getLength() method of the array object.

See also: ARR_CURR(), FGL_SET_ARR_CURR().

ARR_COUNT( )

Purpose:

This function returns the number of records entered in a program array during or after execution of the INPUT ARRAY statement.

Context:

1. Can be called at any place in the program, but should be limited to usage inside or after INPUT ARRAY blocks.

Syntax:

```
CALL ARR_COUNT( ) RETURNING result INTEGER
```

Notes:

1. Returns the current number of records that exist in the current array.
2. Typically used inside INPUT ARRAY blocks.

Usage:

You can use `ARR_COUNT()` to determine the number of program records that are currently stored in a program array. In typical FGL applications, these records

correspond to values from the result set of retrieved database rows from the most recent query. By first calling the SET_COUNT() function or by using the `COUNT` attribute of INPUT ARRAY, you can set an upper limit on the value that `ARR_COUNT()` returns.

`ARR_COUNT()` returns a positive integer, corresponding to the index of the furthest record within the program array that the user accessed. Not all the rows *counted* by `ARR_COUNT()` necessarily contain data (for example, if the user presses the DOWN ARROW key more times than there are rows of data).

See also: INPUT ARRAY, ARR_CURR().

ARR_CURR( )

Purpose:

This function returns the current row in a DISPLAY ARRAY or INPUT ARRAY.

Context:

1. During a DISPLAY ARRAY or INPUT ARRAY statement.

Syntax:

```
CALL ARR_CURR( ) RETURNING result INTEGER
```

Usage:

The `ARR_CURR()` function returns an integer value that identifies the current row of a list of rows in a INPUT ARRAY or DISPLAY ARRAY instruction. The first row is numbered 1.

You can pass `ARR_CURR()` as an argument when you call a function. In this way the function receives as its argument the current record of whatever array is referenced in the INPUT ARRAY or DISPLAY ARRAY statement.

The `ARR_CURR()` function can be used to force a FOR loop to begin beyond the first line of an array by setting a variable to `ARR_CURR()` and using that variable as the starting value for the FOR loop.

The built-in functions `ARR_CURR()` and SCR_LINE() can return different values if the program array is larger than the screen array.

See also: INPUT ARRAY, DISPLAY ARRAY, ARR_COUNT(), FGL_SET_ARR_CURR(), SCR_LINE().

ERR_GET( )

Purpose:

This function returns the text corresponding to an error number.

Context:

1. At any place in the program.

Syntax:

```
CALL ERR_GET( error-number INTEGER ) RETURNING result STRING
```

Notes:

1. *error-number* is a runtime error or an Informix SQL error.
2. For development only.

Warnings:

1. Informix SQL error numbers can only be supported if the program is connected to an Informix database.

See also: ERRORLOG(), STARTLOG(), ERR_QUIT(), ERR_PRINT(), Exceptions.

ERR_PRINT( )

Purpose:

This function prints in the error line the text corresponding to an error number.

Context:

1. At any place in the program.

Syntax:

```
CALL ERR_PRINT( error-number INTEGER )
```

Notes:

1. *error-number* is a runtime error or an Informix SQL error.
2. For development only.

Warnings:

1. Informix SQL error numbers can only be supported if the program is connected to an Informix database.

See also: ERRORLOG(), STARTLOG(), ERR_QUIT(), ERR_GET(), Exceptions.

ERR_QUIT( )

Purpose:

This function prints in the error line the text corresponding to an error number and terminates the program.

Context:

1. At any place in the program.

Syntax:

```
CALL ERR_QUIT( error-number INTEGER )
```

Notes:

1. *error-number* is a runtime error or an Informix SQL error.
2. For development only.

Warnings:

1. Informix SQL error numbers can only be supported if the program is connected to an Informix database.

See also: ERRORLOG(), STARTLOG(), ERR_QUIT(), ERR_GET(), Exceptions.

ERRORLOG( )

Purpose:

This function copies the string passed as parameter into the error log file.

Context:

1. At any place in the program.

Syntax:

```
CALL ERRORLOG( text STRING )
```

Notes:

1. *text* is the character string to be inserted in the error log file.
2. The error log must be started with STARTLOG().

See also: STARTLOG(), Exceptions.

SHOWHELP( )

Purpose:

This function function displays a runtime help message, corresponding to its specified argument, from the current help file.

Context:

1. At any place in the program.

Syntax:

```
CALL SHOWHELP( help-number INTEGER )
```

Notes:

1. *help-number* is the help message number in the current help file.
2. You set the current help file with the `HELP FILE` clause of the OPTIONS instruction.

See also: OPTIONS, Message Files.

STARTLOG( )

Purpose:

This function initializes error logging and opens the error log file passed as the parameter.

Context:

1. At the beginning of the program.

Syntax:

```
CALL STARTLOG( filename STRING )
```

Notes:

1. *filename* is the name of the error log file.
2. Runtime errors are automatically reported.
3. You can insert error records manually with the ERRORLOG() function.

Usage:

Call `STARTLOG()` in the MAIN program block to open or create an error log file. After `STARTLOG()` has been invoked, a record of every subsequent error that occurs during the execution of your program is written in the error log file. If you need to report specific application errors, use the ERRORLOG() function to make an entry in the error log file.

The default format of an error record consists of the date, time, source-module name and line number, error number, and error message. If you invoke the `STARTLOG()` function, the format of the error records appended to the error log file after each subsequent error are as follows:

```
Date: 03/06/99 Time: 12:20:20
Program error at "stock_one.4gl", line number 89.
SQL statement error number -239.
Could not insert new row - duplicate value in a UNIQUE INDEX column.
SYSTEM error number -100
ISAM error: duplicate value for a record with unique key.
```

The `STARTLOG()` and ERRORLOG() functions can be used for *instrumenting* a program, to track how the program is used. This use is not only valuable for improving the program but also for recording work habits and detecting attempts to breach security.

If the argument of `STARTLOG()` is not the name of an existing file, `STARTLOG()` creates one. If the file already exists, `STARTLOG()` opens it and positions the file pointer so that subsequent error messages can be appended to this file. The following program fragment invokes `STARTLOG()`, specifying the name of the error log file in a quoted string that includes a pathname and a file extension. The function definition includes a call to the built-in ERRORLOG() function, which adds a message to the error log file.

See also: ERRORLOG(), Exceptions.

FGL_BUFFERTOUCHED( )

Purpose:

This function returns TRUE if the input buffer was modified after the current field was selected.

Context:

1. In `AFTER FIELD`, `AFTER INPUT`, `AFTER CONSTRUCT`, `ON KEY`, `ON ACTION` blocks.

Syntax:

```
CALL FGL_BUFFERTOUCHED( ) RETURNING result INTEGER
```

Notes:

1. returns TRUE if the input buffer has been touched after the current field was selected.

Warnings:

1. This function is not equivalent to FIELD_TOUCHED(): The flag returned by `FGL_BUFFERTOUCHED()` is reset when you enter a new field, while `FIELD_TOUCHED()` returns always TRUE for a field that was modified during the interactive instruction.

See also: FGL_DIALOG_SETBUFFER(), FGL_DIALOG_GETBUFFER().

FGL_DIALOG_GETBUFFER( )

Purpose:

This function returns the value of the current field as a string.

Context:

1. In INPUT , INPUT ARRAY, CONSTRUCT instructions.

Syntax:

```
CALL FGL_DIALOG_GETBUFFER( ) RETURNING result STRING
```

Notes:

1. Returns the content of the input buffer of the current field.

2. Only useful in a CONSTRUCT instruction, because there is no variable associated to fields in this case.

See also: FGL_DIALOG_SETBUFFER(), FGL_BUFFERTOUCHED(), GET_FLDBUF().

FGL_DIALOG_SETBUFFER( )

Purpose:

This function sets the input buffer of the current field, and assigns corresponding program variable when using `UNBUFFERED` mode.

Context:

1. In INPUT , INPUT ARRAY, CONSTRUCT instructions.

Syntax:

```
CALL FGL_DIALOG_SETBUFFER( value STRING )
```

Notes:

1. *value* is the text to set in the current input buffer.
2. Only useful in a CONSTRUCT instruction, because there is no variable associated to fields in this case.

Warnings:

1. With the default *buffered* input mode, this function modifies the input buffer of the current field; the corresponding input variable is not assigned. It makes no sense to call this function in `BEFORE FIELD` blocks of `INPUT` and `INPUT ARRAY`. However, if the statement is using the `UNBUFFERED` mode, the function will set both the field buffer and the program variable. If the string set by the function does not represent a valid value that can be stored by the program variable, the buffer and the variable will be set to NULL.
2. This function sets the 'touched' flag of the current form field, and the 'touched' flag of the dialog. Therefore, both FIELD_TOUCHED() and FGL_BUFFERTOUCHED() would return TRUE if you call this function.

See also: FGL_DIALOG_GETBUFFER(), FGL_BUFFERTOUCHED(), GET_FLDBUF().

FGL_DIALOG_GETFIELDNAME( )

Purpose:

This function returns the name of the current input field.

Context:

1. In INPUT , INPUT ARRAY, CONSTRUCT or DISPLAY ARRAY instructions.

Syntax:

```
CALL FGL_DIALOG_GETFIELDNAME( ) RETURNING result STRING
```

Notes:

1. Returns the name of the current input field.

Warnings:

1. Only the column part of the field name is returned.

See also: FGL_DIALOG_INFIELD().

FGL_DIALOG_INFIELD( )

Purpose:

This function returns TRUE if the field passed as the parameter is the current input field.

Context:

1. In INPUT , INPUT ARRAY, CONSTRUCT or DISPLAY ARRAY instructions.

Syntax:

```
CALL FGL_DIALOG_INFIELD( field-name STRING ) RETURNING result INTEGER
```

Notes:

1. *field-name* is the name if the form field.

Warnings:

1. Only the column part of the field name is used.

See also: INFIELD().

FGL_DIALOG_SETCURSOR( )

Purpose:

This function sets the position of the input cursor in the current field.

Context:

1. In interactive instructions.

Syntax:

```
CALL FGL_DIALOG_SETCURSOR( index INTEGER )
```

Notes:

1. *index* is the character position in the text.

See also: FGL_GETCURSOR().

FGL_DIALOG_SETFIELDORDER( )

Purpose:

This function enables or disables field order constraint.

Context:

1. At the beginning of the program or around INPUT instructions.

Syntax:

```
CALL FGL_DIALOG_SETFIELDORDER( active INTEGER )
```

Notes:

1. When *active* is TRUE, the field order is constrained.
2. When *active* is FALSE, the field order is not constrained.

Usage:

Typical BDL applications control user input with `BEFORE FIELD` and `AFTER FIELD` blocks. In many cases the field order and the sequential execution of `AFTER FIELD` blocks is important in order to validate the data entered by the user. But with graphical front ends you can use the mouse to move to a field. By default the runtime system executes all `BEFORE FIELD` and `AFTER FIELD` blocks of the fields used by the interactive instruction, from the origin field to the target field selected by mouse click. If needed, you can force the runtime system to ignore all intermediate field triggers, by calling this function with a FALSE attribute.

FGL_DIALOG_SETCURRLINE( )

Purpose:

This function moves to a specific row in a record list.

Context:

1. During a DISPLAY ARRAY or INPUT ARRAY instruction.

Syntax:

```
CALL FGL_DIALOG_SETCURRLINE( line INTEGER, row INTEGER )
```

Notes:

1. *line* is the line number in the screen array.
2. *row* is the row number is the array variable.

Warnings:

1. The *line* parameter is ignored in GUI mode.
2. You can use the FGL_SET_ARR_CURR() function instead.
3. Control blocks like `BEFORE ROW` and field/row validation in `INPUT ARRAY` are performed, as if the user moved to another row, except when the function is called in `BEFORE DISPLAY` or `BEFORE INPUT`.

FGL_SET_ARR_CURR( )

Purpose:

This function moves to a specific row in a record list.

Context:

1. During a DISPLAY ARRAY or INPUT ARRAY instruction.

Syntax:

```
CALL FGL_SET_ARR_CURR( row INTEGER )
```

Notes:

1. *row* is the row number is the array variable.

Usage:

This function is typically used to control navigation in a `DISPLAY ARRAY` or `INPUT ARRAY`.

When a new row is reaching by using with this function, the first field editable gets the focus.

Warning: Control blocks like `BEFORE ROW` and field/row validation in `INPUT ARRAY` are performed, as if the user moved to another row, except when the function is called in `BEFORE DISPLAY` or `BEFORE INPUT`.

FGL_SETENV( )

Purpose:

This function sets the value of an environment variable.

Context:

1. At any place in the program.

Syntax:

```
CALL FGL_SETENV( variable STRING, value STRING )
```

Notes:

1. *variable* is the name of the environment variable.
2. *value* is the value to be set.

Warnings:

1. Use this function at your own risk: You may experience unexpected results if you change environment variables that are already used by the current program - for

example, when you are connected to INFORMIX and you change the INFORMIXDIR environment variable.
2. There is a little difference between Windows and UNIX platforms when passing a NULL as the *value* parameter: On Windows, the environment variable is removed, while on UNIX, the environment variable gets an empty value (i.e. it is not removed from the environment).

See also: FGL_GETENV()

FGL_DRAWBOX( )

Purpose:

This function draws a rectangle based on the character terminal coordinates in the current open window.

Context:

1. At any place in the program.

Syntax:

```
CALL FGL_DRAWBOX( height, width, line, column, color INTEGER )
```

Warnings:

1. This function is provided for backward compatibility.

See also: FGL_DRAWLINE().

FGL_DRAWLINE( )

Purpose:

This function draws a line based on the character terminal coordinates in the current open window.

Context:

1. At any place in the program.

Syntax:

```
CALL FGL_DRAWLINE( posX, posY, width, color INTEGER )
```

Warnings:

1. This function is provided for backward compatibility.

See also: FGL_DRAWBOX().

FGL_LASTKEY( )

Purpose:

This function returns the key code of the last key pressed by the user.

Context:

1. Any interactive instruction.

Syntax:

```
CALL FGL_LASTKEY( ) RETURNING result INTEGER
```

Notes:

1. The function returns NULL if no key has been pressed.

Warnings:

1. This function is provided for backward compatibility.

See also: FGL_KEYVAL().

FGL_KEYVAL( )

Purpose:

This function returns the key code of a logical or physical key.

Context:

1. At any place in the program.

Syntax:

```
CALL FGL_KEYVAL( character STRING ) RETURNING result INTEGER
```

Notes:

1. *character* can be a single character, a digit, a printable symbol like @, #, $ or a special keyword.
2. Keywords recognized by `FGL_KEYVAL()` are: `ACCEPT`, `HELP`, `NEXT`, `RETURN`, `DELETE`, `INSERT`, `NEXTPAGE`, `RIGHT`, `DOWN`, `INTERRUPT`, `PREVIOUS`, `TAB`, `ESC`, `ESCAPE`, `LEFT`, `PREVPAGE`, `UP`, `F1` through `F64`, `CONTROL-`*`character`* (except A, D, H, I, J, L, M, R, or X)
3. If you specify a single character, `FGL_KEYVAL()` considers the case. In all other instances, the function ignores the case of its argument, which can be uppercase or lowercase letters.
4. The function returns NULL if the parameter does not correspond to a valid key.

Warnings:

1. This function is provided for backward compatibility especially for TUI mode applications. `FGL_KEYVAL()` is well supported in text mode, but this function can only be emulated in GUI mode, because the front-ends communicate with the runtime system with other events as keystrokes.

Usage:

`FGL_KEYVAL()` can be used in form-related statements to examine a value returned by the FGL_LASTKEY() function.

To determine whether the user has performed an action, such as inserting a row, specify the logical name of the action (such as INSERT) rather than the name of the physical key (such as F1). For example, the logical name of the default Accept key is ESCAPE. To test if the key most recently pressed by the user was the Accept key, specify `FGL_KEYVAL("ACCEPT")` rather than `FGL_KEYVAL("escape")` or `FGL_KEYVAL("ESC")`. Otherwise, if a key other than ESCAPE is set as the Accept key and the user presses that key, `FGL_LASTKEY()` does not return a code equal to `FGL_KEYVAL("ESCAPE")`. The value returned by `FGL_LASTKEY()` is undefined in a MENU statement.

See also: FGL_LASTKEY().

FGL_REPORT_PRINT_BINARY_FILE( )

Purpose:

This function prints a file containing binary data during a report.

Context:

1. In a REPORT routine.

Syntax:

```
CALL FGL_REPORT_PRINT_BINARY_FILE( filename STRING )
```

Notes:

1. *filename* is the name of the binary file.

Warnings:

1. This function is provided for backward compatibility.

FGL_REPORT_SET_DOCUMENT_HANDLER( )

Purpose:

This function redirects the next report to an XML document handler.

Context:

1. Before / After the execution of a REPORT.

Syntax:

```
CALL FGL_REPORT_SET_DOCUMENT_HANDLER( handler om.SaxDocumentHandler )
```

Notes:

1. *handler* is the document handler variable.

FGL_GETCURSOR( ) / FGL_DIALOG_GETCURSOR()

Purpose:

This function returns the position of the edit cursor in the current field.

Context:

1. In interactive instructions.

Syntax:

```
CALL FGL_GETCURSOR( ) RETURNING index INTEGER
```

Notes:

1. index is the character position in the text.

See also: FGL_DIALOG_SETCURSOR().

FGL_GETWIN_HEIGHT( )

Purpose:

This function returns the number of rows of the current window.

Context:

1. At any place in the program.

Syntax:

```
CALL FGL_GETWIN_HEIGHT( ) RETURNING result INTEGER
```

Warnings:

1. This function is provided for backward compatibility.

See also: FGL_GETWIN_WIDTH().

FGL_GETWIN_WIDTH( )

Purpose:

This function returns the width of the current window as a number of columns.

Context:

1. At any place in the program.

Syntax:

```
CALL FGL_GETWIN_WIDTH( ) RETURNING result INTEGER
```

Warnings:

1. This function is provided for backward compatibility.

See also: FGL_GETWIN_WIDTH().

FGL_GETWIN_X( )

Purpose:

This function returns the horizontal position of the current window.

Context:

1. At any place in the program.

Syntax:

```
CALL FGL_GETWIN_X( ) RETURNING result INTEGER
```

Warnings:

1. This function is provided for backward compatibility.

See also: FGL_GETWIN_Y().

FGL_GETWIN_Y( )

Purpose:

This function returns the vertical position of the current window.

Context:

1. At any place in the program.

Syntax:

```
CALL FGL_GETWIN_X( ) RETURNING result INTEGER
```

Warnings:

1. This function is provided for backward compatibility.

See also: FGL_GETWIN_X().

LENGTH( )

Purpose:

This function returns the number of bytes of the expression passed as parameter.

Context:

1. At any place in the program.

Syntax:

```
CALL LENGTH( expression ) RETURNING result INTEGER
```

Notes:

1. *expression* is any valid expression.
2. Trailing blanks are not counted in the length of the string.
3. If the parameter is NULL, the function returns zero.

Warnings:

1. The function counts bytes, not characters. This is important in a multi-byte environment.
2. Most database servers support an equivalent scalar function in the SQL language, but the result may be different from the FGL built-in function. For example, Oracle's LENGTH() function returns NULL when the string is empty.

See also: FGL_WIDTH().

FGL_GETVERSION( )

Purpose:

This function returns the build number of the runtime system.

Context:

1. At any place in the program.

Syntax:

```
CALL FGL_GETVERSION( ) RETURNING result STRING
```

Warnings:

1. Provided for debugging info only; please do not write business code dependent on the build number.
2. The format of the build number returned by this function is subject of change in future versions.

FGL_GETHELP( )

Purpose:

Returns the help text according to its identifier by reading the current help file.

Context:

1. At any place in the program, after the definition of the current help file (OPTIONS HELP FILE).

Syntax:

```
CALL FGL_GETHELP( id INTEGER ) RETURNING result STRING
```

Notes:

1. *id* is the help text identifier.

See also: The OPTIONS instruction.

FGL_GETPID( )

Purpose:

This function returns the system process identifier.

Context:

1. At any place in the program.

Syntax:

```
CALL FGL_GETPID() RETURNING result INTEGER
```

Notes:

1. The process identifier is provided by the operating system; it is normally unique.

See also: FGL_SYSTEM().

FGL_DIALOG_GETBUFFERSTART( )

Purpose:

This function returns the row offset of the page to feed a paged display array.

Syntax:

```
CALL FGL_DIALOG_GETBUFFERSTART( ) RETURNING result INTEGER
```

Usage:

See DISPLAY ARRAY.

FGL_DIALOG_GETBUFFERLENGTH( )

Purpose:

This function returns the number of rows of the page to feed a paged display array.

Syntax:

```
CALL FGL_DIALOG_GETBUFFERLENGTH( ) RETURNING result INTEGER
```

Usage:

See DISPLAY ARRAY.

FGL_PUTFILE

Purpose:

Transfers a file from the application server machine to the front end workstation.

Syntax:

```
CALL fgl_putfile(src STRING, dst STRING)
```

Notes:

1. *src* contains the name of the source file to send.
2. *dst* contains the name of the file to write on the front end.

FGL_GETFILE

Purpose:

Transfers a file from the front end workstation to the application server machine.

Syntax:

```
CALL fgl_getfile(src STRING, dst STRING)
```

Notes:

1. *src* contains the name of the source file to retrieve from the front end workstation.
2. *dst* contains the name of the file to write on the server side.

FGL_GETENV( )

Purpose:

This function returns the value of the environment variable having the name you specify as the argument.

Syntax:

```
CALL FGL_GETENV( variable STRING ) RETURNING result STRING
```

Notes:

1. *variable* is the name of the environment variable.
2. If the specified environment variable is not defined, the function returns a NULL value.
3. If the environment variable is defined but does not have a value assigned to it, the function returns blank spaces.

Warnings:

1. If the returned value can be a long character string, be sure to declare the receiving variable with sufficient size to store the character value returned by the function. Otherwise, the value will be truncated.

Usage:

The argument of `FGL_GETENV()` must be the name of an environment variable. If the requested value exists in the current user environment, the function returns it as a character string and then returns control of execution to the calling context.

See also: FGL_SETENV()

FGL_GETKEYLABEL( )

Purpose:

This function returns the default label associated to a key.

Syntax:

```
CALL FGL_GETKEYLABEL( keyname STRING ) RETURNING result STRING
```

Notes:

1. *keyname* is the logical name of a key such as `F11` or `DELETE`, `INSERT`, `CANCEL`.

Warnings:

1. This function is provided for backward compatibility.

See also: FGL_SETKEYLABEL(), FGL_DIALOG_GETKEYLABEL().

FGL_SETKEYLABEL( )

Purpose:

This function sets the default label associated to a key.

Syntax:

```
CALL FGL_SETKEYLABEL( keyname STRING, label STRING )
```

Notes:

1. *keyname* is the logical name of a key such as `F11` or `DELETE`, `INSERT`, `CANCEL`.
2. *label* is the text associated to the key.

Warnings:

1. This function is provided for backward compatibility.

See also: FGL_SETKEYLABEL(), FGL_DIALOG_SETKEYLABEL().

FGL_DIALOG_GETKEYLABEL( )

Purpose:

This function returns the label associated to a key for the current interactive instruction.

Syntax:

```
CALL FGL_DIALOG_GETKEYLABEL( keyname STRING ) RETURNING result STRING
```

Notes:

1. *keyname* is the logical name of a key such as `F11` or `DELETE`, `INSERT`, `CANCEL`.

Warnings:

1. This function is provided for backward compatibility.

See also: FGL_SETKEYLABEL(), FGL_DIALOG_SETKEYLABEL().

FGL_DIALOG_SETKEYLABEL( )

Purpose:

This function sets the label associated to a key for the current interactive instruction.

Syntax:

```
CALL FGL_DIALOG_SETKEYLABEL( keyname STRING, label STRING )
```

Notes:

1. *keyname* is the logical name of a key such as `F11` or `DELETE`, `INSERT`, `CANCEL`.

2. *label* is the text associated to the key.

Warnings:

1. This function is provided for backward compatibility.

See also: FGL_SETKEYLABEL(), FGL_DIALOG_GETKEYLABEL().

FGL_SETSIZE( )

Purpose:

This function sets the size of the main application window.

Syntax:

```
CALL FGL_SETSIZE( width INTEGER, height INTEGER )
```

Notes:

1. *width* is the number of columns of the window.
2. *height* is the number of lines of the window.

Warnings:

1. This function is provided for backward compatibility.

See also: FGL_SETTITLE().

FGL_SETTITLE( )

Purpose:

This function sets the title of the main application window.

Syntax:

```
CALL FGL_SETTITLE( label STRING )
```

Notes:

1. *label* is the text of the title.

Warnings:

1. This function is provided for backward compatibility.

See also: FGL_SETSIZE().

FGL_SYSTEM( )

Purpose:

This function starts a program in a UNIX terminal emulator when using a graphical front end.

Syntax:

```
CALL FGL_SYSTEM( command STRING )
```

Notes:

1. *command* is the command line to be executed on the server.

Warnings:

1. This function is provided for backward compatibility.

Usage:

The function starts a program that needs a UNIX terminal emulator when using the Windows Front End, even if the current program has been started without a visible terminal. The *command* parameter is a string or variable that contains the commands to be executed. The UNIX terminal will be raised and activated. The terminal is automatically lowered when the child process finishes.

See also: WINEXEC().

FGL_WIDTH()

Purpose:

This function returns the number of columns needed to represent the printed version of the expression.

Context:

1. At any place in the program.

Syntax:

```
CALL FGL_WIDTH( expression ) RETURNING result INTEGER
```

Notes:

1. *expression* is any valid expression.
2. Trailing blanks are counted in the length of the string.
3. If the parameter is NULL, the function returns zero.

See also: LENGTH().

FGL_WINDOW_GETOPTION( )

Purpose:

This function returns attributes of the current window.

Syntax:

```
CALL FGL_WINDOW_GETOPTION( attribute STRING ) RETURNING result STRING
```

Notes:

1. *attribute* is the name of a window attribute. This can be one of `name`, `x`, `y`, `width`, `height`, `formline`, `messageline`.

Warnings:

1. This function is provided for backward compatibility.

FGL_GETRESOURCE( )

Purpose:

This function returns the value of an FGLPROFILE entry.

Context:

1. At any place in the program.

Syntax:

```
CALL FGL_GETRESOURCE( name STRING ) RETURNING result STRING
```

Notes:

1. *name* is the FGLPROFILE entry name to be read.
2. If the entry does not exist in the configuration file, the function returns NULL.
3. See also FGLPROFILE definition.

Warnings:

1. FGLPROFILE entry names are not case sensitive.

ORD( )

Purpose:

This function accepts as its argument a character expression and returns the integer value of the first byte of that argument.

Context:

1. At any place in the program.

Syntax:

```
CALL ORD( source STRING ) RETURNING result INTEGER
```

Notes:

1. *source* is a string expression.
2. This function is case-sensitive.
3. Only the first byte of the argument is evaluated.
4. Returns NULL if the argument passed is not valid.
5. For a default (U.S. English) locale, this function is the logical inverse of the ASCII() operator.

See also: FGL_KEYVAL(), ASCII().

DOWNSHIFT( )

Purpose:

This function returns returns a string value in which all uppercase characters in its argument are converted to lowercase.

Context:

1. At any place in the program.

Syntax:

```
CALL DOWNSHIFT( source STRING ) RETURNING result STRING
```

Notes:

1. *source* is the character string to convert to lowercase letters.
2. Non-alphabetic or lowercase characters are not altered.

Warnings:

1. Conversion depends on locale settings (the LC_CTYPE environment variable).

See also: UPSHIFT().

UPSHIFT( )

Purpose:

This function returns a string value in which all lowercase characters in its argument are converted to uppercase.

Context:

1. At any place in the program.

Syntax:

```
CALL UPSHIFT( source STRING ) RETURNING result STRING
```

Notes:

1. *source* is the character string to convert to uppercase letters.
2. Non-alphabetic or uppercase characters are not altered.

Warnings:

1. Conversion depends on locale settings (the LC_CTYPE environment variable).

See also: DOWNSHIFT().

The key code table

Warning: These are internal key codes. Avoid hardcoding these numbers in your sources; otherwise your 4gl source will not be compatible with future versions of Genero FGL. Always use the FGL_KEYVAL(*keyname*) function instead.

Value	Key name	Description
`1 to 26`	`Control-`*`x`*	Control key, where *x* is the any letter from A to Z. The key code corresponding to Control-A is 1, Control-B is 2, etc.
`others < 256`	`ASCII chars`	Other codes correspond to the ASCII characters set.
`2000`	`up`	The up-arrow logical key.
`2001`	`down`	The down-arrow logical key.
`2002`	`left`	The left-arrow logical key.
`2003`	`right`	The right-arrow logical key.
`2005`	`nextpage`	The next-page logical key.
`2006`	`prevpage`	The previous-page logical key.
`2008`	`help`	The help logical key.
`2011`	`interrupt`	The interrupt logical key.
`2012`	`home`	The home logical key.
`2013`	`end`	The end logical key.
`2016`	`accept`	The accept logical key.
`2017`	`backspace`	The backspace logical key.
`3000 to 3255`	`F`*`x`*	Function key, where *x* is the number of the function key. The key code corresponding to a function key F*x* is 3000+*x*-1, for example, 3011 corresponds to F12.

Utility Functions

Summary:

- What is a utility function?
- List of utility functions
- List of de-supported utility functions

See also: Built-in Functions.

What is a utility function?

A utility function is a function provided in a separate 4GL library; it is not built in the runtime system. You must link with the utility library to use one of the utility functions.

The library of utility function is named **libfgl4js**. The 42x file, 42m modules and 42f forms are provided in $FGLDIR/lib. The sources of the utility functions and form files are provided in the FGLDIR/src directory.

List of utility functions

Function	Description
Common dialog functions	
FGL_WINBUTTON()	In a separate window, displays an interactive message box with multiple choices
FGL_WINMESSAGE()	In a separate window, Displays an interactive message box with some text
FGL_WINPROMPT()	Displays a dialog box with a field that accepts a value
FGL_WINQUESTION()	In a separate window, displays an interactive message box with Yes, No, Cancel buttons
FGL_WINWAIT()	Displays a dialog box and waits for the user to press a key
Database utility functions	
DB_GET_DATABASE_TYPE()	Returns the type of the current database
DB_GET_SEQUENCE()	Returns a new serial value from a predefined table (SERIALREG)
DB_START_TRANSACTION()	Starts a new transaction if none is started (for nested transaction handling)
DB_FINISH_TRANSACTION()	Ends a nested transaction

DB_IS_TRANSACTION_STARTED()	Returns TRUE if a nested transaction is started
Front End Functions (Use ui.Interface.frontCall() instead)	
WINOPENDIR()	Shows a dialog window to select a directory in the front end workstation file system; use ui.Interface.frontCall("standard","opendir",...) instead.
WINOPENFILE()	Shows a dialog window to select a file in the front end workstation file system; use ui.Interface.frontCall("standard","openfile",...) instead.
WINSAVEFILE()	Shows a dialog window to save a file in the front end workstation file system; use ui.Interface.frontCall("standard","savefile",...) instead.
Microsoft Windows Client Specific Functions (Use ui.Interface.frontCall() instead)	
WINEXEC()	Starts a program on a Microsoft Windows front end without waiting; use ui.Interface.frontCall("standard","execute",...) instead.
WINEXECWAIT()	Starts a program on a Microsoft Windows front end and waits; use ui.Interface.frontCall("standard","execute",...) instead.
WINSHELLEXEC()	Opens a document on a Microsoft Windows front end with the corresponding program; use ui.Interface.frontCall("standard","shellexec",...) instead.

List of de-supported utility functions:

Function	Description
DATE1()	Converts a DATETIME to a DATE.
TIME1( )	Extracts the time part (hour, minute, second) from DATETIME.
FGL_FGLGUI()	Returns TRUE if the application runs in GUI

	mode.
FGL_GETUITYPE()	Returns the type of the front end.
FGL_MSG_NONL()	Returns an error message without trailing blanks.
FGL_INIT4JS( )	Initializes the built-in function library.
FGL_MSG_NONL( )	Returns an error message without the CR at the end.
FGL_WTKCLIENT( )	Returns TRUE if the current front end is the WTK.
FGL_RESOURCE( )	Selects a specific FGLPROFILE file.
FGL_UIRETRIEVE( )	Returns the value of a variable from the WTK front end.
FGL_UISENDCOMMAND( )	Sends a TCL command to the WTK front end.
FGL_WTKCLIENT( )	Returns TRUE if the current front end is the WTK.
FGL_CHARBOOL_TO_INTBOOL()	Converts a character representation of a Boolean value to an INTEGER.
FGL_INTBOOL_TO_CHARBOOL( )	Converts an INTEGER to a character representation of the Boolean value.

DB_GET_DATABASE_TYPE( )

Purpose:

This function returns the database type for the current connection.

Syntax:

```
CALL DB_GET_DATABASE_TYPE()
 RETURNING result STRING
```

Usage:

After connecting to the database, you can get the type of the database server with this function.

The following table shows the codes returned by this function, for the supported database types:

Code	Description
ADS	ANTs / Genero DB
ASA	Sybase ASA

DB2	IBM DB2
IFX	Informix
MYS	MySQL
MSV	Microsoft SQL Server
ORA	Oracle
PGS	PostgreSQL

DB_GET_SEQUENCE( )

Purpose:

This function generates a new sequence for a given identifier.

Syntax:

```
CALL DB_GET_SEQUENCE( id STRING )
 RETURNING result INTEGER
```

Warnings:

1. This function needs a database table called SERIALREG.
2. This function must be used **inside** a transaction block.

Usage:

This function generates a new sequence from a register table created in the current database.

The table must be created as follows:

```
CREATE TABLE SERIALREG
 ( TABLE_NAME VARCHAR(50) NOT NULL PRIMARY KEY,
   LAST_SERIAL INTEGER NOT NULL )
```

Each time you call this function, the sequence is incremented in the database table and returned by the function.

It is mandatory to use this function inside a transaction block, in order to generate unique sequences.

Example:

```
01 MAIN
02   DEFINE ns, s INTEGER
03   DATABASE mydb
04   LET s = DB_START_TRANSACTION()
```

```
05   LET ns = DB_GET_SEQUENCE("mytable")
06   INSERT INTO mytable VALUES ( ns, 'a new sequence' )
07   LET s = DB_FINISH_TRANSACTION(TRUE)
08 END MAIN
```

DB_START_TRANSACTION( )

Purpose:

This function encapsulates the BEGIN WORK instruction to start a transaction.

Syntax:

```
CALL DB_START_TRANSACTION()
 RETURNING result INTEGER
```

Usage:

You can use the transaction management functions to handle nested transactions.

On most database engines, you can only have a unique transaction, that you start with BEGIN WORK and you end with COMMIT WORK or ROLLBACK WORK. But in a complex program, you may have nested function calls doing several SQL transactions.

The transaction management functions execute a real transaction instruction only if the number of subsequent start/end calls of these functions matches.

Example:

```
01 DEFINE s INTEGER
02
03 MAIN
04   DATABASE mydb
05   LET s = DB_START_TRANSACTION() -- real BEGIN WORK
06   CALL do_update()
07   LET s = DB_FINISH_TRANSACTION(TRUE) -- real COMMIT WORK
08 END MAIN
09
10 FUNCTION do_update()
11   LET s = DB_START_TRANSACTION()
12   UPDATE customer SET cust_status = 'X'
13   LET s = DB_FINISH_TRANSACTION(TRUE)
14 END FUNCTION
```

DB_FINISH_TRANSACTION( )

Purpose:

This function encapsulates the COMMIT WORK or ROLLBACK WORK instructions to end a transaction.

Syntax:

```
CALL DB_FINISH_TRANSACTION( commit INTEGER )
 RETURNING result INTEGER
```

Notes:

1. *commit* indicates whether the transaction must be committed.

Usage:

When the number of calls to DB_START_TRANSACTION() matches, this function executes a COMMIT WORK if the passed parameter is TRUE; if the passed parameter is not TRUE, it executes a ROLLBACK WORK.

If the number of calls does not match, the function does nothing.

See also: DB_START_TRANSACTION().

DB_IS_TRANSACTION_STARTED( )

Purpose:

This function indicates whether a transaction is started with the transaction management functions.

Syntax:

```
CALL DB_IS_TRANSACTION_STARTED()
 RETURNING result INTEGER
```

Usage:

The function returns TRUE if a transaction was started with DB_START_TRANSACTION().

FGL_WINBUTTON( )

Purpose:

This function displays an interactive message box containing multiple choices, in a separate window.

Syntax:

```
CALL FGL_WINBUTTON(
  title STRING, text STRING, default STRING,
  buttons STRING, icon STRING, danger SMALLINT )
 RETURNING result STRING
```

Notes:

1. *title* defines the title of the message window.
2. *text* specifies the string displayed in message window.
3. Use '\n' in *text* to separate lines (this does not work on ASCII client).
4. *default* indicates the default button to be pre-selected.
5. *buttons* defines a set of button labels separated by "|".
6. You can define up to 7 buttons that each have 10 characters.
7. *icon* is the name of the icon to be displayed.
8. Supported icon names are: "information", "exclamation", "question", "stop".
9. *danger* (for X11 only), number of the warnings item. Otherwise, this parameter is ignored.
10. The function returns the label of the button which has been selected by the user.

Warnings:

1. You can also use a form or a menu with "popup" style instead.
2. If two buttons start with the same letter, the user will not be able to select one of them on the ASCII client.
3. The "&" before a letter for a button is either displayed (ASCII client), or it underlines the letter.

Example:

```
01 MAIN
02   DEFINE answer STRING
03   LET answer = FGL_WINBUTTON( "Media selection", "What is your
favorite media?",
04       "Lynx", "Floppy Disk|CD-ROM|DVD-ROM|Other", "question", 0)
05   DISPLAY "Selected media is: " || answer
06 END MAIN
```

FGL_WINMESSAGE( )

Purpose:

This function displays an interactive message box containing text, in a separate window.

Syntax:

```
CALL FGL_WINMESSAGE( title STRING, text STRING, icon STRING )
```

Notes:

1. *title* defines message box title.
2. *text* is the text displayed in the message box. Use '\n' to separate lines.
3. *icon* is the name of the icon to be displayed.
4. Supported icon names are: "information", "exclamation", "question", "stop".

Warnings:

1. You can also use a form or a menu with "popup" style instead.
2. *icon* is ignored by the ASCII client.

Example:

```
01 MAIN
02   CALL FGL_WINMESSAGE( "Title", "This is a critical message.",
"stop")
03 END MAIN
```

FGL_WINPROMPT( )

Purpose:

This function displays a dialog box containing a field that accepts a value.

Syntax:

```
CALL FGL_WINPROMPT (
  x INTEGER, y INTEGER, text STRING,
  default STRING, length INTEGER, type STRING )
 RETURNING value STRING
```

Notes:

1. *x* is the column position in characters.
2. *y* is the line position in characters.
3. *text* is the message shown in the box.

4. *default* is the default value.
5. *length* is the maximum length of the input value.
6. *type* is the datatype of the return value : 0=CHAR, 1=SMALLINT, 2=INTEGER, 7=DATE, 255=invisible
7. *value* is the value entered by the user.

Warnings:

1. You can also use your own form instead.
2. Avoid passing NULL values.

Example:

```
01 MAIN
02   DEFINE answer DATE
04   LET answer = FGL_WINPROMPT( 10, 10, "Today", DATE, 10, 7 )
05   DISPLAY "Today is " || answer
06 END MAIN
```

FGL_WINQUESTION( )

Purpose:

This function displays an interactive message box containing Yes, No, and Cancel buttons, in a separate window

Syntax:

```
CALL FGL_WINQUESTION(
  title STRING, text STRING, default STRING,
  buttons STRING, icon STRING, danger SMALLINT )
 RETURNING value STRING
```

Notes:

1. *title* is the message box title.
2. *text* is the message displayed in the message box. Use '\n' to separate lines (does not work on ASCII client).
3. *default* defines the default button that is pre-selected.
4. *buttons* defines the buttons: Either "yes|no" or "yes|no|cancel", not case-sensitive.
5. *icon* is the name of the icon to be displayed.
6. Supported icon names are: "`information`", "`exclamation`", "`question`", "`stop`".
7. *danger* is for X11, it defines the code of the warning item. Otherwise, this parameter is ignored.
8. The function returns the label of the button which has been selected by the user.

Warnings:

1. You can also use a form or a menu with "popup" style instead.
2. Setting *buttons* to another value may result in unpredictable behavior at runtime.
3. Avoid passing NULL values

Example:

```
01 MAIN
02   DEFINE answer STRING
04   LET answer = "yes"
05   WHILE answer = "yes"
06     LET answer = FGL_WINQUESTION(
07         "Procedure", "Would you like to continue ? ",
08         "cancel", "yes|no|cancel", "question", 0)
09   END WHILE
10   IF answer = "cancel" THEN
11      DISPLAY "Canceled."
12   END IF
13 END MAIN
```

FGL_WINWAIT( )

Purpose:

This function displays an interactive message box and waits for the user to press a key

Syntax:

```
CALL FGL_WINWAIT( text STRING )
```

Notes:

1. *text* is the message displayed in the message box. Use '\n' to separate lines (not working on ASCII client).

Warnings:

1. You can also use a form or a menu with "popup" style instead.

WINEXEC( ) MS Windows FE Only!

Purpose:

This function executes a program on the machine where the Windows Front End runs and returns immediately.

Context:

1. At any place in the program, but only after the first instruction has displayed something on the front end.

Syntax:

```
CALL WINEXEC( command STRING )
 RETURNING result INTEGER
```

Notes:

1. *command* is the command to be executed on the front end.
2. The function executes the program without waiting.
3. The function returns FALSE if a problem has occurred.

See also: WINEXECWAIT(), DDE Support.

WINEXECWAIT( ) MS Windows FE Only!

Purpose:

This function executes a program on the machine where the Windows Front End runs and waits for termination.

Context:

1. At any place in the program, but only after the first instruction has displayed something on the front end.

Syntax:

```
CALL WINEXECWAIT( command STRING )
 RETURNING result INTEGER
```

Notes:

1. *command* is the command to be executed on the front end.
2. The function executes the program and waits for its termination.
3. The function returns FALSE if a problem has occurred.

See also: WINEXEC(), DDE Support.

WINSHELLEXEC( ) MS Windows FE Only!

Purpose:

This function opens a document with the corresponding program, based on the file extension.

Context:

1. At any place in the program, but only after the first instruction has displayed something on the front end

Syntax:

```
CALL WINSHELLEXEC( filename STRING )
 RETURNING result INTEGER
```

Notes:

1. *filename* is the file to be opened on the front end.
2. The function executes the program and returns immediately.
3. The function returns FALSE if a problem has occurred.

See also: WINEXEC(), DDE Support.

WINOPENDIR( )

Purpose:

This function shows a dialog window to let the user select a directory path on the front end workstation file system.

Context:

1. At any place in the program, but only after the first instruction has displayed something on the front end

Syntax:

```
CALL WINOPENDIR( dirname STRING, caption STRING )
 RETURNING result STRING
```

Notes:

1. *dirname* is the default path to be displayed in the dialog window.
2. *caption* is the label to be displayed.
3. The function returns the directory path on success.

4. The function returns NULL if a problem has occurred or if the the user canceled the dialog.

WINOPENFILE( )

Purpose:

This function shows a dialog window to let the user select a file path on the front end workstation file system, for displaying.

Context:

1. At any place in the program, but only after the first instruction has displayed something on the front end.

Syntax:

```
CALL WINOPENFILE( dirname STRING, typename STRING,
   extlist STRING, caption STRING )
 RETURNING result STRING
```

Notes:

1. *dirname* is the default path to be displayed in the dialog window.
2. *typename* is the name of the file type to be displayed.
3. *extlist* is a blank-separated list of file extensions defining the file type.
4. *caption* is the label to be displayed.
5. The function returns the file path on success.
6. The function returns NULL if a problem has occurred or if the the user canceled the dialog.

WINSAVEFILE( )

Purpose:

This function shows a dialog window to let the user select a file path on the front end workstation file system, for saving.

Context:

1. At any place in the program, but only after the first instruction has displayed something on the front end

Syntax:

```
CALL WINSAVEFILE( dirname STRING, typename STRING,
   extlist STRING, caption STRING )
 RETURNING result STRING
```

Notes:

1. *dirname* is the default path to be displayed in the dialog window.
2. *typename* is the name of the file type to be saved.
3. *extlist* is a blank separated list of file extensions defining the file type.
4. *caption* is the label to be saved.
5. The function returns the file path on success.
6. The function returns NULL if a problem has occurred or if the the user canceled the dialog.

Windows DDE Support

Summary:

- What is DDE?
- Using DDE API
- The DDE API
- Example
- BDL Wrappers for DDE functions

See also: Built-in Functions.

What is DDE?

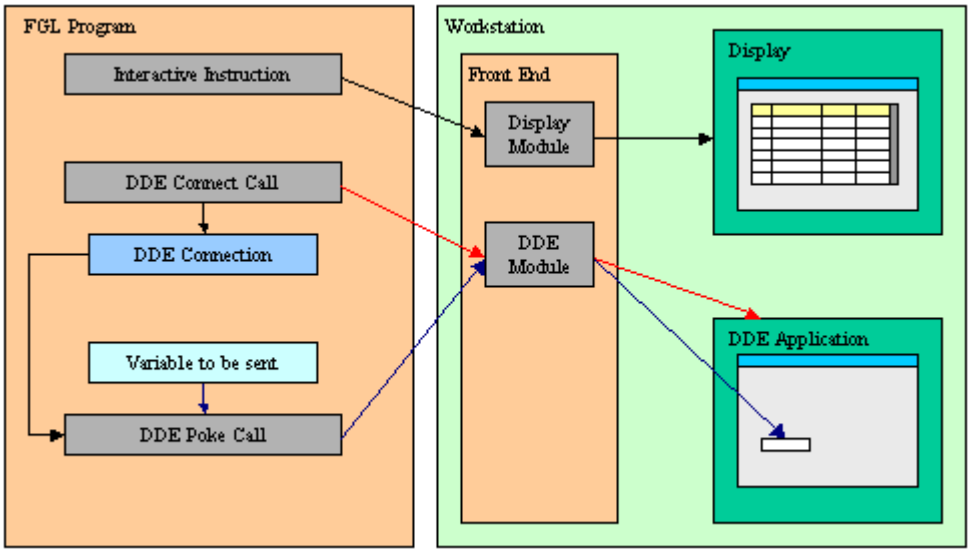
DDE is a form of inter-process communication implemented by Microsoft for Windows platforms. DDE uses shared memory to exchange data between applications. Applications can use DDE for one-time data transfers, and for ongoing exchanges in applications that send updates to one another as new data becomes available.

Please refer to your Microsoft documentation for DDE compatibility between existing versions. As an example, DDE commands were changed between Office 97 and Office 98.

Using DDE API

With DDE Support, you can invoke a Windows application and send data to or receive data from it. To use this functionality, the program must use the Windows Front End.

Before using the DDE functions, the TCP communication channel between the application and the front end must be established with a display (OPEN WINDOW, MENU, DISPLAY TO).



The DDE API is used in a four-part procedure, as described in the following steps:

1. The application sends the Front End the DDE order using the TCP/IP channel.
2. The Front End executes the DDE order and sends the data to the Windows application through the DDE API.
3. The Windows application executes the command and sends the result, which can be data or an error code, to the Front End.
4. The Windows Front End sends back the result to the application using the TCP/IP channel.

A DDE connection is uniquely identified by two values: The name of the DDE Application and the document. Most DDE functions require these two values to identify the DDE source or target.

The DDE API

The DDE API is based on the front call technique described in Front End Functions. All DDE functions are grouped in the **WINDDE** front end function module.

Function name	Description
DDEConnect	This function opens a DDE connection
DDEExecute	This function executes a command in the specified program
DDEFinish	This function closes a DDE connection
DDEFinishAll	This function closes all DDE connections, as well as the DDE server program
DDEError	This function returns DDE error information about the last DDE operation

DDEPeek	This function retrieves data from the specified program and document using the DDE channel
DDEPoke	This function sends data to the specified program and document using the DDE channel

---

DDEConnect

Purpose:

This function opens a DDE connection.

Syntax:

```
CALL ui.Interface.frontCall("WINDDE","DDEConnect",
        [ program, document ], [result] )
```

Notes:

1. *program* is the name of the DDE application.
2. *document* is the document that is to be opened.
3. *result* is an integer variable receiving the status.
4. *result* is TRUE if the function succeeded, FALSE otherwise.
5. If the function failed, use DDEError to get the description of the error.

Warnings:

1. If the function failed with "DMLERR_NO_CONV_ESTABLISHED", then the DDE application was probably not running; use execute or shellexec front call to start the DDE application.

DDEExecute

Purpose:

This function executes a DDE command.

Syntax:

```
CALL ui.Interface.frontCall("WINDDE","DDEExecute",
        [ program, document, command ], [result] )
```

Notes:

1. *program* is the name of the DDE application.

2. *document* is the document that is to be used.
3. *command* is the command that needs to be executed.
4. Refer to the *program* documentation for the syntax of *command*.
5. *result* is an integer variable receiving the status.
6. *result* is TRUE if the function succeeded, FALSE otherwise.
7. If the function failed, use DDEError to get the description of the error.

Warnings:

1. The DDE connection must be opened; see DDEConnect.

DDEFinish

Purpose:

This function closes a DDE connection.

Syntax:

```
CALL ui.Interface.frontCall("WINDDE","DDEFinish",
        [ program, document ], [result] )
```

Notes:

1. *program* is the name of the DDE application.
2. *document* is the document that is to be closed.
3. *result* is an integer variable receiving the status.
4. *result* is TRUE if the function succeeded, FALSE otherwise.
5. If the function failed, use DDEError to get the description of the error.

Warnings:

1. The DDE connection must be opened, see DDEConnect.

DDEFinishAll

Purpose:

This function closes all DDE connections as well as the DDE server program.

Syntax

```
CALL ui.Interface.frontCall("WINDDE","DDEFinishAll", [], [result] )
```

Notes:

1. Closes all DDE connections.
2. *result* is an integer variable receiving the status.
3. *result* is TRUE if the function succeeded, FALSE otherwise.

DDEError

Purpose:

This function returns the error information about the last DDE operation.

Syntax:

```
CALL ui.Interface.frontCall("WINDDE","DDEError", [], [errmsg] )
```

Notes:

1. *errmsg* is the error message. It is set to NULL if no error occurred.

DDEPeek

Purpose:

This function retrieves data from the specified program and document using the DDE channel.

Syntax:

```
CALL ui.Interface.frontCall("WINDDE","DDEPeek",
        [ program, container, cells ], [ result, value ] )
```

Notes:

1. *program* is the name of the DDE application.
2. *container* is the document or sub-document that is to be used.
   A sub-document can, for example, be a sheet in Microsoft Excel.
3. *cells* represents the working items; see the *program* documentation for the format of *cells*.
4. *value* represents the data to be retrieved; see the *program* documentation for the format of *values*.
5. *result* is an integer variable receiving the status.
6. *result* is TRUE if the function succeeded, FALSE otherwise.
7. If the function failed, use DDEError to get the description of the error.
8. *value* is a variable receiving the cells values.

Warnings:

1. The DDE connection must be opened; see DDEConnect.
2. DDEError can only be called once to check if an error occurred.

DDEPoke

Purpose:

This function sends data to the specified program and document using the DDE channel.

Syntax:

```
CALL ui.Interface.frontCall("WINDDE","DDEPoke",
        [ program, container, cells, values ], [result] )
```

Notes:

1. *program* is the name of the DDE application.
2. *container* is the document or sub-document that is to be used.
   A sub-document can, for example, be a sheet in Microsoft Excel.
3. *cells* represents the working items; see the *program* documentation for the format of *cells*.
4. *values* represents the data to be sent; see the *program* documentation for the format of *values*.
5. *result* is an integer variable receiving the status.
6. *result* is TRUE if the function succeeded, FALSE otherwise.
7. If the function failed, use DDEError to get the description of the error.

Warnings:

1. The DDE connection must be opened; see DDEConnect.
2. An error may occur if you try to set many (thousands of) cells in a single operation.

Example

dde_example.per

```
01 DATABASE formonly
02 SCREEN
03 {
04 Value to be given to top-left corner :
05 [f00                                 ]
06 Value found on top-left corner :
07 [f01                                 ]
```

```
08 }
09 ATTRIBUTES
10   f00 = formonly.val;
11   f01 = formonly.rval, NOENTRY;
```

dde_example.4gl

```
01 MAIN
02   -- Excel must be open with "File1.xls"
03   CONSTANT file = "File1.xls"
04   CONSTANT prog = "EXCEL"
05   DEFINE val, rval STRING
06   DEFINE res INTEGER
07   OPEN WINDOW w1 AT 1,1 WITH FORM "dde_example.per"
08   INPUT BY NAME val
09   CALL ui.Interface.frontCall("WINDDE","DDEConnect", [prog,file],
[res] )
10   CALL checkError(res)
11   CALL ui.Interface.frontCall("WINDDE","DDEPoke",
[prog,file,"R1C1",val], [res] );
12   CALL checkError(res)
13   CALL ui.Interface.frontCall("WINDDE","DDEPeek",
[prog,file,"R1C1"], [res,rval] );
14   CALL checkError(res)
15   DISPLAY BY NAME rval
16   INPUT BY NAME val WITHOUT DEFAULTS
17   CALL ui.Interface.frontCall("WINDDE","DDEExecute",
[prog,file,"[save]"], [res] );
18   CALL checkError(res)
19   CALL ui.Interface.frontCall("WINDDE","DDEFinish", [prog,file],
[res] );
20   CALL checkError(res)
21   CALL ui.Interface.frontCall("WINDDE","DDEFinishAll", [], [res] );
22   CALL checkError(res)
23   CLOSE WINDOW w1
24 END MAIN
25
26 FUNCTION checkError(res)
27   DEFINE res INTEGER
28   DEFINE mess STRING
29   IF res THEN RETURN END IF
30   DISPLAY "DDE Error:"
31   CALL ui.Interface.frontCall("WINDDE","DDEError",[],[mess]);
32   DISPLAY mess
33   CALL ui.Interface.frontCall("WINDDE","DDEFinishAll", [], [res] );
34   DISPLAY "Exit with DDE Error."
35   EXIT PROGRAM (-1)
36 END FUNCTION
```

BDL Wrappers to DDE front end functions

The following functions are provided for backward compatibility. We recommend that you use the front call functions if you write new code.

Warning: These functions (especially DDEExecute and DDEPoke) expect escaped TAB, CR and LF characters in the strings passed as parameters. For example, a TAB character must be written as `"\\t"` in a BDL string constant passed as parameter to the DDEPoke function.

Function	Description
DDEConnect( )	This function opens a DDE connection
DDEExecute( )	This function executes a command in the specified program
DDEFinish( )	This function closes a DDE connection
DDEFinishAll( )	This function closes all DDE connections as well as the DDE server program
DDEGetError( )	This function returns DDE error information about the last DDE operation
DDEPeek( )	This function retrieves data from the specified program and document using the DDE channel
DDEPoke( )	This function sends data to the specified program and document using the DDE channel

---

DDEConnect()

Purpose:

This function opens a DDE connection.

Syntax:

```
CALL DDEConnect ( program STRING, document STRING ) RETURNING SMALLINT
```

Notes:

1. *program* is the name of the DDE application.
2. *document* is the document that is to be opened.
3. The function returns TRUE if the connection succeeded, FALSE otherwise.
4. If the return value is FALSE, use DDEGetError() to get the description of the error.

Warnings:

1. If the function failed with "DMLERR_NO_CONV_ESTABLISHED", then the DDE application was probably not running; use WinExec() or WinShellExec() front call to start the DDE application.

DDEExecute()

Purpose:

This function executes a DDE command.

Syntax:

```
CALL DDEExecute ( program STRING, document STRING, command STRING )
RETURNING SMALLINT
```

Notes:

1. *program* is the name of the DDE application.
2. *document* is the document that is to be used.
3. *command* is the command that needs to be executed.
4. Refer to the *program* documentation for the syntax of *command*.
5. The function returns TRUE if the command execution succeeded, FALSE otherwise.
6. If the return value is FALSE, use DDEGetError() to get the description of the error.

Warnings:

1. The DDE connection must be opened see DDEConnect().

DDEFinish()

Purpose:

This function closes a DDE connection.

Syntax:

```
CALL DDEFinish ( program STRING, document STRING ) RETURNING SMALLINT
```

Notes:

1. *program* is the name of the DDE application.
2. *document* is the document that is to be closed.
3. The function returns TRUE if the function succeeded, FALSE otherwise.
4. If the return value is FALSE, use DDEGetError() to get the description of the error.

Warnings:

1. The DDE connection must be opened, see DDEConnect().

DDEFinishAll()

Purpose:

This function closes all DDE connections, as well as the DDE server program.

Syntax

```
CALL DDEFinishAll()
```

Notes:

1. Closes all DDE connections.

DDEGetError()

Purpose:

This function returns the error information about the last DDE operation.

Syntax:

```
CALL DDEGetError() RETURNING STRING
```

Notes:

1. The function returns the error message or NULL if no error occurred.

DDEPeek()

Purpose:

This function retrieves data from the specified program and document using the DDE channel.

Syntax:

```
CALL DDEPeek ( program STRING, container STRING, cells STRING )
RETURNING value
```

Notes:

1. *program* is the name of the DDE application.
2. *container* is the document or sub-document that is to be used. A sub-document can, for example, be a sheet in Microsoft Excel.
3. *cells* represents the working items; see the *program* documentation for the format of *cells*.
4. *value* represents the data to be retrieved; see the *program* documentation for the format of *values*.
5. If the function succeeded, DDEGetError() function returns NULL.

Warnings:

1. The DDE connection must be opened; see DDEConnect().
2. DDEGetError() can only be called once to check if an error occurred.

DDEPoke()

Purpose:

This function sends data to the specified program and document using the DDE channel.

Syntax:

```
CALL DDEPoke ( program STRING, container STRING, cells STRING, values
STRING ) RETURNING SMALLINT
```

Notes:

1. *program* is the name of the DDE application.
2. *container* is the document or sub-document that is to be used. A sub-document can, for example, be a sheet in Microsoft Excel.
3. *cells* represents the working items; see the *program* documentation for the format of *cells*.
4. *values* represents the data to be sent; see the *program* documentation for the format of *values*.
5. The function returns TRUE if the function succeeded, FALSE otherwise.
6. If the return value is FALSE, use DDEGetError() to get the description of the error.

Warnings:

1. The DDE connection must be opened; see DDEConnect().
2. An error may occur if you try to set many (thousands of) cells in a single operation.

XML Utilities

This pages describes the XML utility API provided by the language.

See also: Built-in Classes, DomDocument class, DomNode class, SaxAttributes class, XmlReader class, XmlWriter class, SaxDocumentHandler class.

DOM and SAX standards

The DOM (**D**ocument **O**bject **M**odel) is a programming interface specification being developed by the World Wide Web Consortium (W3C), that lets a programmer create and modify HTML pages and XML documents as full-fledged program objects. DOM is a full-fledged object-oriented, complex but complete API, providing methods to manipulate the full XML document as a whole. DOM is designed for small XML trees manipulation.

The SAX (**S**imple **A**PI for **X**ML) is a programming interface for XML, simpler as DOM. SAX is event-driven, streamed-data based, and designed for large trees.

The DOM and SAX APIs

The runtime system includes a set of built-in classes based on DOM and SAX.

The DOM API is composed of:

- The DomDocument class, that defines the interface to a DOM document. Instances of this class can be used to identify and manipulate an XML tree. *DomNode* object manipulation methods are provided by this class.
- The DomNode class, that defines the interface to an DOM node. Instances of this class can be used to identify and manipulate a branch of an XML tree. Child nodes and node attributes management methods are provided by this class.

The SAX API is composed of:

- The SaxAttributes class represents a set of element attributes. It is used with an XmlReader or an XmlWriter object.
- The XmlReader class, that is defined to read XML. The XML document processing is based on events.
- The XmlWriter class, that is defined to write XML. The XML document processing is based on events.
- The SaxDocumentHandler class, which provides an interface to implement a SAX filter using functions.

Controlling the user interface with DOM/SAX APIs

The runtime system represents the user interface of a program with a DOM tree. User interface elements can be manipulated with the built-in classes described in this section.

For more details about the user interface manipulation, see the Dynamic User Interface.

Database Connections

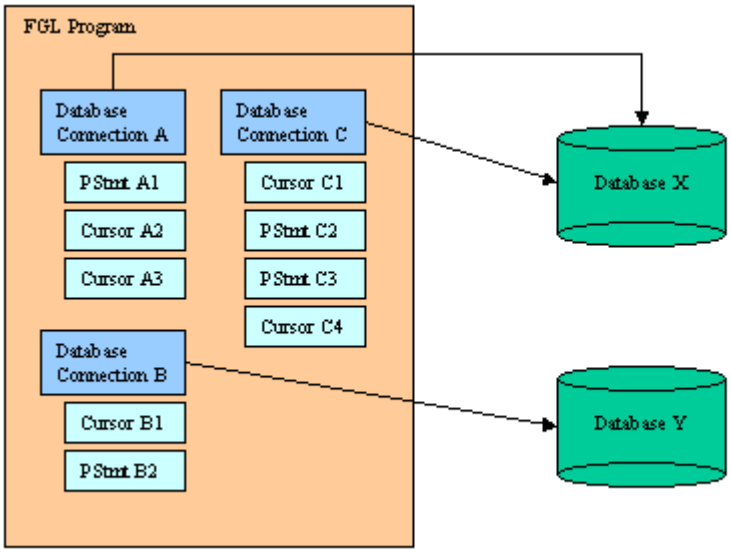
Summary:

- What is a database connection?
- Database Specification
  - Connection parameters in connection string
  - Keep the compiled programs configurable
  - Database Specification when using Informix
    - Informix environment variables on Windows platforms
  - Database Specification when using other databases
    - Direct database specification
    - Indirect database specification
    - Informix emulations parameters
    - Database vendor specific parameters
  - Database user authentication
    - Specifying user name and login with CONNECT
    - Authenticating users with Informix
    - Authenticating users with Oracle
    - Authenticating user with SQL Server
- Global Configuration Parameters
  - Default Database Driver
- Database Client Environments
- The FGLSQLDEBUG environment variable
- The SQLCA record
- STATUS, SQLCA.SQLCODE, SQLSTATE and SQLERRMESSAGE
- Interrupting SQL Statements
- Unique-session mode:
  - Opening a connection (`DATABASE`)
  - Closing a connection (`CLOSE DATABASE`)
- Multi-session mode:
  - Opening connections (`CONNECT TO`)
  - Selecting connections (`SET CONNECTION`)
  - Closing connections (`DISCONNECT`)

See also: Transactions, Static SQL, Dynamic SQL, Result Sets, SQL Errors, Programs.

What is a database connection?

A Database Connection is a session of work, opened by the program to communicate with a specific database server, in order to execute SQL statements as a specific user.



The database user can be identified explicitly for each connection. Usually, the user is identified by a login and a password, or by using the authentication mechanism of the operating system (or even from a tier security system).

The database connection instructions can not be prepared as Dynamic SQL statements; they must be static SQL statements.

There are two kind of connection modes: *unique-session* and *multi-session* mode. When using the `DATABASE` and `CLOSE DATABASE` instructions, you are in *unique-session* mode. When using the `CONNECT TO`, `SET CONNECTION` and `DISCONNECT` instructions you are in *multi-session* mode. The modes are not compatible. It is strongly recommended that you choose the session mode and not mix both kinds of instructions.

In *unique-session* mode, you simply connect with the `DATABASE` instruction; that creates a current session. You disconnect from the current session with the `CLOSE DATABASE` instruction, or when another `DATABASE` instruction is executed, or when the program ends.

In *multi-session* mode, you open a session with the `CONNECT TO` instruction; that creates a current session. You can open other connections with subsequent `CONNECT TO` instructions. To switch to a specific session, use the `SET CONNECTION` instruction; this suspends other opened connections. Finally, you disconnect from the current, from a specific, or from all sessions with the `DISCONNECT` instruction. The end of the program disconnects all sessions automatically.

Once connected to a database server, you have a current database session. Any subsequent SQL statement is executed in the context of the current database session.

Warning:

1. Before creating database connections, make sure you have properly installed and configured Genero BDL, using the correct database client environment and driver. For more information, see Installation and Setup.

Database Specification

The Database Specification identifies the data source (the database and database server) you want to connect to.

There are different ways to identify the data source, depending on the database type. For example, when you connect to Oracle, you cannot specify the database server as you do with Informix by using the '`dbname@dbserver`' notation.

For portability reasons, it is not recommended that you use database vendor specific syntax in the database specification (like '`dbname@dbserver`'). We recommend using a simple symbol instead, and configuring the connection parameters in external resource files. The **ODI** architecture allows this indirect database specification using the FGLPROFILE configuration file.

Specifying connection parameters in the connection string

Although this is not recommended for abstract programming reasons, you can specify connection parameters in the string used by the connection instructions.

This behavior is enabled when you use a **plus sign** in the connection string:

```
dbname+property='value'[,...]
```

In this syntax, *property* can be one of the following parameters:

Parameter	Description
`resource`	Specifies which '`dbi.database`' entries have to be read from the FGLPROFILE configuration file. When this property is set, the database interface reads `dbi.database.`*`name`*`.*` entries, where *name* is the value specified for the resource parameter.
`driver`	Defines the database driver library to be loaded (filename without extension).
`source`	Specifies the data source of the database (for example, Oracle's TNS name).
`username`	Defines the name of the database user.
`password`	Defines the password of the database user. **Warning:**

Should not be used in production!

In the following example, **driver**, **source** and **resource** are specified in the connection string:

```
01 MAIN
02    DEFINE db CHAR(50)
03    LET db = "stores+driver='dbmora',source='orcl',resource='ora'"
04    DATABASE db
05    ...
06 END MAIN
```

Keep the compiled programs configurable

You can use a string variable with the DATABASE or CONNECT TO statement, in order to specify the database source at runtime (to be loaded from your own configuration file or from an environment variable). This solution gives you the best flexibility.

```
01 MAIN
02    DEFINE db, us, pwd CHAR(50)
03    LET db = arg_val(1)
03    LET us = arg_val(2)
03    LET pwd = arg_val(3)
04    CONNECT TO db USER us USING pwd
05    ...
06 END MAIN
```

Database specification when using the Informix driver Informix only!

When using an Informix database driver, you can use the following syntax for the database specification:

Database Specification	Description
`dbname`	Connects to the database server identified by the Informix environment (for example, with the INFORMIXSERVER environment variable) and opens the database `dbname`.
`@dbserver`	Connects to the database server identified by `dbserver`. This database specification does not select any database, the program is only connected to the database server.
`dbname@dbserver`	Connects to the database server identified by `dbserver` and opens the database `dbname`.

Informix environment variables on Windows platforms

On Windows platforms, in a C console application, the Informix environment variables must be set with a call to **ifx_putenv()**. See INFORMIX ESQL/C documentation for more details about environment settings.

By default, the database driver automatically calls **ifx_putenv()** for all standard Informix environment variables such as INFORMIXDIR with the current value set in the console environment. You can specify additional environment variables to be set with the following FGLPROFILE entries:

dbi.stdifx.environment.count = *max*
dbi.stdifx.environment.*index* = "*variable*"

Database specification when using a non-Informix database

To connect to a database server, additional connection parameters are often required. Most database engines require a name to identify the server, a name to identify the database entity, a user name and a password. Some parameters might be omitted: For example, when using Oracle, the server can be implicitly defined by the ORACLE_SID environment variable if the program and the database server run on the same system. The ODI architecture allows you to define these parameters indirectly in the FGLPROFILE configuration file.

Direct database specification method

The **Direct Database Specification** method takes place when the database name used in the program is not used in FGLPROFILE to define the data source with a '`dbi.database.dbname.source`' entry. In this case, the database specification used in the connection instruction is used as the data source.

This method is well known for standard Informix drivers, where you directly specify the database name and, if needed, the Informix server:

```
01 MAIN
02    DATABASE stores@orion
03    ...
04 END MAIN
```

In the next example, the database server is PostgreSQL. The string used in the connection instruction defines the PostgreSQL database (stock), the host (localhost), and the TCP service (5432) the postmaster is listening to. As PostgreSQL syntax is not allowed in standard BDL, a CHAR variable must be used:

```
01 MAIN
02    DEFINE db CHAR(50)
03    LET db = "stock@localhost:5432"
04    DATABASE db
05    ...
06 END MAIN
```

Indirect database specification method

Indirect Database Specification method takes place when the database specification used in the connection instruction corresponds to a '`dbi.database.dbname.source`' entry defined in the FGLPROFILE configuration file. In this case, the database specification is considered a key to read the connection information from the configuration file:

Program:

```
01 MAIN
02    DATABASE stores
03    ...
04 END MAIN
```

FGLPROFILE:

```
dbi.database.stores.source   = "stock@localhost:5432"
dbi.database.stores.driver   = "dbmpgs721"
```

This technique is flexible: The database name in programs is a kind of alias used to define the real database. Using this method, your can develop your application with the database name "stores" and connect to the real database "stores1" in a production environment.

In FGLPROFILE, the entries starting with '`dbi.database`' group information defining data sources by name:

```
dbi.database.dsname.source   = "value"
dbi.database.dsname.driver   = "value"
dbi.database.dsname.username = "value"
dbi.database.dsname.password = "value"  -- Warning: not encrypted, do
not use in production!
```

The "**source**" entry identifies the data source name. The following table describes the meaning of this parameter for the supported databases:

Database Type	Value of "source" entry	Description
Genero DB	`datasource`	ODBC Data Source
Generic ODBC	`datasource`	ODBC Data Source
Informix	`dbname[@dbserver]`	Informix database specification
IBM DB2	`dsname`	DB2 Catalogued Database
MySQL	`dbname[@host[:port]]` or `dbname[@localhost~socket]`	Database Name @ Host Name : TCP Port or

		Database Name @ Local host ~ Unix socket file
ORACLE	`tnsname`	Oracle TNS Service name
PostgreSQL	`dbname[@host[:port]]`	Database Name @ Host Name : TCP Port
SQL Server	`datasource`	ODBC Data Source
Sybase ASA	`dbname[@engine]`	Database Name @ Engine Name

If the "**source**" entry is defined with an empty value (""), the database interface connects to the default database server, which is usually the local server. If this entry is not present in FGLPROFILE, the Direct Database Specification method takes place (see above for more details).

The "**driver**" entry identifies the shared library or DLL to be used. Driver file names do not have to be specified with a file extension.

If you have a lot of databases, you can define a default driver with the Default Database Driver entry.

Database drivers shared libraries are located in **FGLDIR/dbdrivers** on both UNIX and Windows platforms. Some drivers may not be available on a specific platform (for example if the target database client does not exist). Contact your support if you do not find the driver you are looking for.

The following table defines the database driver names according to the database client type:

Database Type	Driver library prefix	Example
Genero DB	`dbmads`	`dbmads3x.so`
Generic ODBC	`dbmodc`	`dbmodc3x.dll`
Informix	`dbmifx`	`dbmifx9xx.so`
IBM DB2	`dbmdb2`	`dbmdb28x.so`
MySQL	`dbmmys`	`dbmmys41x.so`
ORACLE	`dbmora`	`dbmora92x.so`
PostgreSQL	`dbmpgs`	`dbmpgs80x.so`
SQL Server (MDAC)	`dbmmsv`	`dbmmsv80.dll`
SQL Server (Native Client)	`dbmsnc`	`dbmsnc90.dll`
SQL Server (FreeTDS)	`dbmftm`	`dbmftm90.dll`
Sybase ASA	`dbmasa`	`dbmasa8x.so`

The "**username**" and "**password**" entries define the default database user, when the program uses the DATABASE instruction or the CONNECT TO instruction without the `USER` clause.

Warning: The "username" and "password" entries are not encrypted. These parameters are provided to simplify migration and should not be used in production. You better use CONNECT TO with a USER / USING clause to avoid any security hole, or OS user authentication. Example of database servers supporting OS user authentication: Informix, Oracle, SQL Server and Genero db.

The "**username**" and "**password**" entries take effect based on the connection instruction as described in the following table:

Connection Instruction	FGLPROFILE	Effect
`DATABASE dbname` or `CONNECT TO "dbname"`	No default user definition	No user specification is provided to the database server. Usually, the Operating System authentication takes place.
`DATABASE dbname` or `CONNECT TO "dbname"`	With default user definition	The FGLPROFILE user name and password are used to connect to the database server.
`CONNECT TO "dbname" USER "username" USING "password"`	N/A	The user information of the CONNECT TO instruction are used to identify the actual user

Informix emulation parameters in FGLPROFILE

To simplify the migration process to other databases, the database interface and drivers can emulate some Informix-specific features like SERIAL columns and temporary tables; the drivers can also do some SQL syntax translation.

Warning: Avoid using Informix emulations; write portable SQL code instead as described in SQL Programming. Informix emulations are only provided to help you in the migration process. Disabling Informix emulations improves performance, because SQL statements do not have to be parsed to search for Informix-specific syntax.

Emulations can be controlled with FGLPROFILE parameters. You can disable all possible switches step-by-step, in order to test your programs for SQL compatibility.

Global switch to enable or disable Informix emulations:

```
dbi.database.dbname.ifxemul = { true | false }
```

Feature specific switches:

The 'ifxemul.datatype' switches define whether the specified data type must be converted to a native type (for example, when creating a table):

```
dbi.database.dbname.ifxemul.datatype.type = { true | false }
```

Here, *type* can be one of: char, varchar, datetime, decimal, money, float, real, integer, smallint, serial, text, byte.

To control SERIAL generation type, you can use the following switch:

```
dbi.database.dbname.ifxemul.datatype.serial.emulation = { "native" |
"regtable" | "trigseq" }
```

When using "native", the database driver creates a native sequence generator - it is fast, but not fully compatible to Informix SERIAL. When using "regtable", you must have the SERIALREG table created - it is slower than the "native" emulation, but compatible to Informix SERIAL. The serial emulation "trigseq", can be used by some database drivers, to use triggers with native sequence generators.

The 'temptables' switch can be used to control temporary table emulation:

```
dbi.database.dbname.ifxemul.temptables = { true | false }
```

The 'temptables.emulation' switch can be used to specify what type of tables must be used to emulate temporary tables:

```
dbi.database.dbname.ifxemul.temptables.emulation = { "default" |
"global" }
```

The 'dblquotes' switch can be used to define whether double quoted strings must be converted to single quoted strings:

```
dbi.database.dbname.ifxemul.dblquotes = { true | false }
```

If this emulation is enabled, all double quoted strings are converted, including database object names.

The 'outers' switch can be used to control Informix OUTER specification:

```
dbi.database.dbname.ifxemul.outers = { true | false }
```

It is better to use standard ISO outer joins in your SQL statements.

The 'today' switch can be used to convert the TODAY keyword to a native expression returning the current date:

```
dbi.database.dbname.ifxemul.today = { true | false }
```

The 'current' switch can be used to convert the CURRENT X TO Y expressions to a native expression returning the current time:

```
dbi.database.dbname.ifxemul.current = { true | false }
```

The 'selectunique' switch can be used to convert the SELECT UNIQUE to SELECT DISTINCT:

```
dbi.database.dbname.ifxemul.selectunique = { true | false }
```

It is better to replace all UNIQUE keywords by DISTINCT.

The 'colsubs' switch can be used to control column sub-strings expressions (col[x,y]) to native sub-string expressions:

```
dbi.database.dbname.ifxemul.colsubs = { true | false }
```

The 'matches' switch can be used to define whether MATCHES expressions must be converted to LIKE expressions:

```
dbi.database.dbname.ifxemul.matches = { true | false }
```

It is better to use the LIKE operator in your SQL statements.

The 'length' switch can be used to define whether LENGTH function names have to be converted to the native equivalent:

```
dbi.database.dbname.ifxemul.length = { true | false }
```

The 'rowid' switch can be used to define whether ROWID keywords have to be converted to native equivalent:

```
dbi.database.dbname.ifxemul.rowid = { true | false }
```

It is better to use primary keys instead.

The 'listupdate' switch can be used to convert the UPDATE statements using non-ANSI syntax:

```
dbi.database.dbname.ifxemul.listupdate = { true | false }
```

The 'extend' switch can be used to convert simple EXTEND() expressions to native date/time expressions:

```
dbi.database.dbname.ifxemul.extend = { true | false }
```

Defining database vendor specific parameters in FGLPROFILE

Some database vendor specific connection parameters can be configured by using FGLPROFILE entries with the following syntax:

```
dbi.database.dsname.dbtype.param.[.subparam] = "value"
```

The table below describes all database vendor specific parameters supported:

Database Server	Parameters
Genero DB	
	dbi.database.*dsname*.ads.schema Name of the database schema to be selected after connection is established. Example: `dbi.database.stores.ads.schema = "store2"` Usage: Set this parameter to a specific schema in order to share the same table with all users.
IBM DB2	

	dbi.database.*dsname*.db2.schema
	Name of the database schema to be selected after connection is established.
	Example:
	`dbi.database.stores.db2.schema = "store2"`
	Usage:
	Set this parameter to a specific schema in order to share the same table with all users.
	dbi.database.*dsname*.db2.prepare.deferred
	True/False Boolean to enable/disable deferred prepare.
	Example:
	`dbi.database.stores.db2.prepare.deferred = true`
	Usage:
	Set this parameter to true if you do not need to get SQL errors during PREPARE statements: SQL statements will be sent to the server when executing the statement (OPEN or EXECUTE). The default is false (SQL statements are sent to the server when doing the PREPARE).

ORACLE	

	dbi.database.*dsname*.ora.schema
	Name of the database schema to be selected after connection is established.
	Example:
	`dbi.database.stores.ora.schema = "store2"`
	Usage:
	Set this parameter to a specific schema in order to share the same table with all users.
	dbi.database.*dsname*.ora.prefetch.rows
	Maximum number of rows to be pre-fetched.
	Example:
	`dbi.database.stores.ora.prefetch.rows = 50`
	Usage:
	You can use this parameter to increase performance by

defining the maximum number of rows to be fetched automatically. However, the bigger this parameter is, the more memory is used by each program. This parameter applies to all cursors in the application.
The default is 10 rows.

dbi.database.*dsname*.ora.prefetch.memory
Maximum buffer size for pre-fetching (in bytes).
Example:
```
dbi.database.stores.ora.prefetch.memory = 4096
```
Usage:
This parameter is equivalent to prefetch.rows, but here you can specify the memory size instead of the number of rows. As prefetch.rows, this parameter applies to all cursors in the application.
The default is 0, which means that memory size is not included in computing the number of rows to pre-fetch.

dbi.database.*dsname*.ora.sid.command
SQL command (SELECT) to generate a unique session id (used for temp table names).
Example:
```
dbi.database.stores.ora.sid.command = "SELECT
TO_CHAR(SID)||'_'||TO_CHAR(SERIAL#) FROM V$SESSION
WHERE AUDSID=USERENV('SESSIONID')"
```
Usage:
By default the driver uses "SELECT USERENV('SESSIONID') FROM DUAL". This is the standard session identifier in Oracle, but it can become a very large number and can't be reset.
This parameter gives you the freedom to provide your own way to generate a session id.
The SELECT statement must return a single row with one single column.
Value can be an integer or an identifier.

SQL Server (MDAC)

dbi.database.*dsname*.msv.logintime
Connection timeout (in seconds).
Example:
```
dbi.database.stores.msv.logintime = 5
```
Usage:
Set this parameter to raise an SQL error if the connection can not be established after the given number of seconds.
The default is 5 seconds.

dbi.database.*dsname*.msv.prefetch.rows
Maximum number of rows to be pre-fetched.
Example:
```
dbi.database.stores.msv.prefetch.rows = 50
```
Usage:

	You can use this parameter to increase performance. However, the bigger this parameter is, the more memory is used by each program. The default is 10 rows.
SQL Server (NCLI)	

	dbi.database.*dsname*.snc.logintime
	Connection timeout (in seconds).
	Example:
	`dbi.database.stores.snc.logintime = 5`
	Usage:
	Set this parameter to raise an SQL error if the connection can not be established after the given number of seconds.
	The default is 5 seconds.
	dbi.database.*dsname*.snc.prefetch.rows
	Maximum number of rows to be pre-fetched.
	Example:
	`dbi.database.stores.snc.prefetch.rows = 50`
	Usage:
	You can use this parameter to increase performance. However, the bigger this parameter is, the more memory is used by each program.
	The default is 10 rows.

SQL Server (FreeTDS)	

	dbi.database.*dsname*.ftm.logintime
	Connection timeout (in seconds).
	Example:
	`dbi.database.stores.ftm.logintime = 5`
	Usage:
	Set this parameter to raise an SQL error if the connection can not be established after the given number of seconds.
	The default is 5 seconds.
	dbi.database.*dsname*.ftm.prefetch.rows
	Maximum number of rows to be pre-fetched.
	Example:
	`dbi.database.stores.ftm.prefetch.rows = 50`
	Usage:
	You can use this parameter to increase performance. However, the bigger this parameter is, the more memory is used by each program.
	The default is 10 rows.

Sybase ASA	
	dbi.database.*dsname*.asa.logintime Connection timeout (in seconds). Example: `dbi.database.stores.asa.logintime = 10`

Usage:
Set this parameter to raise an SQL error if the connection can not be established after the given number of seconds. The default is 5 seconds.

Database user authentication

Connecting to a database server is not just specifying a database name. Informix 4gl programmers are used to write "`DATABASE dbname`" to get connected. But this is only possible when the database server is configured to trust Operating System users. Only a few database server support OS authentication. Database users are usually defined in the database server and must be explicitly identified by a user name and password. Note also that some database servers support external authentication methods, which can be used with Genero. See DB specific documentation for more details.

Specifying a user name and password with the CONNECT

In order to specify a user name and password, you must use the `USER`/`USING` clause of the CONNECT instruction:

```
01 MAIN
02    CONNECT TO "orc1fox+driver='dbmoraA2x'" USER "scott" USING
"tiger"
03    ...
04 END MAIN
```

User name and login could be specified with FGLPROFILE entries but we strongly discourage you to do this for security reasons.

Authenticating users with Informix

Informix users are operating system users with database connection privileges, if the client program resides on the same machine as the database server, you typically use OS authentication and don't need to provide a user name and password.

However, you need to specify a user name and password if you want to connect to a remove server that does not have trusted connection configured.

Authenticating users with Oracle

Oracle users can be authenticated in different manner: as DB users, as OS users or with another external authentication method like Kerberos.

If you don't specify the `USER`/`USING` clause, OS authentication takes place.

An Oracle connection can also be established as SYSDBA or SYSOPER users. This is possible with Genero by specifying the following strings after the user name in the `USER` clause of the CONNECT instruction:

String passed to USER clause after user name	Effect as Oracle connection
/SYSDBA	Connection will be established as SYSDBA user.
/SYSOPER	Connection will be established as SYSOPER user.

Note that you must specify the user login before the /SYSDBA or /SYSOPER strings:

```
01 CONNECT TO "orc1fox+driver='dbmoraA2x'" USER "orauser/SYSDBA" USING
"fourjs"
```

Authenticating users with SQL Server

SQL Server users can be authenticated as DB users or with the Windows users.

If you don't specify the `USER`/`USING` clause, OS authentication takes place.

Global Configuration Parameters

Default Database Driver

With the following entry, you can define a default driver identifying the shared library or DLL to be used to connect to the database:

```
dbi.default.driver = "value"
```

Database Client Environment

To connect to a database server, the BDL programs must be executed in the correct database client environment. The database client software is usually included in the database server software, so you do not need to install it when your programs are executed on the same machine as the database server. However, you may need to install the database client software in three-tier configurations, where applications and database servers run on different systems.

This section describes basic configuration elements of the database client environment for some well-known database servers.

Genero DB

1. The **ANTSHOME** environment variable must define the Genero DB software installation path.

2. The **PATH** environment variable must define the access path to database client programs.
3. On UNIX, **LD_LIBRARY_PATH** (or equivalent) must hold the path to **$ANTSHOME/antsodbc**.
4. The ANTS ODBC client library '**libaodbc***' must be available.
5. You can make a connection test with the ANTS **antscmd** tool.

IBM DB2 Universal Database

1. The **DB2DIR** environment variable must define the DB2 software installation path.
2. The **PATH** environment variable must define the access path to database client programs.
3. On UNIX, **LD_LIBRARY_PATH** (or equivalent) must hold the path to **$DB2DIR/lib**.
4. The DB2 client library '**DB2DIR/lib/libdb2***' must be available.
5. The **remote server node** and the **remote database** must be declared locally with the CATALOG db2 command.
6. You can make a connection test with the IBM **db2** tool.

IBM Informix Dynamic Server

1. The **INFORMIXDIR** environment variable must define the Informix software installation path.
2. The **PATH** environment variable must define the access path to database client programs.
3. On UNIX, **LD_LIBRARY_PATH** (or equivalent) must hold the path to **$INFORMIXDIR/lib:$INFORMIXDIR/lib/esql**.
4. The Informix client libraries '**INFORMIXDIR/lib/***' must be available.
5. The **INFORMIXSERVER** environment variable can be used to define the name of the database server.
6. The **sqlhost** file must define the database server identified by INFORMIXSERVER.
7. You can make a connection test with the Informix **dbaccess** tool.

MySQL

1. The **MYSQLDIR** environment variable must define the MySQL software installation path.
2. On UNIX, **LD_LIBRARY_PATH** (or equivalent) must hold the path to **$MYSQLDIR/lib**.
3. The **PATH** environment variable must define the access path to database client programs.
4. You can make a connection test with the **mysql** tool.

Oracle

1. The **ORACLE_HOME** environment variable must define the Oracle software installation path.

2. The **ORACLE_SID** environment variable can be used to define the name of the local database instance.
3. The **PATH** environment variable must define the access path to database client programs.
4. On UNIX, **LD_LIBRARY_PATH** (or equivalent) must hold the path to **$ORACLE_HOME/lib**.
5. The Oracle client library '**ORACLE_HOME/lib/libclntsh***' must be available.
6. The **TNSNAMES.ORA** file must define the database server identifiers for remote connections (the **Oracle Listener** must be started on the database server to allow remote connections).
7. The **SQLNET.ORA** file must define network settings for remote connections.
8. You can make a connection test with the Oracle **sqlplus** tool.

PostgreSQL

1. The **PGDIR** environment variable must define the PostgreSQL software installation path.
2. The **PATH** environment variable must define the access path to database client programs.
3. On UNIX, **LD_LIBRARY_PATH** (or equivalent) must hold the path to **$PGDIR/lib**.
4. The PostgreSQL client library '**PGDIR/lib/libpq***' must be available.
5. On the database server, the **pg_hba.conf** file must define security policies.
6. You can make a connection test with the PostgreSQL **psql** tool.

SQL Server

1. Make sure that **ODBC data source** is defined on database client and database server systems, with the correct ODBC driver. Note that Genero FGL provides different sort of SQL Server drivers:
    - The **MSV** driver is based on the Microsoft Data Access Components (MDAC) ODBC driver (SQLSRV32.DLL). This driver is obsolete if you are using SQL Server 2005.
    - The **SNC** driver is based on the **SQL Native Client** ODBC driver (SQLNCLI.DLL). This is the new driver to be used with SQL Server 2005.
    - The **FTM** driver is based on the **FreeTDS** ODBC driver (libtdsodbc.so). This driver can be used if you want to connect to SQL Server from a UNIX machine.
2. The **PATH** environment variable must define the access path to database client programs (**ODBC32.DLL**).
3. Check the SQL Server Client configuration with the **Client Network Utility** tool: Verify that the ANSI to OEM conversion corresponds to the execution of FGL applications in a CONSOLE environment.
4. You can make a connection test with the Microsoft **Query Analyzer** tool.

Sybase ASA

1. The **ASADIR** environment variable must define the Sybase ASA software installation path.

2. The **PATH** environment variable must define the access path to database client programs.
3. Check the Sybase Client configuration.
4. You can make a connection test with the Sybase **ISQL** tool.

FGLSQLDEBUG

You can set the **FGLSQLDEBUG** environment variable to get debug information on SQL instructions. This variable can be set to an integer value from 0 to 10, according to the debugging details you want to see. The debug messages are sent to the standard error stream. If needed, you can redirect the standard error output into a file.

Unix (shell) example:

```
FGLSQLDEBUG=3
export FGLSQLDEBUG
fglrun myprog 2>sqldbg.txt
```

SQLCA

Purpose:

The `SQLCA` variable is a predefined record containing information about the execution of an SQL statement.

Syntax:

```
SQLCA
```

Definition:

```
DEFINE SQLCA RECORD
  SQLCODE INTEGER,
  SQLERRM CHAR(71),
  SQLERRP CHAR(8),
  SQLERRD ARRAY[6] OF INTEGER,
  SQLAWARN CHAR(8)
END RECORD
```

Notes:

1. The `SQLCA` record is filled after any SQL statement execution.
2. The "SQLCA" name stands for "SQL Communication Area".
3. `SQLCODE` contains the SQL execution code (0 = OK, 100 = not row found, <0 = error).
4. `SQLERRM` contains the Informix error message parameter.

5. SQLERRP is not used at this time.
6. SQLERRD[1] is not used at this time.
7. SQLERRD[2] contains the last SERIAL or the native SQL error code.
8. SQLERRD[3] contains the number of rows processed in the last statement (server dependent).
9. SQLERRD[4] contains the estimated CPU cost for the query (server dependent).
10. SQLERRD[5] contains the offset of the error in the SQL statement text (server dependent).
11. SQLERRD[6] contains the ROWID of the last row that was processed (server dependent).
12. SQLAWARN contains the ANSI warning represented by a W character at a given position in the string.
13. SQLAWARN[1] is set to W when any of the other warning characters have been set to W.
14. SQLAWARN[2] is set to W when:
    - a CHAR value has been truncated.
    - an Informix database without transactions has been selected.
15. SQLAWARN[3] is set to W when:
    - an aggregate like SUM() encountered a NULL value.
    - an Informix database in ANSI/ISO mode has been selected.
16. SQLAWARN[4] is set to W when:
    - the number of SELECT items is not the same as the number of INTO variables.
    - an Informix Dynamic Server database has been selected.
17. SQLAWARN[5] is set to W when:
    - a float to decimal conversion is used.
18. SQLAWARN[6] is set to W when:
    - the program executes an extension to the ANSI/ISO standard.
19. SQLAWARN[7] is set to W when:
    - query skips a table fragment.
20. SQLAWARN[8] is not used at this time .

Warnings:

1. SQLCA can be modified by hand, but this is not recommended because it may become read-only in a later release.

Example:

```
01 MAIN
02   WHENEVER ERROR CONTINUE
02   DATABASE stores
03   SELECT COUNT(*) FROM foo   -- Table should not exist!
04   DISPLAY SQLCA.SQLCODE, SQLCA.SQLERRD[2]
05 END MAIN
```

STATUS, SQLCA.SQLCODE, SQLSTATE and SQLERRMESSAGE

If an error occurs during an SQL statement execution, you can get the error description in the `STATUS`, `SQLCA.SQLCODE`, `SQLSTATE` and `SQLERRMESSAGE` built-in registers.

`STATUS` is the global language error code register, set for any kind of error (even non-SQL). When an SQL error occurs, the Informix SQL error code held by `SQLCA.SQLCODE` is copied into `STATUS`. The register `SQLCA.SQLCODE` returns the Informix error code. `SQLSTATE` returns the standard ANSI error code and `SQLERRMESSAGE` returns the database specific error message.

Use `SQLCA.SQLCODE` for SQL error management, and `STATUS` for 4gl errors.

Warnings:

1. `SQLSTATE` is an ANSI standard specification, but not all database servers support this register. For example, Oracle 8.x and 9.0 engines do not support this

See also: STATUS, SQLCA, SQLSTATE, SQLERRMESSAGE, Exceptions.

Interrupting SQL Statements

Syntax:

```
OPTIONS SQL INTERRUPT { ON | OFF }
```

Notes:

1. By default, SQL interruption is OFF.

Warnings:

1. Not all database servers support SQL interruption.
2. You must set INT_FLAG to FALSE before executing the SQL statement.

Usage:

Typical FGL programs control the interrupt signals, by using the following instructions:

```
DEFER INTERRUPT
DEFER QUIT
```

If the database server supports SQL interruption, the runtime system can enable interruption of long SQL queries when you set the SQL INTERRUPT program option. When the program gets an interrupt signal from the system, the running SQL statement is stopped and the INT_FLAG global variable is set to TRUE.

Example:

```
01 MAIN
02    DEFER INTERRUPT
03    DEFER QUIT
04    DATABASE stock
05    OPTIONS SQL INTERRUPT ON
06    LET INT_FLAG = FALSE
07    SELECT COUNT(*) FROM items WHERE items_value > 100
08    IF INT_FLAG THEN
09       DISPLAY "Query was interrupted by user"
10    END IF
11 END MAIN
```

DATABASE

Purpose:

Opens a new database connection in unique-session mode.

Syntax:

```
DATABASE { dbname[@dbserver] | variable | string } [EXCLUSIVE]
```

Notes:

1. *dbname* identifies the database name.
2. *dbserver* identifies the Informix database server (INFORMIXSERVER). **Informix only!**
3. *variable* can be any character string defined variable containing the database specification.
4. *string* can be a string literal containing the database specification.

Usage:

The `DATABASE` instruction opens a connection to the database server, like CONNECT TO, but without user and password specification.

By default the database user is identified by the current operating system user, but it can be authenticated according to database specification parameters.

The `EXCLUSIVE` keyword can be used to open an Informix database in exclusive mode to prevent access by anyone but the current user. **Informix only!**

If a current connection exists, it is automatically closed before connecting to the new database.

If the connection could not be established, the instruction raises an exception. For example, if you specify a database that the runtime system cannot locate, or cannot open, or for which the user of your program does not have access privileges, an exception is raised.

Warnings:

1. When used in a program block, the `DATABASE` instruction has a different meaning than when it is used outside a program block. See Database Schema Specification for more details.
2. The `EXCLUSIVE` keyword is specific to Informix databases; do not use this keyword when programming for non-Informix databases.

Tips:

1. The CONNECT TO instructions allow better control over database connections; you should use these instructions instead of `DATABASE` and `CLOSE DATABASE`.

Example 1: Using a static database name.

```
01 MAIN
02    DATABASE stores
03    SELECT COUNT(*) FROM customer
04 END MAIN
```

Example 2: Using a variable.

```
01 MAIN
02    DEFINE dbname VARCHAR(100)
03    LET dbname = arg_val(1)
04    DATABASE dbname
05    SELECT COUNT(*) FROM customer
06 END MAIN
```

CLOSE DATABASE

Purpose:

Closes the current database connection when in unique-session mode.

Syntax:

```
CLOSE DATABASE
```

Usage:

The `CLOSE DATABASE` instruction closes the current database connect opened by the DATABASE instruction.

Warnings:

1. The current connection is automatically closed when the program ends.

Example:

```
01 MAIN
02    DATABASE stores1
03    CLOSE DATABASE
04    DATABASE stores2
05    CLOSE DATABASE
06 END MAIN
```

CONNECT TO

Purpose:

Opens a new database session in multi-session mode.

Syntax:

```
CONNECT TO { dbname | DEFAULT } [ AS session ]
    [ USER username USING password ]
    [ WITH CONCURRENT TRANSACTION ]
```

Notes:

1. *dbname* is a string expression identifying the database specification.
2. *session* is a string expression identifying the database session. By default, it is *dbname*.
3. *username* is a string expression identifying the name of the database user.
4. *password* is a string expression identifying the password of the database user.

Usage:

The `CONNECT TO` instruction opens a database connection. If the instruction successfully connects to the database environment, the connection becomes the *current database session* for the application.

An application can connect to several database environments at the same time, and it can establish multiple connections to the same database environment, provided each connection has a unique connection name. If you need only one connexion to a database, you can use the DATABASE instruction.

With Informix database servers, when using the `DEFAULT` keyword, you connect to the default Informix database server, identified by the INFORMIXSERVER environment variable, without any database selection.

By default the database user is identified by the current operating system user, but it can be authenticated according to database specification parameters.

When the `USER username USING password` clause is specified, the database user is identified by *username* and *password*, ignoring all other settings defined by the database specification. See also Database user authentication.

The `WITH CONCURRENT TRANSACTION` clause allows a program to open several transactions concurrently in different database sessions.

Warnings:

1. The session name is case-sensitive.
2. You cannot include a `CONNECT TO` statement within a PREPARE instruction.
3. When using Informix databases on UNIX, the only restriction on establishing multiple connections to the same database environment is that an application can establish only one connection to each local server that uses the shared-memory connection mechanism. To find out whether a local server uses the shared-memory connection mechanism or the local-loopback connection mechanism, examine the **$INFORMIXDIR/etc/sqlhosts** file.

Example:

```
01 MAIN
02    CONNECT TO "stores1" -- Session name is "stores1"
03    CONNECT TO "stores1" AS "SA" -- Session name is "SA"
04    CONNECT TO "stores2" AS "SB" USER "scott" USING "tiger"
05 END MAIN
```

SET CONNECTION

Purpose:

Selects the current session when in multi-session mode.

Syntax:

```
SET CONNECTION { { session | DEFAULT } [DORMANT] | CURRENT DORMANT }
```

Notes:

1. *session* is a string expression identifying the name of the database session to be set as current.

Usage:

The `SET CONNECTION` instruction make a given connection current.

When using the `DEFAULT` keyword, it identifies the default database server connection established with a `CONNECT TO DEFAULT` or a `DATABASE` instruction. **Informix only!**

To make the current connection dormant, use `CURRENT DORMANT` keyword. **Informix only!**

Warnings:

1. The session name is case-sensitive.
2. You cannot include a `SET CONNECTION` statement within a PREPARE instruction.
3. The `CURRENT DORMANT` option is only supported for compatibility with Informix; there is no need to make a connection dormant in FGL programs.

Example:

```
01 MAIN
02    CONNECT TO "stores1"
03    CONNECT TO "stores1" AS "SA"
04    CONNECT TO "stores2" AS "SB"
05    SET CONNECTION "stores1"    -- Select first session
06    SET CONNECTION "SA"         -- Select second session
07    SET CONNECTION "stores1"    -- Select first session again
08 END MAIN
```

DISCONNECT

Purpose:

Terminates database sessions when in multi-session mode.

Syntax:

```
DISCONNECT { ALL | CURRENT | session }
```

Notes:

1. *session* is a string expression identifying the name of the database session to be terminated.

Usage:

The `DISCONNECT` instruction closed a given database connection.

When using the `DEFAULT` keyword, it identifies the default database server connection established with a `CONNECT TO DEFAULT` or a `DATABASE` instruction. **Informix only!**

Use the ALL keyword to terminate all opened connections. From that point, you must establish a new connection to execute SQL statements.

Use the CURRENT keyword to terminate the current connection only. From that point, you must select another connection or establish a new connection to execute SQL statements.

Warnings:

1. The session name is case-sensitive.
2. You cannot include a DISCONNECT statement within a PREPARE instruction.
3. If a DISCONNECT statement is used while a transaction is active, it is automatically rolled back.

Example:

```
01 MAIN
02    CONNECT TO "stores1"
03    CONNECT TO "stores1" AS "SA"
04    CONNECT TO "stores2" AS "SB" USER "scott" USING "tiger"
05    -- SB is the current database session
06    DISCONNECT "stores1" -- Continue with SB
07    DISCONNECT "SB" -- SB is no longer the current session
08    SET CONNECTION "SA" -- Select second session
09 END MAIN
```

Database Transactions

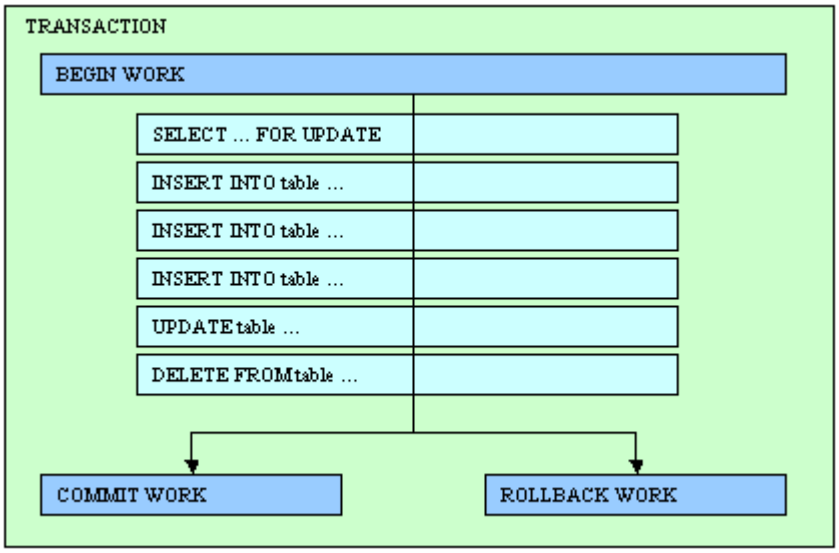
Summary:

- What is a database transaction?
- Transaction Management Model
- Starting a transaction (`BEGIN WORK`)
- Validating a transaction (`COMMIT WORK`)
- Cancelling a transaction (`ROLLBACK WORK`)
- Setting the Isolation Level (`SET ISOLATION`)
- Setting the Lock Mode (`SET LOCK MODE`)
- Examples

See also: Connections, Static SQL, Dynamic SQL, Result Sets, SQL Errors, Programs.

What is a database transaction?

A Database Transaction delimits a set of database operations that are processed as a whole. Database operations included inside a transaction are validated or canceled as a unique operation.



The database server is in charge of *Data Concurrency* and *Data Consistency* control. Data Concurrency control allows the simultaneous access of the same data by many users, while Data Consistency control gives each user a consistent view of the database.

Without adequate concurrency and consistency controls, data could be changed improperly, compromising data integrity. If you want to write applications that can work with different kinds of database servers, you must adapt the program logic to the

behavior of the database servers regarding concurrency and consistency management. This requires good knowledge of multi-user application programming, transactions, locking mechanisms, isolation levels and wait mode. If you are not familiar with these concepts, carefully read the documentation of each database server that covers this subject.

Usually, database servers set exclusive locks on rows that are modified or deleted inside a transaction. These locks are held until the end of the transaction to control concurrent access to that data. Some database servers like Oracle implement row versioning (before modifying a row, the server makes a copy). This technique allows readers to see a consistent copy of the rows that are updated during a transaction not yet committed. When the isolation level is high (Repeatable Read) or when using a `SELECT FOR UPDATE` statement, the database server sets shared locks on read rows to prevent other users from changing the data fetched by the reader. Again, these locks are held until the end of the transaction. Some database servers like Informix allow read locks to be held regardless of the transactions (`WITH HOLD` cursor option), but this is not a standard.

Processes accessing the database can change transaction parameters such as the isolation level or lock wait mode. The main problem is to find a configuration which results in similar behavior on every database engine. Programs using Informix-specific behavior must be adapted to work with other database servers.

Here is the recommended configuration to get common behavior with all kinds of database engines:

- The database must support transactions; this is usually the case.
- Transactions must be as short as possible (a few seconds).
- The Isolation Level must be at least "Committed Read" (= "Cursor Stability").
- The Wait Mode for locks must be "WAIT" or "WAIT n" (timeout).

When using this configuration, the locking granularity does not have to be set at the row level. For example, to improve performance with Informix databases, you can use the "LOCK MODE PAGE" locking level, which is the default.

A lot of applications have been developed for old Informix SE databases that do not manage transaction logging. These applications often work in the default lock wait mode which is "NOT WAIT". Additionally, applications using databases without transactions usually do not change the isolation level, which defaults to "Dirty Read". You must review the program logic of these applications in order to conform to the portable configuration.

Transaction Management Model

To write portable SQL applications, programmers use the instructions described in this section to delimit transaction blocks and define concurrency parameters such as the isolation level and the lock wait mode. At runtime, the database driver generates the appropriate SQL commands to be used with the target database server.

If you initiate a transaction with a `BEGIN WORK` statement, you must issue a `COMMIT WORK` statement at the end of the transaction. If you fail to issue the `COMMIT WORK` statement, the database server rolls back any modifications that the transaction made to the database. If you do not issue a `BEGIN WORK` statement to start a transaction, each statement executes within its own transaction. These single-statement transactions do not require either a `BEGIN WORK` statement or a `COMMIT WORK` statement.

For historical reasons, the language is based on IBM Informix SQL language, which defines the transaction management instructions. IBM Informix database servers can work in different transaction logging modes:

1. **Native**, without logging
2. **Native**, non-buffered logging
3. **Native**, buffered logging
4. **ANSI**, buffered logging

The first mode does not allow transaction management and should be avoided. In the second and third modes, you can use the `BEGIN WORK`, `COMMIT WORK` and `ROLLBACK WORK` statements. In **ANSI** mode, you can only use the `COMMIT` and `ROLLBACK` statements, because transactions are implicit.

When using Informix databases, the type of logging defines the way you manage transactions in your programs. For example, when using an ANSI-compliant Informix database, you do not have to start transactions with `BEGIN WORK`, since these are implicit.

When using the Standard Database Interface (SDI) architecture, you are free to use any type of transaction logging with Informix databases. When using the Open Database Interface (ODI) architecture you are free to use the native transaction management statements supported by the underlying database server, but it is recommended that you follow the default (native) Informix logging, by using `BEGIN WORK`, `COMMIT WORK` and `ROLLBACK WORK` to manage transactions. At runtime, the database drivers can manage the execution of the appropriate instructions for the target database server. This allows you to use the same source code for different kinds of database servers.

The instructions described in this section must be executed as Static SQL statements. Even if it is supported by the Informix API, it is not recommended that you use the Dynamic SQL instructions to `PREPARE` and `EXECUTE` transaction management statements, because it can result in unexpected behavior when using other database servers.

BEGIN WORK

Purpose:

Starts a database transaction in the current connection.

Syntax:

```
BEGIN WORK
```

Usage:

Use this instruction to indicate where the database transaction starts in your program. If supported by the database server, the underlying database driver starts a transaction. Each row that an UPDATE, DELETE, or INSERT statement affects during a transaction is locked and remains locked throughout the transaction. When using a non-Informix database, the ODI driver executes the native SQL statement corresponding to `BEGIN WORK`.

Warnings:

1. Some database servers do not support a Data Definition Language statement (like `CREATE TABLE`) inside transactions, or even auto-commit the transaction when such a statement is executed. Therefore, it is strongly recommended that you avoid DDL statements inside transactions.
2. A transaction that contains statements that affect many rows can exceed the limits that your operating system or the database server configuration imposes on the maximum number of simultaneous locks.

Tips:

1. Include a limited number of SQL operations in a transaction to execute short transactions. In a standard database session configuration (wait mode), it is not recommended that you have a transaction block running a long time, since it may block concurrent processes which want to access the same data.

COMMIT WORK

Purpose:

Validates and terminates a database transaction in the current connection.

Syntax:

```
COMMIT WORK
```

Usage:

Use this instruction to commit all modifications made to the database from the beginning of a transaction. The database server takes the required steps to make sure that all modifications that the transaction makes are completed correctly and saved to disk. The `COMMIT WORK` statement releases all exclusive locks. With some databases like Informix, shared locks are not released if the `FOR UPDATE` cursor is declared `WITH HOLD` option.

The `COMMIT WORK` statement closes all cursors not declared with the `WITH HOLD` option. When using a non-Informix database, the ODI driver executes the native SQL statement corresponding to `COMMIT WORK`.

ROLLBACK WORK

Purpose:

Cancels and terminates a database transaction in the current connection.

Syntax:

```
ROLLBACK WORK
```

Usage:

Use this instruction to cancel the current transaction and invalidate all changes since the beginning of the transaction. After the execution of this instruction, the database is restored to the state that it was in before the transaction began. All row and table locks that the canceled transaction holds are released. If you issue this statement when no transaction is pending, an error occurs. When using a non-Informix database, the ODI driver executes the native SQL statement corresponding to `ROLLBACK WORK`.

Warnings:

1. Normally, the `ROLLBACK WORK` statement closes all cursors not declared with the `WITH HOLD` option. This is not the case with some databases like IBM DB2, which closes all kind of cursors when doing a `ROLLBACK`.

SET ISOLATION

Purpose:

Defines the transaction isolation level for the current connection.

Syntax:

```
SET ISOLATION TO
  { DIRTY READ
  | COMMITTED READ
  | CURSOR STABILITY
  | REPEATABLE READ }
```

Usage:

Sets the isolation level for the current connection. See database concepts in your database server documentation for more details about isolation levels and concurrency management.

When using a non-Informix database, the ODI driver executes the native SQL statement that corresponds to the specified isolation level.

Warnings:

1. When using the `DIRTY READ` isolation level, the database server might return a phantom row, which is an uncommitted row that was inserted or modified within a transaction that has subsequently rolled back. No other isolation level allows access to a phantom row.

Tips:

1. On most database servers, the default isolation level is usually `COMMITTED READ`, which is appropriate to portable database programming. Therefore, we do not recommend that you change the isolation level.

SET LOCK MODE

Purpose:

Defines the behavior of the program that tries to access a locked row or table.

Syntax:

```
SET LOCK MODE TO { NOT WAIT | WAIT [ seconds ] }
```

Notes:

1. This instruction defines the timeout for lock acquisition for the current connection.
2. When possible, the underlying database driver sets the corresponding connection parameter to define the timeout for lock acquisition. But some database servers may not support setting the lock timeout parameter. In this case, the runtime system generates an exception.
3. When using the `NOT WAIT` clause, the timeout is set to zero. If the resource is locked, the database server ends the operation immediately and returns an SQL Error.
4. *seconds* defines the number of seconds to wait for lock acquisition. If the resource is locked, the database server ends the operation after the elapsed time and returns an SQL Error.
5. When using the `WAIT` clause without a number of seconds, the database server waits for lock acquisition for an infinite time.

6. On most database servers, the default is to wait for locks to be released.

Warnings:

1. Make sure that the database server and corresponding database driver both support a lock acquisition timeout option, otherwise the program would generate an exception. For example, the IBM DB2 V8.1 database server does not support this option at the session level.

Examples

Example 1:

```
01 MAIN
02    DATABASE stock
03    BEGIN WORK
04    INSERT INTO items VALUES ( ... )
04    UPDATE items SET ...
05    COMMIT WORK
06 END MAIN
```

Static SQL Statements

Summary:

- What are Static SQL Statements?
- Using program variables in Static SQL
- Table and column names in Static SQL
- What SQL string was generated by the compiler?
- Supported Static SQL Statements
- Adding rows (`INSERT`)
- Deleting rows (`DELETE`)
- Updating rows (`UPDATE`)
- Selecting rows (`SELECT`)

See also: Transactions, Positioned Updates, Dynamic SQL, Result Sets, SQL Errors.

What are Static SQL Statements?

Static SQL Statements are SQL instructions that are a part of the language syntax. Static SQL Statements can be used directly in the source code as a normal procedural instruction. The static SQL statements are parsed and validated at compile time. At runtime, these SQL statements are automatically prepared and executed by the runtime system.

Program variables are detected by the compiler and handled as SQL parameters.

The following example defines two variables that are directly used in an `INSERT` statement:

```
01 MAIN
02    DEFINE iref INTEGER, name CHAR(10)
03    DATABASE stock
04    LET iref = 65345
05    LET name = "Kartopia"
06    INSERT INTO item ( item_ref, item_name ) VALUES ( iref, name )
07    SELECT item_name INTO name
08      FROM item WHERE item_ref = iref
09 END MAIN
```

Using Static SQL Statements clarifies the source code (you do not need to use Dynamic SQL Instructions to prepare and execute the SQL statement), but you cannot modify the SQL text at runtime.

A limited number of SQL statements is directly supported in the language (see below), but most common statements like `INSERT`, `UPDATE`, `DELETE`, `SELECT` can be executed without problems using a simple standard syntax.

Using program variables in Static SQL statements

The syntax of Static SQL statements supports the usage of program variables directly as SQL parameters. This gives a better understanding of the source code and requires less lines as using SQL parameters with Dynamic SQL:

```
01 MAIN
02    DEFINE c_num INTEGER
03    DEFINE c_name CHAR(10)
04    DATABASE stock
05    SELECT cust_name INTO c_name FROM customer WHERE cust_num = c_num
06 END MAIN
```

If a database column name conflicts with a program variable, you can use the @ sign as the column prefix. The compiler will treat the identifier following the @ as a table column:

```
01 MAIN
02    DEFINE cust_name CHAR(10)
03    DEFINE cnt INTEGER
04    DATABASE stock
05    SELECT COUNT(*) INTO cnt FROM customer WHERE @cust_name =
cust_name
06 END MAIN
```

The @ sign will not figure in the resulting SQL statement stored in the **42m** module.

Table and column names in Static SQL

In Static SQL, table and column names will be converted to lowercase by the fglcomp compiler. The SQL keywords are always converted to uppercase.

For example:

```
01  UPDATE CUSTOMER set CUST_name = 'undef' WHERE cust_name is null
```

will be converted to:

```
UPDATE customer SET cust_name = 'undef' WHERE cust_name IS NULL
```

While SQL keywords are not case sensitive for database servers, table names and column names can be case-sensitive.

For more details, see Naming database objects.

What SQL string was generated by the compiler?

As described in the above sections, the fglcomp compiler parses the Static SQL statements and modifies them before writing the resulting SQL text to the **42m** module.

You can extract all SQL statements from the source by using the -S option of fglcomp:

```
01 MAIN
02    DEFINE c_name CHAR(10)
03    DEFINE cnt INTEGER
04    DATABASE stock
05    SELECT COUNT(*) INTO cnt FROM customer WHERE customer.cust_name =
c_name
06 END MAIN
$ fglcomp -S test.4gl
test.4gl^5^SELECT COUNT(*) FROM customer WHERE cust_name = ?
```

Supported Static SQL Statements

The following table shows all SQL statements supported by the language as Static SQL Statements.

Lines marked with a pink background show SQL statements that are specific to IBM Informix SQL language. These are supported for backward compatibility with the IBM Informix 4GL compiler, and it is not recommended that you use them in your programs if you want to write portable SQL. Other statements can be used, as long as you use standard SQL syntax.

SQL Statement	Description
`ALTER INDEX ...`	Modify the definition of an index.
`ALTER TABLE ...`	Modify the definition of a table.
`ALTER SEQUENCE ...`	Modify the definition of a sequence.
`CREATE AUDIT ...`	Create audit recording for a given table.
`CREATE DATABASE ...`	Create a database.
`CREATE INDEX ...`	Create an index.
`CREATE TABLE ...`	Create a table.
`CREATE SEQUENCE ...`	Create a sequence.
`CREATE SYNONYM ...`	Create a synonym for a database table or view.
`CREATE TEMP TABLE ...`	Create a temporary table.
`CREATE VIEW ...`	Create a view.
`DELETE FROM ...`	Delete rows in a table.
`DROP AUDIT ...`	Remove audit for a given table.
`DROP INDEX ...`	Delete an index.
`DROP SEQUENCE ...`	Delete a sequence.

`DROP SYNONYM ...`	Delete a table or view synonym.
`DROP TABLE ...`	Delete a table.
`DROP VIEW ...`	Delete a view.
`GRANT ...`	Grant access rights.
`INSERT INTO ...`	Insert rows into a table.
`RECOVER TABLE ...`	Re-build an SE database table from log files.
`RENAME COLUMN ...`	Rename a table column.
`RENAME INDEX ...`	Rename an index.
`RENAME SEQUENCE ...`	Rename a sequence.
`RENAME TABLE ...`	Rename a table.
`REVOKE ...`	Revoke access rights.
`ROLLFORWARD DATABASE ...`	Put an SE database in a safe state.
`SELECT ...`	Select rows from a table.
`SELECT ... INTO TEMP` *`ttab`*	Create a temporary table from a result set.
`SET EXPLAIN ...`	Enable/disable query execution plan trace.
`SET LOG ...`	Set the logging of an On-line database.
`START DATABASE ...`	Initialize an SE database.
`TRUNCATE TABLE table ...`	Cleanup a table without logging changes (no rollback possible)
`UPDATE table ...`	Update rows in a table.
`UPDATE STATISTICS ...`	Collect statistics information for the query optimizer.

INSERT

Purpose:

Inserts a row in a table in the current database session.

Syntax:

```
INSERT INTO table-specification [ ( column [,...] ) ]
{
  VALUES ( { variable | literal | NULL } [,...] )
|
  select-statement
}
```

where *table-specification* is:

```
[dbname[@dbserver]:][owner.]table
```

Notes:

1. *dbname* identifies the database name. **Informix only!**
2. *dbserver* identifies the Informix database server (INFORMIXSERVER). **Informix only!**
3. *owner* identifies the owner of the table, with optional double quotes. **Informix only!**
4. *table* is the name of the database table.
5. *column* is a name of a table column.
6. *variable* is a program variable, a record or an array used as a parameter buffer to provide values.
7. When you use records, you can specify all record members with the star notation (`rec.*`).
8. *literal* is any literal expression supported by the language.
9. *select-statement* is a static SELECT statement with or without parameters as variables.
10. When you use the `VALUES` clause, the statement inserts a row in the table with the values specified in variables, as literals, or with `NULL`.
11. When you use a *select-statement*, the statement insert all rows returned in the result set of the SELECT statement.

Warnings:

1. For portability, it is not recommended that you use the *select-statement* syntax.
2. When you use a *select-statement*, the columns returned by the result set must match the column number and data types of the target table.

Example:

```
01 MAIN
02    DEFINE myrec RECORD
03              key INTEGER,
04              name CHAR(10),
05              cdate DATE,
06              comment VARCHAR(50)
07           END RECORD
08    DATABASE stock
09    LET myrec.key     = 123
10    LET myrec.name    = "Katos"
11    LET myrec.cdate   = TODAY
12    LET myrec.comment = "xxxxxx"
13    INSERT INTO items VALUES ( 123, 'Practal', NULL, myrec.comment )
14    INSERT INTO items VALUES ( myrec.* )
15    INSERT INTO items SELECT * FROM histitems WHERE name = myrec.name
16 END MAIN
```

UPDATE

Purpose:

Updates rows of a table in the current database session.

Syntax 1:

```
UPDATE table-specification
   SET
       column = { variable | literal | NULL }
       [,...]
   [ WHERE { condition | CURRENT OF cursor } ]
```

Syntax 2: Informix only!

```
UPDATE table-specification
   SET { ( [table.]* ) | ( column [,...] ) }
     = ( { variable | literal | NULL } [,...] )
   [ WHERE { condition | CURRENT OF cursor } ]
```

where *table-specification* is:

```
[dbname[@dbserver]:][owner.]table
```

Notes:

1. *dbname* identifies the database name. **Informix only!**
2. *dbserver* identifies the Informix database server (INFORMIXSERVER). **Informix only!**
3. *owner* identifies the owner of the table, with optional double quotes. **Informix only!**
4. *table* is the name of the database table.
5. *column* is a name of a table column.
6. *column* can be specified with a sub-script expression (`column[a,b]`). **Informix only!**
7. *variable* is a program variable, a record or an array used as a parameter buffer to provide values.
8. *literal* is any literal expression supported by the language.
9. *condition* is an SQL expression to select the rows to be updated.
10. *cursor* is the identifier of a database cursor.
11. For more details about the `WHERE CURRENT OF` clause, see Positioned Updates.

Warnings:

1. *column* with a sub-script expression (`column[a,b]`) is not recommended because most database servers do not support this notation.
2. Although a few database servers support **Syntax 2,** it is strongly recommended that you use **Syntax 1** only.

Example:

```
01 MAIN
02    DEFINE myrec RECORD
03              key INTEGER,
04              name CHAR(10),
05              cdate DATE,
06              comment VARCHAR(50)
07           END RECORD
08    DATABASE stock
09    LET myrec.key     = 123
10    LET myrec.name    = "Katos"
11    LET myrec.cdate   = TODAY
12    LET myrec.comment = "xxxxxx"
13    UPDATE items SET
14         name    = myrec.name,
15         cdate   = myrec.cdate,
16         comment = myrec.comment
17     WHERE key = myrec.key
18 END MAIN
```

DELETE

Purpose:

Deletes rows from a table in the current database session.

Syntax:

```
DELETE FROM table-specification
  [ WHERE { condition | CURRENT OF cursor } ]
```

where *table-specification* is:

```
[dbname[@dbserver]:][owner.]table
```

Notes:

1. *dbname* identifies the database name. **Informix only!**
2. *dbserver* identifies the Informix database server (INFORMIXSERVER). **Informix only!**
3. *owner* identifies the owner of the table, with optional double quotes. **Informix only!**
4. *table* is the name of the database table.
5. *condition* is an SQL expression to select the rows to be deleted.
6. *cursor* is the identifier of a database cursor.
7. For more details about the `WHERE CURRENT OF` clause, see Positioned Updates.

Warnings:

1. If you do not specify the `WHERE` clause, all rows in the table will be deleted.

Example:

```
01 MAIN
02    DATABASE stock
03    DELETE FROM items WHERE name LIKE 'A%'
04 END MAIN
```

SELECT

Purpose:

Selects rows from one or more tables in the current database session.

Syntax:

```
select-statement [ UNION [ALL] select-statement ] [...]
```

where *select-statement* is:

```
SELECT [{ALL|DISTINCT|UNIQUE}] { * | select-list }
  [ INTO variable [,...] ]
  FROM table-list [,...]
  [ WHERE condition ]
  [ GROUP BY column-list [ HAVING condition ] ]
  [ ORDER BY column [{ASC|DESC}] [,...] ]
```

where *select-list* is:

```
{ [@]table-specification.*
| [table-specification.]column
| literal
} [ [AS] column-alias ]
[,...]
```

where *table-list* is:

```
{ table-name
| OUTER table-name
| OUTER ( table-name [,...] )
}
[,...]
```

where *table-name* is:

```
table-specification [ [AS] table-alias ]
```

where *table-specification* is:

```
[dbname[@dbserver]:][owner.]table
```

where *column-list* is:

```
column-name [,...]
```

where *column-name* is:

```
[table.]column
```

Notes:

1. *dbname* identifies the database name. **Informix only!**
2. *dbserver* identifies the Informix database server (INFORMIXSERVER). **Informix only!**
3. *owner* identifies the owner of the table, with optional double quotes. **Informix only!**
4. *table* is the name of the database table.
5. *table-alias* defines a new name to reference the *table* in the rest of the statement.
6. *column* is a name of a table column.
7. *column-alias* defines a new name to reference the *column* in the rest of the statement.
8. *condition* is an SQL expression to select the rows to be deleted.
9. The `INTO` clause provides the list of fetch buffers. This clause is not part of the SQL language sent to the database server; it is extracted from the statement by the compiler.

Warnings:

1. The language supports the `SELECT INTO TEMP` statement to create temporary tables (this statement does not return a result set).

Usage:

If the `SELECT` statement returns only one row of data, you can write it directly as a procedural instruction. However, you must use the `INTO` clause to provide the list of variables where column values will be fetched:

```
01 MAIN
02    DEFINE myrec RECORD
03              key INTEGER,
04              name CHAR(10),
05              cdate DATE,
06              comment VARCHAR(50)
07           END RECORD
08    DATABASE stock
09    LET myrec.key = 123
10    SELECT name, cdate
11      INTO myrec.name, myrec.cdate
```

```
12      FROM items
13      WHERE key=myrec.key
14 END MAIN
```

If the SELECT statement returns more than one row of data, you must declare a database cursor to process the result set:

```
01 MAIN
02    DEFINE myrec RECORD
03              key INTEGER,
04              name CHAR(10),
05              cdate DATE,
06              comment VARCHAR(50)
07           END RECORD
08    DATABASE stock
09    LET myrec.key = 123
10    DECLARE c1 CURSOR FOR
11      SELECT name, cdate
12        FROM items
13        WHERE key=myrec.key
14    OPEN c1
15    FETCH c1 INTO myrec.name, myrec.cdate
16    CLOSE c1
17 END MAIN
```

The SELECT statement can include the INTO clause, but it is strongly recommended that you use that clause in the FETCH instruction only.

See Result Sets Processing for more details.

Dynamic SQL Management

Summary:

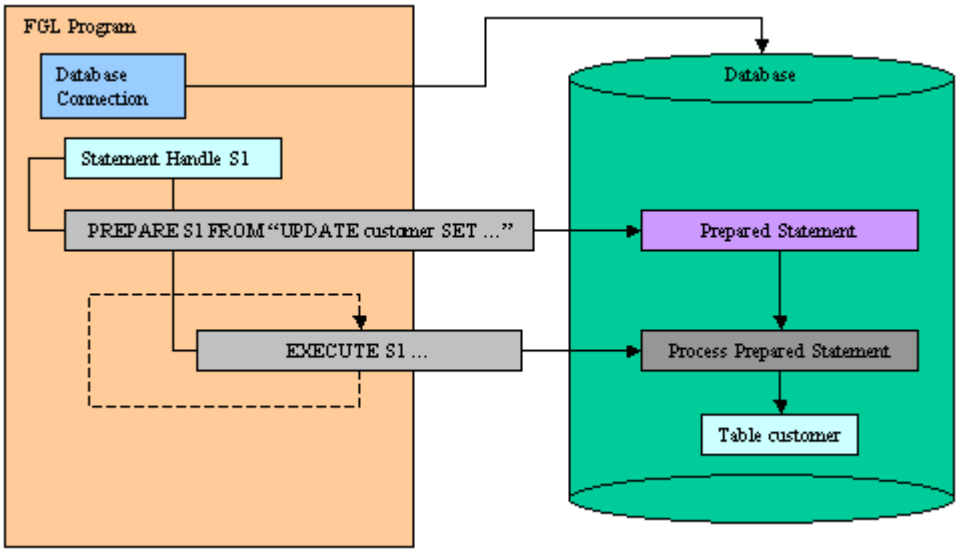
- What is Dynamic SQL Management?
- Preparing an SQL statement (`PREPARE`)
- Executing prepared statements (`EXECUTE`)
- Releasing prepared statements (`FREE`)
- Immediate execution (`EXECUTE IMMEDIATE`)

See also: Transactions, Positioned Updates, Static SQL, Result Sets, SQL Errors, Declaring a cursor (`DECLARE`).

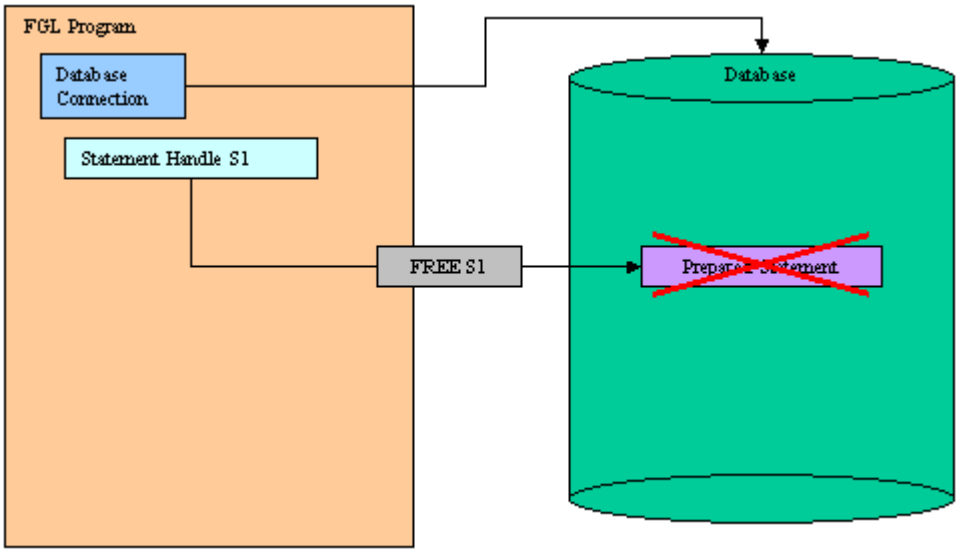
What is Dynamic SQL management?

BDL includes basic SQL instructions in the language syntax (see Static SQL), but only a limited number of SQL instructions are supported this way. Dynamic SQL Management allows you to execute any kind of SQL statement, hard coded or created at runtime, with or without SQL parameters, returning or not returning a result set.

In order to execute an SQL statement with Dynamic SQL, you must first prepare the SQL statement to initialize a statement handle, then you execute the prepared statement one or more times:



When you no longer need the prepared statement, you can free the statement handle to release allocated resources:



When using insert cursors or SQL statements that produce a result set (like `SELECT`), you must declare a cursor with a prepared statement handle.

Prepared SQL statements can contain SQL parameters by using `?` placeholders in the SQL text. In this case, the `EXECUTE` or `OPEN` instruction supplies input values in the `USING` clause.

To increase performance efficiency, you can use the `PREPARE` instruction, together with an `EXECUTE` instruction in a loop, to eliminate overhead caused by redundant parsing and optimizing. For example, an `UPDATE` statement located within a `WHILE` loop is parsed each time the loop runs. If you prepare the `UPDATE` statement outside the loop, the statement is parsed only once, eliminating overhead and speeding statement execution.

PREPARE

Purpose:

This instruction prepares an SQL statement for execution in the current database connection.

Syntax:

```
PREPARE sid FROM sqltext
```

Notes:

1. *sid* is an identifier to handle the prepared SQL statement.
2. *sqltext* is a string expression containing the SQL statement to be prepared.

Usage:

The PREPARE instruction allocates resources for an SQL statement handle, in the context of the current connection. The SQL text is sent to the database server for parsing, validation and to generate the execution plan.

Prepared SQL statements can be executed with the EXECUTE instruction, or, when the SQL statement generates a result set, the prepared statement can be used to declare cursors with the DECLARE instruction.

A statement identifier (*sid*) can represent only one SQL statement at a time. You can execute a new PREPARE instruction with an existing statement identifier if you wish to assign the text of a different SQL statement to the statement identifier. The scope of reference of the *sid* statement identifier is local to the module where it is declared.

The SQL statement can have parameter placeholders, identified by the question mark (?) character.

Resources allocated by PREPARE can be released later by the FREE instruction.

Warnings:

1. You cannot directly reference a variable in the text of a prepared SQL statement; you must use question mark (?) placeholders instead.
2. The number of prepared statements in a single program is limited by the database server and the available memory. Make sure that you free the resources when you no longer need the prepared statement.
3. The identifier of a statement that was prepared in one module cannot be referenced from another module.
4. You cannot use question mark (?) placeholders for SQL identifiers such as a table name or a column name; you must specify these identifiers in the statement text when you prepare it.
5. Some database servers like Informix support multiple SQL statement preparation in a unique PREPARE instruction, but most database servers avoid multiple statements.

Example:

```
01 FUNCTION deleteOrder(n)
02   DEFINE n INTEGER
03   PREPARE s1 FROM "DELETE FROM order WHERE key=?"
04   EXECUTE s1 USING n
05   FREE s1
06 END FUNCTION
```

EXECUTE

Purpose:

This instruction runs an SQL statement previously prepared in the same database connection.

Syntax:

```
EXECUTE sid [ USING pvar {IN|OUT|INOUT} [,...] ] [ INTO fvar [,...] ]
```

Notes:

1. *sid* is an identifier to handle the prepared SQL statement.
2. *pvar* is a variable containing an input value for an SQL parameter.
3. *fvar* is a variable used as fetch buffer, when the prepared statement returns a single database row.

Usage:

The `EXECUTE` instruction performs the execution of a prepared SQL statement. Once prepared, an SQL statement can be executed as often as needed.

If the SQL statement has (`?`) parameter placeholders, you must specify the `USING` clause to provide a list of variables as parameter buffers. Parameter values are assigned by position.

If the SQL statement returns a result set with one row, you can specify the `INTO` clause to provide a list of variables to receive the result set column values. Fetched values are assigned by position. If the SQL statement returns a result set with more than one row, the instruction raises an exception.

The `IN`, `OUT` or `INOUT` options can be used to call stored procedures having input / output parameters. Use the `IN`, `OUT` or `INOUT` options to indicate if a parameter is respectively for input, output or both. For more details about stored procedure calls, see SQL Programming.

Warnings:

1. You cannot use strings or numeric constants in the `USING` or `INTO` list. All elements must be program variables.
2. You cannot execute a prepared SQL statement based on database tables if the table structure has changed (`ALTER TABLE`) since the PREPARE instruction; you must re-prepare the SQL statement.
3. The `IN`, `OUT` or `INOUT` options can only be used for simple variables, you cannot specify those options for a complete record with the record.* notation.

Example:

```
01 MAIN
02   DEFINE var1 CHAR(20)
03   DEFINE var2 INTEGER
04
05   DATABASE stores
06
07   PREPARE s1 FROM "UPDATE tab SET col=? WHERE key=?"
08   LET var1 = "aaaa"
09   LET var2 = 345
10   EXECUTE s1 USING var1, var2
11
12   PREPARE s2 FROM "SELECT col FROM tab WHERE key=?"
13   LET var2 = 564
14   EXECUTE s2 USING var2 INTO var1
15
16   PREPARE s3 FROM "CALL myproc(?,?)"
17   LET var1 = 'abc'
18   EXECUTE s3 USING var1 IN, var2 OUT
19
20 END MAIN
```

FREE

Purpose:

This instruction releases the resources allocated to a prepared statement.

Syntax:

```
FREE sid
```

Notes:

1. *sid* is the identifier of the prepared SQL statement.

Usage:

The `FREE` instruction takes the name of a statement as parameter.

All resources allocated to the SQL statement handle are released.

Warnings:

1. After resources are released, the statement identifier cannot be referenced by a cursor, or by the EXECUTE statement, until you prepare the statement again.

Tips:

1. Free the statement if it is not needed anymore, this saves resources on the database client and database server side.

Example:

```
01 FUNCTION update_customer_name( key, name )
02   DEFINE key INTEGER
03   DEFINE name CHAR(10)
04   PREPARE s1 FROM "UPDATE customer SET name=? WHERE customer_num=?"
05   EXECUTE s1 USING name, key
06   FREE s1
07 END FUNCTION
```

EXECUTE IMMEDIATE

Purpose:

This instruction performs a simple SQL execution without SQL parameters or result set.

Syntax:

```
EXECUTE IMMEDIATE sqltext
```

Notes:

1. *sqltext* is a string expression containing the SQL statement to be executed.

Usage:

The `EXECUTE IMMEDIATE` instruction passes an SQL statement to the database server for execution in the current database connection.

The SQL statement must be a single statement without parameters, returning no result set.

This instruction performs the functions of PREPARE, EXECUTE and FREE in one step.

Warnings:

1. The SQL statement cannot contain SQL parameters.
2. The SQL statement must not produce a result set.

Example:

```
01 MAIN
```

```
02   DATABASE stores
03   EXECUTE IMMEDIATE "UPDATE tab SET col='aaa' WHERE key=345"
04 END MAIN
```

Database Result Set Processing (Cursor)

Summary:

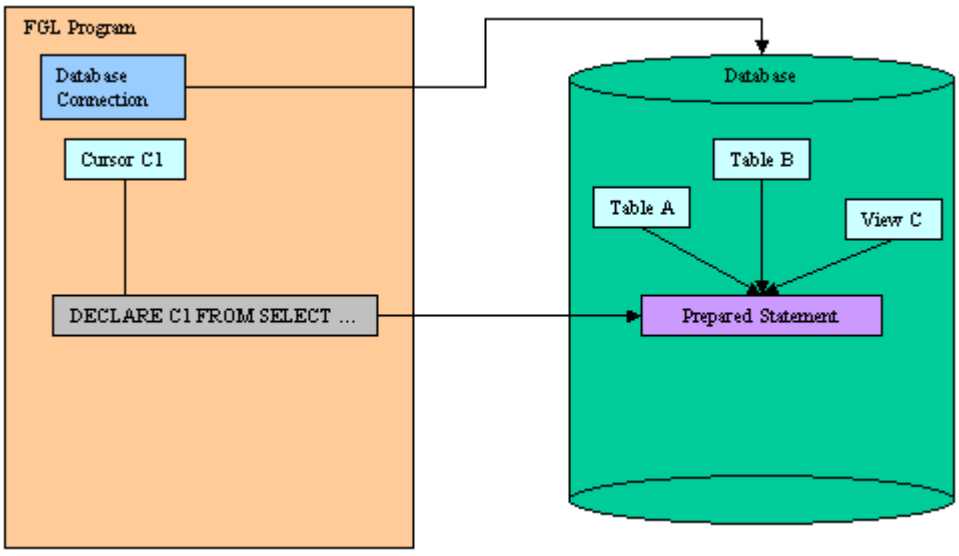
- What is a database result set?
- Cursor declaration (`DECLARE`)
- Opening cursors (`OPEN`)
- Retrieving data (`FETCH`)
- Closing cursors (`CLOSE`)
- Freeing cursors (`FREE`)
- Browsing rows in a loop (`FOREACH`)

See also: Transactions, Positioned Updates, Static SQL, Dynamic SQL, SQL Errors.

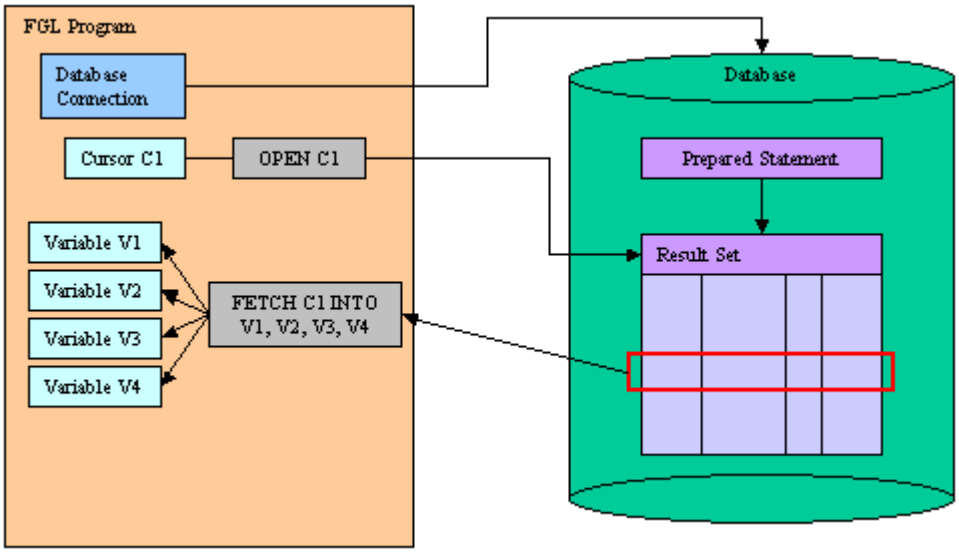
What is a database result set?

A Database Result Set is a set of rows generated by an SQL statement that produces rows, such as `SELECT`. The result set is maintained by the database server. In a program, you handle a result set with a Database Cursor.

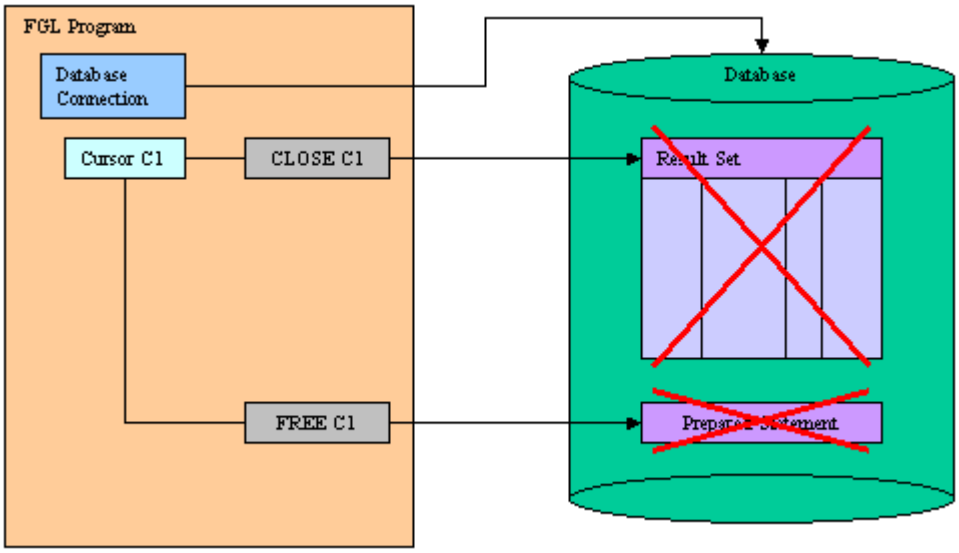
First you must declare the database cursor with the DECLARE instruction. This instruction sends the SQL statement to the database server for parsing, validation and to generate the execution plan.



The result set is produced after execution of the SQL statement, when the database cursor is associated with the result set by the OPEN instruction. At this point, no data rows are transmitted to the program. You must use the FETCH instruction to retrieve data rows from the database server.



When finished with the result set processing, you must CLOSE the cursor to release the resources allocated for the result set on the database server. The cursor can be re-opened if needed. If the SQL statement is no longer needed, you can free the resources allocated to statement execution with the FREE instruction.



The scope of reference of a database cursor is local to a module, so a cursor that was declared in one source file cannot be referenced in a statement in another file.

The language supports *sequential* and *scrollable* cursors. Sequential cursors, which are unidirectional, are used to retrieve rows for a report, for example. Scrollable cursors allow you to move backwards or to an absolute or relative position in the result set. Specify whether a cursor is scrollable with the `SCROLL` option of the DECLARE instruction.

DECLARE

Purpose:

This instruction associates a database cursor with an SQL statement in the current connection.

Syntax 1: Cursor declared with a static SQL statement.

```
DECLARE cid [SCROLL] CURSOR [WITH HOLD] FOR select-statement
```

Syntax 2: Cursor declared with a prepared statement.

```
DECLARE cid [SCROLL] CURSOR [WITH HOLD] FOR sid
```

Syntax 3: Cursor declared with a string expression.

```
DECLARE cid [SCROLL] CURSOR [WITH HOLD] FROM expr
```

Notes:

1. *cid* is the identifier of the database cursor.
2. *select-statement* is a `SELECT` statement defined in Static SQL.
3. *sid* is the identifier of a prepared SQL statement.
4. *expr* is any expression that evaluates to a string.
5. In all supported syntaxes, you can use the ? question mark as a parameter placeholder.

Warnings:

1. The maximum number of declared cursors in a single program is limited by the database server and the available memory. Make sure that you free the resources when you no longer need the declared cursor.
2. The identifier of a cursor that was declared in one module cannot be referenced from another module.
3. When declaring a cursor with a static *select-statement*, the statement can include an `INTO` clause. However, this is not recommended, to be consistent with prepared statements. If you prepare the statement, you must omit the `INTO` clause in the SQL text provided to the PREPARE instruction and use the `INTO` clause of the FETCH statement to retrieve the values from the result set.
4. You can add the `FOR UPDATE` clause in the `SELECT` statement to declare an update cursor. You can use the update cursor to modify (update or delete) the current row.
5. Use the `WITH HOLD` option carefully, because this feature is specific to IBM Informix servers. Other database servers do not behave as Informix does with this type of cursor. For example, if the `SELECT` is not declared `FOR UPDATE`, most

database servers keep cursors open after the end of a transaction, but IBM DB2 automatically closes all cursors when the transaction is rolled back.

Usage:

The `DECLARE` instruction allocates resources for an SQL statement handle, in the context of the current connection. The SQL text is sent to the database server for parsing, validation and to generate the execution plan.

After declaring the cursor, you can use the OPEN instruction to execute the SQL statement and produce the result set.

`DECLARE` must precede any other statement that refers to the cursor during program execution.

The scope of reference of the *cid* cursor identifier is local to the module where it is declared.

Resources allocated by the `DECLARE` can be released later by the FREE instruction.

Forward only cursors

If you use only the `DECLARE CURSOR` keywords, you create a **sequential cursor**, which can fetch only the next row in sequence from the result set. The sequential cursor can read through the result set only once each time it is opened. If you are using a sequential cursor for a select cursor, on each execution of the FETCH statement, the database server returns the contents of the current row and locates the next row in the result set.

Example 1: Declaring a cursor with a static SELECT statement.

```
01 MAIN
02    DATABASE stores
03    DECLARE c1 CURSOR FOR SELECT * FROM customer
04 END MAIN
```

Example 2: Declaring a cursor with a prepared statement.

```
01 MAIN
02   DEFINE key INTEGER
03   DEFINE cust RECORD
04             num INTEGER,
05             name CHAR(50)
06           END RECORD
07   DATABASE stores
08   PREPARE s1
09      FROM "SELECT customer_num, cust_name FROM customer WHERE
customer_num>?"
10   DECLARE c1 CURSOR FOR s1
11   LET key=101
12   FOREACH c1 USING key INTO cust.*
```

```
13      DISPLAY cust.*
14   END FOREACH
15 END MAIN
```

Scrollable cursors

Use the `DECLARE SCROLL CURSOR` keywords to create a **scrollable cursor**, which can fetch rows of the result set in any sequence. Until the cursor is closed, the database server retains the result set of the cursor in a static data set (for example, in a temporary table like Informix). You can fetch the first, last, or any intermediate rows of the result set as well as fetch rows repeatedly without having to close and reopen the cursor. On a multi-user system, the rows in the tables from which the result set rows were derived might change after the cursor is opened and a copy of the row is made in the static data set. If you use a scroll cursor within a transaction, you can prevent copied rows from changing, either by setting the isolation level to Repeatable Read or by locking the entire table in share mode during the transaction. Scrollable cursors cannot be declared `FOR UPDATE`.

The `DECLARE [SCROLL] CURSOR FROM` syntax allows you to declare a cursor directly with a string expression, so that you do not have to use the `PREPARE` instruction. This simplifies the source code and speeds up the execution time for non-Informix databases, because the SQL statement is not parsed twice.

Example 3: Declaring a scrollable cursor with string expression.

```
01 MAIN
02   DEFINE key INTEGER
03   DEFINE cust RECORDs
04             num INTEGER,
05             name CHAR(50)
06           END RECORD
07   DATABASE stores
08   DECLARE c1 SCROLL CURSOR
09      FROM "SELECT customer_num, cust_name FROM customer WHERE
customer_num>?"
10   LET key=101
11   FOREACH c1 USING key INTO cust.*
12      DISPLAY cust.*
13   END FOREACH
14 END MAIN
```

Hold cursors

Informix only: Use the `WITH HOLD` option to create a **hold cursor**. A hold cursor allows uninterrupted access to a set of rows across multiple transactions. Ordinarily, all cursors close at the end of a transaction. A hold cursor does not close; it remains open after a transaction ends. A hold cursor can be either a sequential cursor or a scrollable cursor. Hold cursors are only supported by Informix database engines.

You can use the `?` question mark place holders with prepared or static SQL statements, and provide the parameters at execution time with the `USING` clause of the OPEN or FOREACH instructions.

Example 4: Declaring a hold cursor with ? parameter place holders.

```
01 MAIN
02   DEFINE key INTEGER
03   DEFINE cust RECORDs
04             num INTEGER,
05             name CHAR(50)
06           END RECORD
07   DATABASE stores
08   DECLARE c1 CURSOR WITH HOLD
09      FOR SELECT customer_num, cust_name FROM customer WHERE
customer_num > ?
10   LET key=101
11   FOREACH c1 USING key INTO cust.*
12      BEGIN WORK
13      UPDATE cust2 SET name=cust.cust_name WHERE num=cust.num
14      COMMIT WORK
15   END FOREACH
16 END MAIN
```

OPEN

Purpose:

Executes the SQL statement associated with a database cursor declared in the same connection.

Syntax:

```
OPEN cid
   [ USING pvar {IN|OUT|INOUT} [,...] ]
   [ WITH REOPTIMIZATION ]
```

Notes:

1. *cid* is the identifier of the database cursor.
2. *pvar* is a program variable, a record, or an array used as a parameter buffer to provide SQL parameter values.

Usage:

The `OPEN` instruction executes the SQL statement of a declared cursor. The result set is produced on the server side and rows can be fetched.

The `USING` clause is required to provide the SQL parameters as program variables, if the cursor was declared with a prepared statement that includes (`?`) question mark placeholders.

A subsequent OPEN statement closes the cursor and then reopens it. When the database server reopens the cursor, it creates a new result set, based on the current values of the variables in the USING clause. If the variables have changed since the previous OPEN statement, reopening the cursor can generate an entirely different result set.

The IN, OUT or INOUT options can be used to call stored procedures having input / output parameters and generating a result set. Use the IN, OUT or INOUT options to indicate if a parameter is respectively for input, output or both. For more details about stored procedure calls, see SQL Programming.

Sometimes, query execution plans need to be re-optimized when SQL parameter values change. Use the WITH REOPTIMIZATION clause to indicate that the query execution plan has to be re-optimized on the database server (this operation is normally done during the DECLARE instruction). If this option is not supported by the database server, it is ignored.

In a IBM Informix database that is ANSI-compliant, you receive an error code if you try to open a cursor that is already open. **Informix only!**

With the CLOSE instruction, you can release resources allocated for the result set on the database server.

Warnings:

1. You cannot use string or numeric constants in the USING list. All elements must be program variables.
2. The database server evaluates the values that are named in the USING clause of the OPEN statement only when it opens the cursor. While the cursor is open, subsequent changes to program variables in the OPEN clause do not change the result set of the cursor; you must re-open the cursor to re-execute the statement.
3. If you release cursor resources with a FREE instruction, you cannot use the cursor unless you declare the cursor again.
4. The IN, OUT or INOUT options can only be used for simple variables, you cannot specify those options for a complete record with the record.* notation.

Example:

```
01 MAIN
02    DEFINE k INTEGER
03    DEFINE n VARCHAR(50)
04    DATABASE stores
05    DECLARE c1 CURSOR FROM "SELECT cust_name FROM customer WHERE
cust_id>?"
06    LET k = 102
07    OPEN c1 USING k
08    FETCH c1 INTO n
09    LET k = 103
10    OPEN c1 USING k
11    FETCH c1 INTO n
12 END MAIN
```

FETCH

Purpose:

Moves a cursor to a new row in the corresponding result set and retrieves the row values into fetch buffers.

Syntax:

```
FETCH [ direction ] cid [ INTO fvar [,...] ]
```

where *direction* is one of:

```
{
  NEXT
| { PREVIOUS | PRIOR }
| CURRENT
| FIRST
| LAST
| ABSOLUTE position
| RELATIVE offset
}
```

Notes:

1. *cid* is the identifier of the database cursor.
2. *fvar* is a program variable, a record or an array used as a fetch buffer to receive a row value.
3. *direction* options different from `NEXT` can only be used with scrollable cursors.
4. *position* is an positive integer expression.
5. *offset* is a positive or negative integer expression.

Usage:

The `FETCH` instruction retrieves a row from a result set of an opened cursor. The cursor must be opened before using the `FETCH` instruction.

The `INTO` clause can be used to provide the fetch buffers that receive the result set column values.

A sequential cursor can fetch only the next row in sequence from the result set.

The `NEXT` clause (the default) retrieves the next row in the result set. If the row pointer was on the last row before executing the instruction, the SQL Code is set to 100 (`NOTFOUND`), and the row pointer remains on the last row. (if you issue a `FETCH PREVIOUS` at this time, you get the next-to-last row).

The PREVIOUS clause retrieves the previous row in the result set. If the row pointer was on the first row before executing the instruction, the SQL Code is set to 100 (NOTFOUND), and the row pointer remains on the first row. (if you issue a FETCH NEXT at this time, you get the second row).

The CURRENT clause retrieves the current row in the result set.

The FIRST clause retrieves the first row in the result set.

The LAST clause retrieves the last row in the result set.

The ABSOLUTE clause retrieves the row at *position* in the result set. If the *position* is not correct, the SQL Code is set to 100 (NOTFOUND). Absolute row positions are numbered from 1.

The RELATIVE clause moves *offset* rows in the result set and returns the row at the current position. The offset can be a negative value. If the *offset* is not correct, the SQL Code is set to 100 (NOTFOUND). If *offset* is zero, the current row is fetched.

Warnings:

1. Fetching rows can have specific behavior when the cursor was declared FOR UPDATE. See Positioned Updates for more details.

Example:

```
01 MAIN
02    DEFINE cnum INTEGER
03    DEFINE cname CHAR(20)
04    DATABASE stores
05    DECLARE c1 SCROLL CURSOR FOR SELECT customer_num, cust_name FROM
customer
06    OPEN c1
07    FETCH c1 INTO cnum, cname
08    FETCH LAST c1 INTO cnum, cname
09    FETCH PREVIOUS c1 INTO cnum, cname
10    FETCH FIRST c1 INTO cnum, cname
11    FETCH LAST c1
12    FETCH FIRST c1
13 END MAIN
```

CLOSE

Purpose:

Closes a database cursor and frees resources allocated on the database server for the result set.

Syntax:

```
CLOSE cid
```

Notes:

1. *cid* is the identifier of the database cursor.

Usage:

The `CLOSE` instruction releases the resources allocated for the result set on the database server.

After using the `CLOSE` instruction, you must re-open the cursor with OPEN before retrieving values with FETCH.

Tips:

1. Close the cursor when the result set is no longer used, this saves resources on the database client and database server side.

Example:

```
01 MAIN
02    DATABASE stores
03    DECLARE c1 CURSOR FOR SELECT * FROM customer
04    OPEN c1
05    CLOSE c1
06    OPEN c1
07    CLOSE c1
08 END MAIN
```

FREE

Purpose:

This instruction releases resources allocated to the database cursor with the DECLARE instruction.

Syntax:

```
FREE cid
```

Notes:

1. *cid* is the identifier of the database cursor.

Usage:

The FREE instruction takes the name of a cursor as parameter.

All resources allocated to the database cursor are released.

The cursor should be explicitly closed before it is freed.

Warnings:

1. If you release cursor resources with this instruction, you cannot use the cursor unless you declare the cursor again.

Tips:

1. Free the cursor when the result set is no longer used; this saves resources on the database client and database server side.

Example:

```
01 MAIN
02    DEFINE i, j INTEGER
03    DATABASE stores
04    FOR i=1 TO 10
05        DECLARE c1 CURSOR FOR SELECT * FROM customer
06        FOR j=1 TO 10
07            OPEN c1
08            FETCH c1
09            CLOSE c1
10        END FOR
11        FREE c1
12    END FOR
13 END MAIN
```

FOREACH

Purpose:

A FOREACH block applies a series of actions to each row of data that is returned from a database cursor.

Syntax:

```
FOREACH cid
   [ USING pvar {IN|OUT|INOUT} [,...] ]
   [ INTO fvar [,...] ]
   [ WITH REOPTIMIZATION ]
       {
         statement
```

```
        | CONTINUE FOREACH
        | EXIT FOREACH
        }
        [...]
END FOREACH
```

Notes:

1. *cid* is the identifier of the database cursor.
2. *pvar* is a program variable, a record or an array used as a parameter buffer to provide SQL parameter values.
3. *fvar* is a program variable, a record or an array used as a fetch buffer to receive a row value.

Usage:

Use the `FOREACH` instruction to retrieve and process database rows that were selected by a query. This instruction is equivalent to using the OPEN, FETCH and CLOSE cursor instructions:

1. Open the specified cursor
2. Fetch the rows selected
3. Close the cursor (after the last row has been fetched)

You must declare the cursor (by using the DECLARE instruction) before the `FOREACH` instruction can retrieve the rows. A compile-time error occurs unless the cursor was declared prior to this point in the source module. You can reference a sequential cursor, a scroll cursor, a hold cursor, or an update cursor, but `FOREACH` only processes rows in sequential order.

The `FOREACH` statement performs successive fetches until all rows specified by the `SELECT` statement are retrieved. Then the cursor is automatically closed. It is also closed if a `WHENEVER NOT FOUND` exception handler within the `FOREACH` loop detects a `NOTFOUND` condition (that is, SQL Code = 100).

The `USING` clause is required to provide the SQL parameter buffers, if the cursor was declared with a prepared statement that includes (`?`) question mark placeholders.

The `IN`, `OUT` or `INOUT` options can be used to call stored procedures having input / output parameters and generating a result set. Use the `IN`, `OUT`, or `INOUT` options to indicate if a parameter is respectively for input, output, or both. For more details about stored procedure calls, see SQL Programming.

The `INTO` clause can be used to provide the fetch buffers that receive the row values.

Use the `WITH REOPTIMIZATION` clause to indicate that the query execution plan has to be re-optimized.

The `CONTINUE FOREACH` instruction interrupts processing of the current row and starts processing the next row. The runtime system fetches the next row and resumes processing at the first statement in the block.

The `EXIT FOREACH` instruction interrupts processing and ignores the remaining rows of the result set.

Warnings:

1. Infinite loops may occur if the cursor preparation failed.
2. The `IN`, `OUT`, or `INOUT` options can only be used for simple variables; you cannot specify those options for a complete record with the record.* notation.

Example:

```
01 MAIN
02    DEFINE clist ARRAY[200] OF RECORD
03             cnum INTEGER,
04             cname CHAR(50)
05           END RECORD
06    DEFINE i INTEGER
07    DATABASE stores
08    DECLARE c1 CURSOR FOR SELECT customer_num, cust_name FROM
customer
09    LET i=0
10    FOREACH c1 INTO clist[i+1].*
11       LET i=i+1
12       DISPLAY clist[i].*
13    END FOREACH
14    DISPLAY "Number of rows found: ", i
15 END MAIN
```

SQL Positioned Updates

Summary:

- What is a Positioned Update?
- Declaring a cursor for update (`DECLARE`)
- Updating a row by cursor position (`UPDATE ... WHERE CURRENT OF`)
- Deleting a row by cursor position (`DELETE ... WHERE CURRENT OF`)
- Examples

See also: Transactions, Static SQL, Dynamic SQL, Result Sets, SQL Errors.

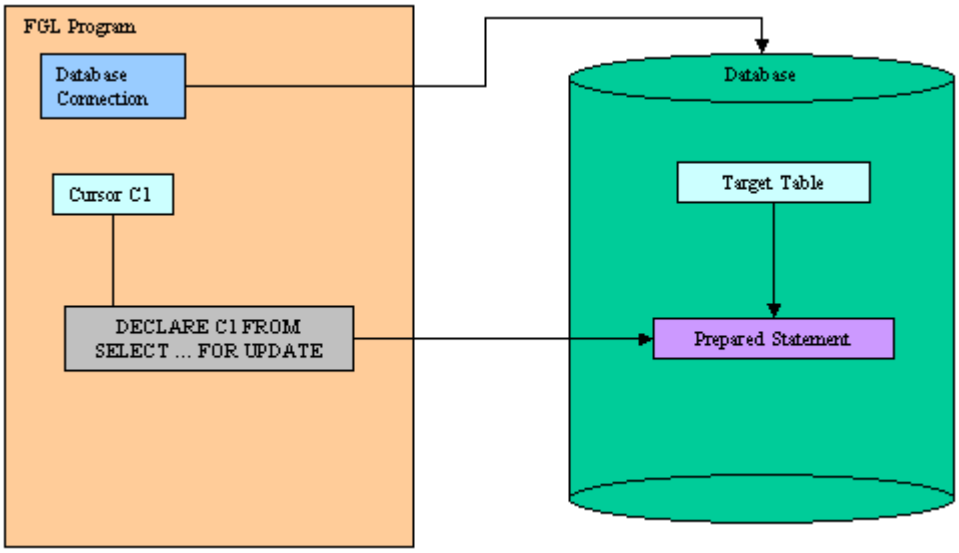
What is a Positioned Update?

When declaring a database cursor with a `SELECT` statement using a unique table and including the `FOR UPDATE` keywords, you can update or delete database rows by using the `WHERE CURRENT OF` keywords in the UPDATE or DELETE statements. Such an operation is called Positioned Update or Positioned Delete.

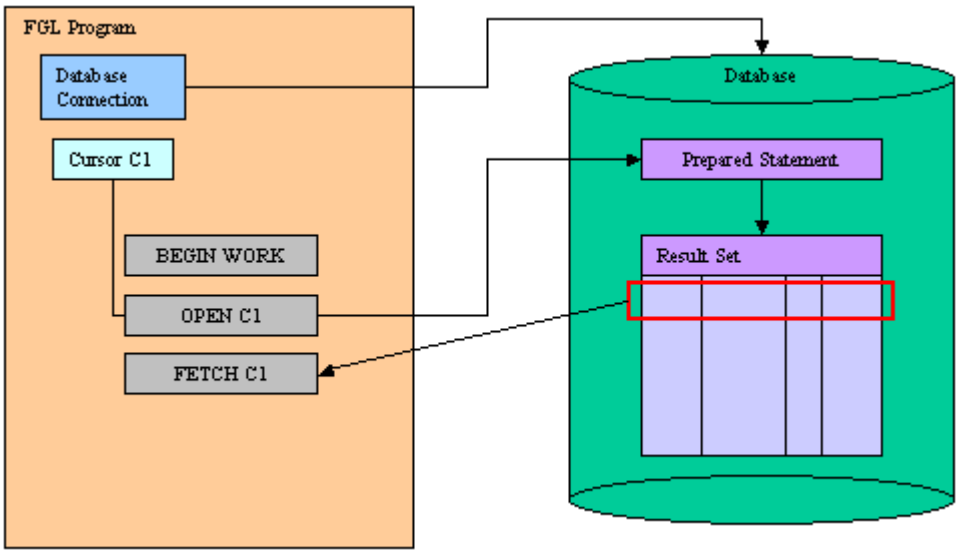
Some database servers do not support **hold** cursors (`WITH HOLD`) declared with a `SELECT` statement including the `FOR UPDATE` keywords. The SQL standards require 'for update' cursors to be automatically closed at the end of a transaction. Therefore, it is strongly recommended that you use positioned updates in a transaction block.

Do not confuse positioned update with the use of `SELECT FOR UPDATE` statements that are not associated with a database cursor. Executing `SELECT FOR UPDATE` statements is supported by the language, but you cannot perform positioned updates since there is no cursor identifier associated to the result set.

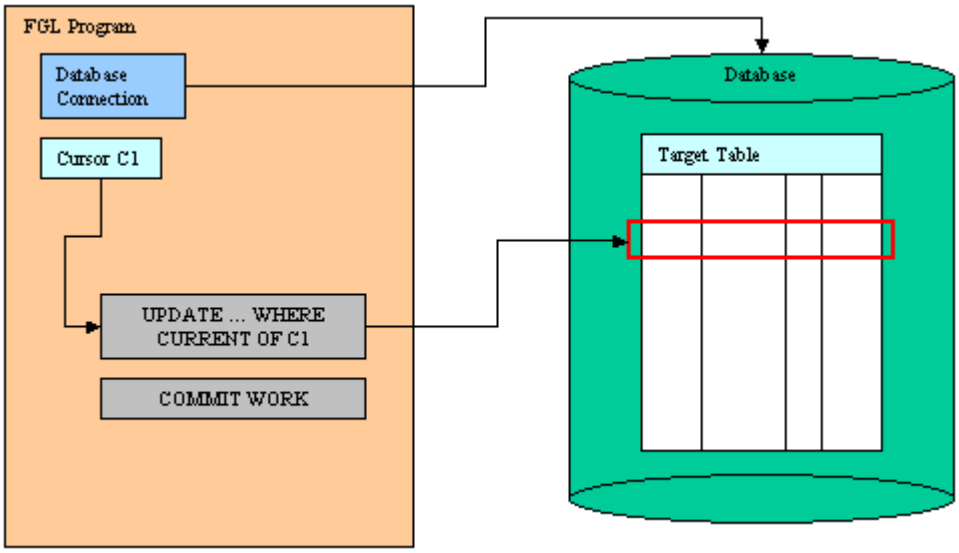
To perform a positioned update or delete, you must declare the database cursor with a `SELECT FOR UPDATE` statement:



Then, start a transaction, open the cursor and fetch a row:



Finally, you update or delete the current row and you commit the transaction:



DECLARE

Purpose:

Use this instruction to associate a database cursor with a `SELECT` statement to perform positioned updates in the current connection.

Syntax:

```
DECLARE cid [SCROLL] CURSOR [WITH HOLD] FOR { select-statement | sid }
```

Notes:

1. *cid* is the identifier of the database cursor.
2. *select-statement* is a `SELECT` statement defined in Static SQL.
3. To perform positioned updates, the *select-statement* must include the `FOR UPDATE` keywords.
4. *sid* is the identifier of a prepared `SELECT` statement including the `FOR UPDATE` keywords.
5. See the `DECLARE` instruction description in Result Sets Processing.
6. `DECLARE` must precede any other statement that refers to the cursor during program execution.

Warnings:

1. The scope of reference of the *cid* cursor identifier is local to the module where it is declared. Therefore, you must execute the `DECLARE`, `UPDATE` or `DELETE` instructions in the same module.

2. Use the `WITH HOLD` option carefully, because this feature is specific to IBM Informix servers. Other database servers do not behave as Informix does with such cursors. For example, if the `SELECT` is not declared `FOR UPDATE`, most database servers keep cursors open after the end of a transaction, but IBM DB2 automatically closes all cursors when the transaction is rolled back.

UPDATE ... WHERE CURRENT OF

Purpose:

Updates the current row in a result set of a database cursor declared for update.

Syntax:

```
UPDATE table-specification
   SET
       column = { variable | literal | NULL }
       [,...]
   WHERE CURRENT OF cid
```

Notes:

1. *table-specification* identifies the target table (see UPDATE for more details).
2. *column* is a name of a table column.
3. *variable* is a program variable, a record or an array used as a parameter buffer to provide values.
4. *literal* is any literal expression supported by the language.
5. *cid* is the identifier of the database cursor declared for update.
6. The `UPDATE` statement does not advance the cursor to the next row, so the current row position remains unchanged.

Warnings:

1. The scope of reference of the *cid* cursor identifier is local to the module where it is declared. Therefore, you must execute the `DECLARE`, `UPDATE` or `DELETE` instructions in the same module.
2. There must be a current row in the result set. Make sure that the SQL status returned by the last FETCH is equal to zero.
3. If the `DECLARE` statement that created the cursor specified one or more columns in the `FOR UPDATE` clause, you are restricted to updating only those columns in a subsequent `UPDATE ... WHERE CURRENT OF` statement.

DELETE ... WHERE CURRENT OF

Purpose:

Deletes the current row in a result set of a database cursor declared for update.

Syntax:

```
DELETE FROM table-specification
   WHERE CURRENT OF cid
```

Notes:

1. *table-specification* identifies the target table (see DELETE for more details).
2. *cid* is the identifier of the database cursor declared for update.
3. After the deletion, no current row exists; you cannot use the cursor to delete or update a row until you re-position the cursor with a FETCH statement.

Warnings:

1. The scope of reference of the *cid* cursor identifier is local to the module where it is declared. Therefore, you must execute the `DECLARE`, `UPDATE` or `DELETE` instructions in the same module.
2. There must be a current row in the result set. Make sure that the SQL status returned by the last FETCH is equal to zero.

Examples

Example 1:

```
01 MAIN
02    DEFINE pname CHAR(30)
03    DATABASE stock
04    DECLARE uc CURSOR FOR
05      SELECT name FROM item WHERE key=123 FOR UPDATE
06    BEGIN WORK
07      OPEN uc
08      FETCH uc INTO pname
09      IF sqlca.sqlcode=0 THEN
10         LET pname = "Dummy"
11         UPDATE item SET name=pname WHERE CURRENT OF uc
12      END IF
13      CLOSE uc
14    COMMIT WORK
15    FREE uc
16 END MAIN
```

SQL Insert Cursors

Summary:

- What is an Insert Cursor?
- Declaring the Insert Cursor (`DECLARE`)
- Initializing the Insert Cursor (`OPEN`)
- Adding Rows to the Buffer (`PUT`)
- Flushing the insert Buffer (`FLUSH`)
- Finalizing the Insert Cursor (`CLOSE`)
- Freeing Allocated Resources (`FREE`)
- Examples

See also: Transactions, Static SQL, Dynamic SQL, Result Sets, SQL Errors.

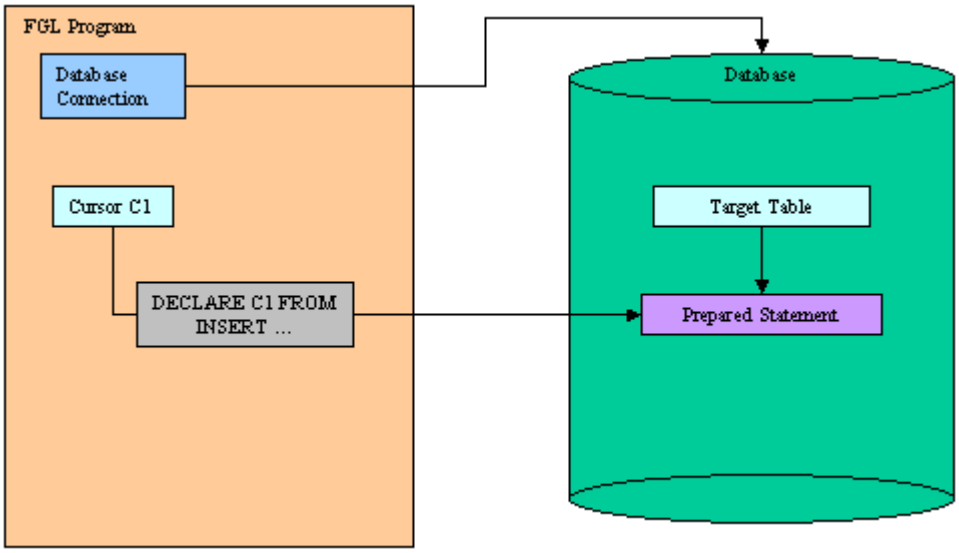
What is an Insert Cursor?

An Insert Cursor is a database cursor declared with a restricted form of the INSERT statement, designed to perform buffered row insertion in database tables.

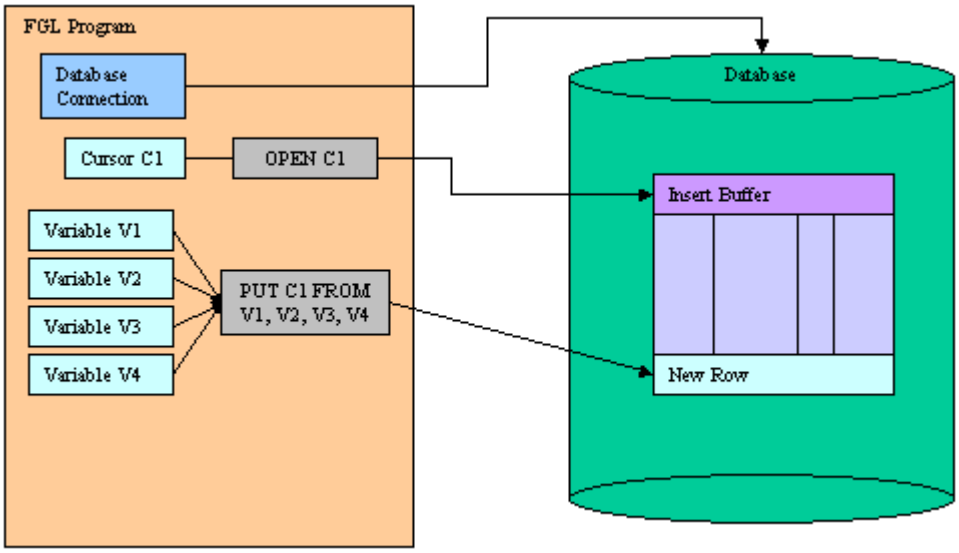
The insert cursor simply inserts rows of data; it cannot be used to fetch data. When an insert cursor is opened, a buffer is created in memory to hold a block of rows. The buffer receives rows of data as the program executes PUT statements. The rows are written to disk only when the buffer is full. You can use the CLOSE, FLUSH, or COMMIT WORK statement to flush the buffer when it is less than full. You must close an insert cursor to insert any buffered rows into the database before the program ends. You can lose data if you do not close the cursor properly.

When the database server supports buffered inserts, an insert cursor increases processing efficiency (compared with embedding the `INSERT` statement directly). This process reduces communication between the program and the database server and also increases the speed of the insertions.

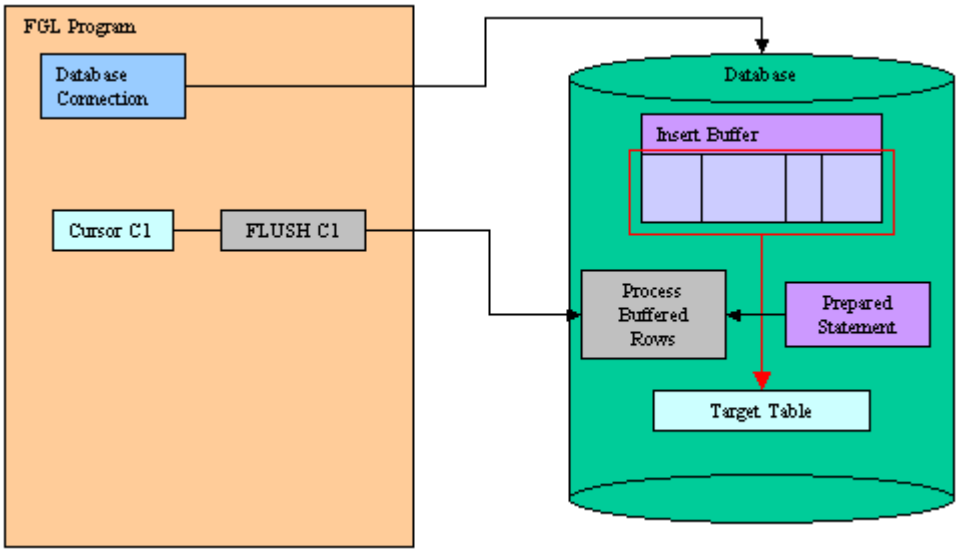
Before using the insert cursor, you must declare it with the DECLARE instruction using an `INSERT` statement:



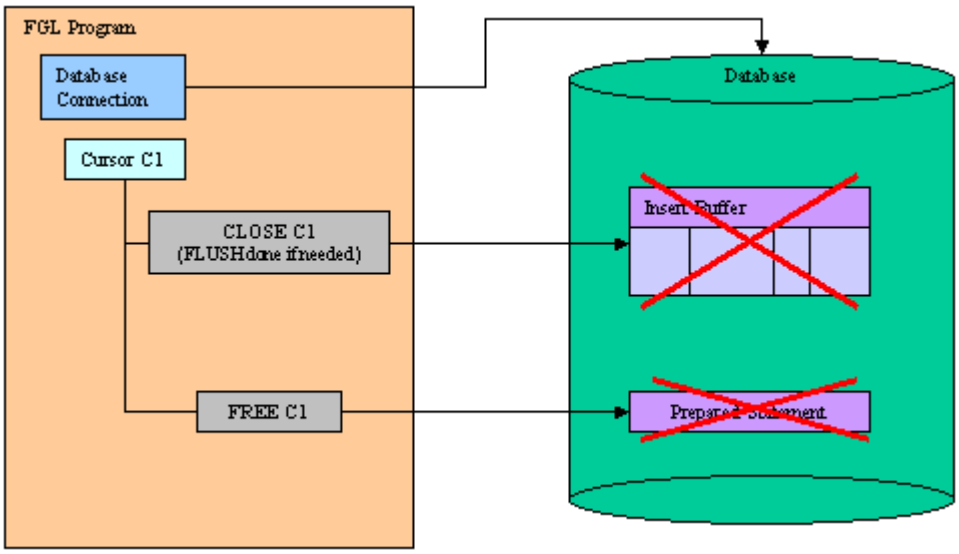
Once declared, you can open the insert cursor with the OPEN instruction. This instruction prepares the insert buffer. When the insert cursor is opened, you can add rows to the insert buffer with the PUT statement:



Rows are automatically added to the database table when the insert buffer is full. To force row insertion in the table, you can use the FLUSH instruction:



Finally, when all rows are added, you can CLOSE the cursor and if you no longer need it, you can de-allocate resources with the FREE instruction:



By default, insert cursors must be opened inside a transaction block, with `BEGIN WORK` and `COMMIT WORK`, and they are automatically closed at the end of the transaction. If needed, you can declare insert cursors with the `WITH HOLD` clause, to allow uninterrupted row insertion across multiple transactions. See example 3 at the bottom of this page.

DECLARE

Purpose:

Declares a new insert cursor in the current database session.

Syntax:

```
DECLARE cid CURSOR [WITH HOLD] FOR { insert-statement | sid }
```

Notes:

1. *cid* is the identifier of the insert cursor.
2. *insert-statement* is an `INSERT` statement defined in Static SQL.
3. *sid* is the identifier of a prepared `INSERT` statement including (`?`) question mark placeholders in the `VALUES` clause.
4. The `INSERT` statement is parsed, validated and the execution plan is created.
5. `DECLARE` must precede any other statement that refers to the cursor during program execution.
6. The scope of reference of the *cid* cursor identifier is local to the module where it is declared.
7. When declaring a cursor with a static *insert-statement*, the statement can include a list of variables in the `VALUES` clause. These variables are automatically read by the PUT statement; you do not have to provide the list of variables in that statement. See Example 1 for more details.
8. When declaring a cursor with a prepared *sid* statement, the statement can include (`?`) question mark placeholders for SQL parameters. In this case you must provide a list of variables in the `FROM` clause of the PUT statement. See Example 2 for more details.
9. Use the `WITH HOLD` option to declare cursors that have uninterrupted inserts across multiple transactions.
10. Resources allocated by the `DECLARE` can be released later by the FREE instruction.

Warnings:

1. The number of declared cursors in a single program is limited by the database server and the available memory. Make sure that you free the resources when you no longer need the declared insert cursor.
2. The identifier of a cursor that was declared in one module cannot be referenced from another module.

OPEN

Purpose:

Opens an insert cursor in the current database session.

Syntax:

```
OPEN cid
```

1. *cid* is the identifier of the insert cursor.
2. A subsequent `OPEN` statement closes the cursor and then reopens it.
3. With the CLOSE instruction, you can release resources allocated for the insert buffer on the database server.

Warnings:

1. When used with an insert cursor, the `OPEN` instruction cannot include a `USING` clause.
2. If the insert cursor was not declared `WITH HOLD` option, the `OPEN` instruction generates an SQL error if there is no current transaction started.
3. If you release cursor resources with a FREE instruction, you cannot use the cursor unless you declare the cursor again.

PUT

Purpose:

Adds a new row to the insert cursor buffer in the current database session.

Syntax:

```
PUT cid FROM paramvar [,...]
```

Notes:

1. *cid* is the identifier of the insert cursor.
2. *paramvar* is a program variable, a record or an array used as a parameter buffer to provide SQL parameter values.

Warnings:

1. If the insert cursor was not declared `WITH HOLD` option, the `PUT` instruction generates an SQL error if there is no current transaction started.
2. If the insert buffer has no room for the new row when the statement executes, the buffered rows are written to the database in a block, and the buffer is emptied. As

a result, some `PUT` statement executions cause rows to be written to the database, and some do not.

FLUSH

Purpose:

Flushes the buffer of an insert cursor in the current database session.

Syntax:

```
FLUSH cid
```

Notes:

1. *cid* is the identifier of the insert cursor.
2. All buffered rows are inserted into the target table.
3. The insert buffer is cleared.

Warnings:

1. The insert buffer may be automatically flushed by the runtime system if there no room when a new row is added with the `PUT` instruction.

CLOSE

Purpose:

Closes an insert cursor in the current database session.

Syntax:

```
CLOSE cid
```

Notes:

1. *cid* is the identifier of the insert cursor.
2. If rows are present in the insert buffer, they are inserted into the target table.
3. The insert buffer is discarded.
4. This instruction releases the resources allocated for the insert buffer on the database server.
5. After using the `CLOSE` instruction, you must re-open the cursor with OPEN before adding new rows with PUT / FLUSH.

FREE

Purpose:

Releases resources allocated for an insert cursor in the current database session.

Syntax:

```
FREE cid
```

Notes:

1. *cid* is the identifier of the insert cursor.
2. All resources allocated to the insert cursor are released.
3. The cursor should be explicitly closed before it is freed.

Warnings:

1. If you release cursor resources with this instruction, you cannot use the cursor unless you declare the cursor again.

Examples

Example 1: Insert Cursor declared with a Static INSERT

```
01 MAIN
02    DEFINE i INTEGER
03    DEFINE rec RECORD
04             key INTEGER,
05             name CHAR(30)
06           END RECORD
07    DATABASE stock
08    DECLARE ic CURSOR FOR
09      INSERT INTO item VALUES (rec.*)
10    BEGIN WORK
11      OPEN ic
12      FOR i=1 TO 100
13          LET rec.key = i
14          LET rec.name = "Item #" || i
15          PUT ic
16          IF i MOD 50 = 0 THEN
17              FLUSH ic
18          END IF
19      END FOR
20      CLOSE ic
21    COMMIT WORK
22    FREE ic
```

```
23 END MAIN
```

Example 2: Insert Cursor declared with a Prepared INSERT

```
01 MAIN
02    DEFINE i INTEGER
03    DEFINE rec RECORD
04            key INTEGER,
05            name CHAR(30)
06          END RECORD
07    DATABASE stock
08    PREPARE is FROM "INSERT INTO item VALUES (?,?)"
09    DECLARE ic CURSOR FOR is
10    BEGIN WORK
11      OPEN ic
12      FOR i=1 TO 100
13          LET rec.key = i
14          LET rec.name = "Item #" || i
15          PUT ic FROM rec.*
16          IF i MOD 50 = 0 THEN
17              FLUSH ic
18          END IF
19      END FOR
20      CLOSE ic
21    COMMIT WORK
22    FREE ic
23    FREE is
24 END MAIN
```

Example 3: Insert Cursor declared with 'hold' option

```
01 MAIN
02    DEFINE name CHAR(30)
03    DATABASE stock
04    DECLARE ic CURSOR WITH HOLD FOR
05      INSERT INTO item VALUES (1,name)
06    OPEN ic
07    LET name = "Item 1"
08    PUT ic
09    BEGIN WORK
10      UPDATE refs SET name="xyz" WHERE key=123
11    COMMIT WORK
12    PUT ic
13    PUT ic
14    FLUSH ic
15    CLOSE ic
16    FREE ic
17 END MAIN
```

I/O SQL Instructions

Summary:

- Loading data from files (LOAD)
- Writing data to files (UNLOAD)

See also: Connections.

LOAD

Purpose:

The LOAD instruction inserts data from a file into an existing table in the current database connection.

Syntax:

```
LOAD FROM filename [ DELIMITER delimiter ]
{
  INSERT INTO table-specification [ ( column [,...] ) ]
|
  insert-string
}
```

where *table-specification* is:

```
[dbname[@dbserver]:][owner.]table
```

Notes:

1. *filename* is the name of the file the data is read from.
2. *delimiter* is the character used as the value delimiter. If this clause is not specified, the delimiter is defined by DBDELIMITER environment variable. If this variable is not set, the default is a pipe.
3. The INSERT clause is a pseudo INSERT statement (without the VALUES clause), where you can specify the list of columns in braces.
4. *dbname* identifies the database name. **Informix only!**
5. *dbserver* identifies the Informix database server (INFORMIXSERVER). **Informix only!**
6. *owner* identifies the owner of the table, with optional double quotes. **Informix only!**
7. *table* is the name of the database table.
8. *column* is a name of a table column.
9. *insert-string* is a program variable or a string literal containing the pseudo-INSERT statement. This allows you to create the pseudo-INSERT statement at runtime.

Warnings:

1. The number and the order of columns in the INSERT statement must match in the input file.
2. You cannot use the PREPARE statement to preprocess a `LOAD` statement.
3. At this time, data type description of the input file fields is implicit; in order to create the SQL parameter buffers to hold the field values for inserts, the `LOAD` instruction uses the current database connection to get the column data types of the target table. Those data types depend on the type of database server. For example, IBM Informix DATE columns do not store the same data as the Oracle DATE data type. Therefore, be careful when using this instruction; if your application connects to different kinds of database servers, you may get data conversion errors.

Tips:

1. `LOAD` provides better performance when the *table* that the `INSERT INTO` clause references has no index, no constraint, and no trigger. If one or more triggers, constraints, or indexes exist on the table, however, it is recommended that you disable these objects if the database server allows such SQL operations. For example, with IBM Informix, you can issue one of the following SQL statements:
   - `SET INDEX ... DISABLED`
   - `SET CONSTRAINT ... DISABLED`
   - `SET TRIGGER ... DISABLED`

Usage:

The `LOAD` statement must include a pseudo-INSERT statement (either directly or as text in a variable) to specify where to store the data. `LOAD` appends the new rows to the specified table, synonym, or view, but does not overwrite existing data. It cannot add a row that has the same key as an existing row.

Warning: The DELIMITER cannot be backslash or any hexadecimal digit (0-9, A-F, a-f).

The variable or string following the `LOAD FROM` keywords must specify the name of a file of ASCII characters (or characters that are valid for the client locale) that represent the data values that are to be inserted. How data values in this input file should be represented by a character string depends on the SQL data type of the receiving column in *table.*

Data Type	Input Format
`CHAR, VARCHAR, TEXT`	Values can have more characters than the declared maximum length of the column, but any extra characters are ignored. A backslash (\ ) is required before any literal backslash or any literal delimiter character, and before any NEWLINE character anywhere in character value. Blank values can be represented as one or more blank characters between delimiters, but leading blanks must not precede

	other CHAR, VARCHAR, or TEXT values.
DATE	In the default locale, values must be in *month/day/year* format unless another format is specified by DBDATE environment variable. You must represent the month as a 2-digit number. You can use a 2-digit number for the year if you are satisfied with the DBCENTURY setting. Values must be actual dates; for example, February 30 is invalid.
DATETIME	DATETIME values must be in the format: *year-month-day hour:minute:second.fraction* or a contiguous subset, without the DATETIME keyword or qualifiers. Time units outside the declared column precision can be omitted. The *year* must be a four-digit number; all other time units (except *fraction*) require two digits.
INTERVAL	INTERVAL values must be formatted: *year-month* or else *day hour:minute:second.fraction* or a contiguous subset thereof, without the INTERVAL keyword or qualifiers. Time units outside the declared column precision can be omitted. All time units (except *year* and *fraction*) require two digits.
MONEY	Values can include currency symbols, but these are not required.
BYTE	Values must be ASCII-hexadecimals; no leading or trailing blanks.
SERIAL	Values can be represented as 0 to tell the database server to supply a new SERIAL value. You can specify a literal integer greater than zero, but if the column has a unique index, an error results if this number duplicates an existing value.

Each set of data values in *filename* that represents a new row is called an input record. The NEWLINE character must terminate each input record in *filename*. Specify only values that the language can convert to the data type of the database column. For database columns of character data types, inserted values are truncated from the right if they exceed the declared length of the column.

NULL values of any data type must be represented by consecutive delimiters in the input file; you cannot include anything between the delimiter symbols.

Each input record must contain the same number of delimited data values. If the INSERT clause has no list of columns, the sequence of values in each input record must match the columns of *table* in number and order. Each value must have the literal format of the column data type, or of a compatible data type.

A file created by the UNLOAD statement can be used as input for the LOAD statement if its values are compatible with the schema of *table*.

The statement expects incoming data in the format specified by environment variables like DBFORMAT, DBMONEY, DBDATE, GL_DATE, and GL_DATETIME. The precedence of these format specifications is consistent with forms and reports. If there is an inconsistency, an error is reported and the LOAD is cancelled.

If LOAD is executed within a transaction, the inserted rows are locked, and they remain locked until the COMMIT WORK or ROLLBACK WORK statement terminates the transaction. If no other user is accessing the table, you can avoid locking limits and reduce locking overhead by locking the table with the LOCK TABLE statement after the transaction begins. This exclusive table lock is released when the transaction terminates. Consult the documentation for your database server about the limit on the number of locks available during a single transaction.

If the current database has no transaction log, a failing LOAD statement cannot remove any rows that were loaded before the failure occurred. You must manually remove the already loaded records from either the load file or from the receiving table, repair the erroneous records, and rerun LOAD.

Regarding transaction, you can take one of the following actions when the database has a transaction log:

- Run LOAD as a singleton transaction, so that any error causes the entire LOAD statement to be automatically rolled back.
- Run LOAD within an explicit transaction with BEGIN WORK / COMMIT WORK, so that a data error merely stops the LOAD statement in place with the transaction still open.

Example:

```
01 MAIN
02    DATABASE stores
03    BEGIN WORK
04    DELETE FROM items
05    LOAD FROM "items01.unl" INSERT INTO items
06    LOAD FROM "items02.unl" INSERT INTO items
07    COMMIT WORK
08 END MAIN
```

UNLOAD

Purpose:

The UNLOAD instruction copies data from a current database to a file.

Syntax:

```
UNLOAD TO filename [ DELIMITER delimiter ]
{
```

```
  select-statement
|
  select-string
}
```

Notes:

1. *filename* is the name of the file the data is written to.
2. *delimiter* is the character used as the value delimiter. If this clause is not specified, the delimiter is defined by the DBDELIMITER environment variable. If this variable is not set, the default is a pipe.
3. *select-statement* is any kind of Static SELECT statement supported by the language.
4. *select-string* is a program variable or a string literal containing the `SELECT` statement to produce the rows. This allows you to create the `SELECT` statement at runtime.

Warnings:

1. You cannot use the PREPARE statement to preprocess an `UNLOAD` statement.
2. When using a *select-string*, do not attempt to substitute question marks (`?`) in place of host variables to make the `SELECT` statement dynamic, because this usage has binding problems.
3. At this time, data type description of the output file fields is implicit; in order to create the fetch buffers to hold the column values, the `UNLOAD` instruction uses the current database connection to get the column data types of the generated result set. Those data types depend on the type of database server. For example, IBM Informix INTEGER columns are 4-bytes integer values, while Oracle INTEGER data type is actually a NUMBER value. Therefore, you should take care when using this instruction; if your application connects to different kinds of database servers, you may get data conversion errors.

Usage:

The `UNLOAD` statement must include a `SELECT` statement (directly, or in a variable) to specify what rows to copy into *filename*. `UNLOAD` does not delete the copied data.

Warning: The DELIMITER cannot be backslash or any hexadecimal digit (0-9, A-F, a-f).

The *filename* identifies an output file in which to store the rows retrieved from the database by the `SELECT` statement. In the default (U.S. English) locale, this file contains only ASCII characters. (In other locales, output from `UNLOAD` can contain characters from the code-set of the locale.)

A set of values in output representing a row from the database is called an *output record*. A NEWLINE character (ASCII 10) terminates each output record.

The UNLOAD statement represents each value in the output file as a string of ASCII characters, according to the declared data type of the database column:

Data Type	Output Format
CHAR, VARCHAR, TEXT	Trailing blanks are dropped from CHAR and TEXT (but not from VARCHAR) values. A backslash (\ ) is inserted before any literal backslash or delimiter character and before a NEWLINE character in a character value.
DECIMAL, FLOAT, INTEGER, MONEY, SMALLFLOAT, SMALLINT	Values are written as literals with no leading blanks. MONEY values are represented with no leading currency symbol. Zero values are represented as 0 for INTEGER or SMALLINT columns, and as 0.00 for FLOAT, SMALLFLOAT, DECIMAL, and MONEY columns.
DATE	Values are written in the format *month/day/year* unless some other format is specified by the DBDATE environment variable.
DATETIME	DATETIME values are formatted *year-month-day hour:minute:second.fraction* or a contiguous subset, without DATETIME keyword or qualifiers. Time units outside the declared precision of the database column are omitted.
INTERVAL	INTERVAL values are formatted *year-month* or else as *day hour:minute:second.fraction* or a contiguous subset, without INTERVAL keyword or qualifiers. Time units outside the declared precision of the database column are omitted.
BYTE	BYTE Values are written in ASCII hexadecimal form, without any added blank or NEWLINE characters. The logical record length of an output file that contains BYTE values can be very long, and thus might be very difficult to print or to edit.

NULL values of any data type are represented by consecutive delimiters in the output file, without any characters between the delimiter symbols.

The backslash symbol (\) serves as an escape character in the output file to indicate that the next character in a data value is a literal. The UNLOAD statement automatically inserts a preceding backslash to prevent literal characters from being interpreted as special characters in the following contexts:

- The backslash character appears anywhere in the value.
- The delimiter character appears anywhere in the value.
- The NEWLINE character appears anywhere in a value.

Example:

```
01 MAIN
02    DEFINE var INTEGER
03    DATABASE stores
04    LET var = 123
05    UNLOAD TO "items.unl" SELECT * FROM items WHERE item_num > var
06 END MAIN
```

SQL Programming

Summary:

- 1. Programming
  - 1.1 Database utility library
  - 1.2 Implicit database connection
  - 1.3 Managing transaction commands
  - 1.4 Executing stored procedures
  - 1.5 Cursors and Connections
  - 1.6 SQL Error identification
- 2. Performance
  - 2.1 Using dynamic SQL
  - 2.2 Using transactions
  - 2.3 Avoiding long transactions
  - 2.4 Declaring prepared statements
  - 2.5 Saving SQL resources
- 3. Portability
  - 3.1 Database entities
  - 3.2 Database users and security
  - 3.3 Database creation statements
  - 3.4 Data definition statements
  - 3.5 Using portable data types
  - 3.6 CHAR and VARCHAR types
  - 3.7 Concurrent data access
  - 3.8 The SQLCA register
  - 3.9 Optimistic locking
  - 3.10 Auto-incremented columns (SERIALs)
  - 3.11 Informix SQL ANSI mode
  - 3.12 Positioned Updates/Deletes
  - 3.13 WITH HOLD and FOR UPDATE
  - 3.14 String literals in SQL statements
  - 3.15 Date and time literals in SQL statements
  - 3.16 Naming database objects
  - 3.17 Temporary tables
  - 3.18 Outer joins
  - 3.19 Sub-string expressions
  - 3.20 Using ROWIDs
  - 3.21 MATCHES operator
  - 3.22 GROUP BY clause
  - 3.23 LENGTH() function
  - 3.24 SQL Interruption

See also: Connections, Transactions, Static SQL, Dynamic SQL, Result Sets, SQL Errors, Programs.

1. Programming

1,1 Database utility library

The BDL library "**fgldbutl.4gl**" provides several utility functions. For example, this library implements a function to get the type of the database engine at runtime. You will find this library in the **FGLDIR/src** directory. See the source file for more details.

1.2 Implicit database connection

In BDL, the DATABASE statement can be used in two distinct ways, depending on the context of the statement within its source module :

- To specify a default database : Typically used in a "GLOBALS" module, to define variables with the LIKE clause, but it is also used for the INITIALIZE and VALIDATE statements. Using the DATABASE statement in this way results in that database being opened automatically at run time.
- To specify a current database : In MAIN or in a FUNCTION, used to connect to a database. A variable can be used in this context (DATABASE *varname*).

A default database is almost always used, because many BDL applications contain DEFINE ... LIKE statements. A problem occurs when the production database name differs from the development database name, because the default database specification will result in an automatic connection (just after MAIN):

```
01 DATABASE stock_dev
02 DEFINE
03    p_cust RECORD LIKE customer.*
04 MAIN
05    DEFINE dbname CHAR(30)
06    LET dbname = "stock1"
07    DATABASE dbname
08 END MAIN
```

In order to avoid the implicit connection, you can use the SCHEMA instruction instead of DATABASE:

```
01 SCHEMA stock_dev
02 DEFINE
03    p_cust RECORD LIKE customer.*
04 MAIN
05    DEFINE dbname CHAR(30)
06    LET dbname = "stock1"
07    DATABASE dbname
08 END MAIN
```

This instruction will define the database schema for compilation only, and will not make an implicit connection at runtime.

1.3 Managing transaction commands

A BDL program can become very complex if a lot of nested functions do SQL processing. When using a database supporting transactions, you must sometimes execute all SQL statements in the same transaction block. This can be done easily by centralizing transaction control commands in wrapper functions.

The **fgldbutl.4gl** library contains special functions to manage the beginning and the end of a transaction with an internal counter, in order to implement nested function calls inside a unique transaction.

Example:

```
01 MAIN
02     IF a() <> 0 THEN
03        ERROR "..."
04     END IF
05     IF b() <> 0 THEN
06        ERROR "..."
07     END IF
08 END MAIN
09
10 FUNCTION a()
11   DEFINE s INTEGER
12   LET s = db_start_transaction( )
13   UPDATE ...
14   LET s = SQLCA.SQLCODE
15   IF s = 0 THEN
16      LET s = b( )
17   END IF
18   LET s = db_finish_transaction((s==0))
19   RETURN s
20 END FUNCTION
21
22 FUNCTION b()
23    DEFINE s INTEGER
24    LET s = db_start_transaction( )
25    UPDATE ...
26    LET s = SQLCA.SQLCODE
27    LET s = db_finish_transaction((s==0))
28    RETURN s
29 END FUNCTION
```

In this example, you see in the MAIN block that both functions a() and b() can be called separately. However, the transaction SQL commands will be used only if needed: When function a() is called, it starts the transaction, then calls b(), which does not start the transaction since it was already started by a(). When function b() is called directly, it starts the transaction.

The function `db_finish_transaction()` is called with the expression `(s==0)`, which is evaluated before the call. This allows you to write in one line the equivalent of the following `IF` statement:

```
01 IF s==0 THEN
02    LET s = db_finish_transaction(1)
03 ELSE
04    LET s = db_finish_transaction(0)
05 END IF
```

1.4 Executing stored procedures

Specifying output parameters

Beginning with Genero version 2.00, it is now possible to specify OUTPUT parameters to get values from stored procedures. While this new feature is generic, stored procedures execution needs to be addressed specifically according to the database type. There are different ways to execute a stored procedure. This section describes how to execute stored procedures on the supported database engines.

Tip: In order to write reusable code, you should encapsulate each stored procedure execution in a program function performing database-specific SQL based on a global database type variable. The program function would just take the input parameters and return the output parameters of the stored procedure, hiding database-specific execution steps from the caller.

Stored procedures returning a result set

With some database servers it is possible to execute stored procedures that produce a result set, and fetch the rows as normal `SELECT` statements, by using `DECLARE`, `OPEN`, `FETCH`. Some databases can return multiple result sets and cursor handles declared in a stored procedure as output parameters, but Genero supports only unique and anonymous result sets. See below for examples.

Calling stored procedures with supported databases

- Stored procedure call with Informix
- Stored procedure call with Genero DB
- Stored procedure call with Oracle
- Stored procedure call with DB2 UDB
- Stored procedure call with SQL Server
- Stored procedure call with PostgreSQL
- Stored procedure call with MySQL

Stored procedure call with Informix

Informix distinguishes *stored procedures* from *stored functions*. Both must be written in the Informix stored procedure language called SPL.

Stored functions returning values

There is no output parameter concept for typical SPL stored procedures or functions. If you want to return values from a database routine, you must use a *stored function* with a `RETURNING` clause. Informix *stored procedures* do not return values.

To execute a stored function with Informix, you must use the `EXECUTE FUNCTION` SQL instruction:

```
14    PREPARE stmt FROM "execute function proc1(?)"
```

In order to retrieve returning values into program variables, you must use an `INTO` clause in the EXECUTE instruction.

The following example shows how to call a stored function with Informix:

```
01 MAIN
02    DEFINE n INTEGER
03    DEFINE d DECIMAL(6,2)
04    DEFINE c VARCHAR(200)
05    DATABASE test1
06    EXECUTE IMMEDIATE "create function proc1( p1 integer )"
07                   || " returning decimal(6,2), varchar(200);"
08                   || "  define p2 decimal(6,2);"
09                   || "  define p3 varchar(200);"
10                   || "  let p2 = p1 + 0.23;"
11                   || "  let p3 = 'Value = ' || p1;"
12                   || "  return p2, p3;"
13                   || " end function;"
14    PREPARE stmt FROM "execute function proc1(?)"
15    LET n = 111
16    EXECUTE stmt USING n INTO d, c
17    DISPLAY d
18    DISPLAY c
19 END MAIN
```

Stored procedure call with Genero DB

Genero DB implements *stored procedures* as a group of statements that you can call by name. A subset of RDBMS-specific languages are supported by Genero DB; you can write Genero DB stored procedures in Informix SPL, Oracle PL/SQL or SQL Server Transact-SQL.

Stored procedures with output parameters

Genero DB *stored procedures* must be called with the input and output parameters specification in the `USING` clause of the EXECUTE, OPEN or FOREACH instruction. As in normal dynamic SQL, parameters must correspond by position, and the `IN/OUT/INOUT` options must match the parameter definition of the stored procedure.

To execute the stored procedure, you must use the `CALL` SQL instruction:

```
11   PREPARE stmt FROM "call proc1(?,?,?)"
```

Here is a complete example creating and calling a stored procedure:

```
01 MAIN
02    DEFINE n INTEGER
03    DEFINE d DECIMAL(6,2)
04    DEFINE c VARCHAR(200)
05    DATABASE test1
06    EXECUTE IMMEDIATE "create procedure proc1( p1 in int, p2 out
number(6,2), p3 in out varchar2 )"
07                  || " is begin"
08                  || "  p2 := p1 + 0.23;"
09                  || "  p3 := 'Value = ' || p1;"
10                  || "end;"
11    PREPARE stmt FROM "call proc1(?,?,?)"
12    LET n = 111
13    EXECUTE stmt USING n IN, d OUT, c INOUT
14    DISPLAY d
15    DISPLAY c
16 END MAIN
```

Stored procedures producing a result set

With Genero DB, you can execute stored procedures returning a result set. To do so, you must declare a cursor and fetch the rows:

```
01 MAIN
02    DEFINE i, n INTEGER
03    DEFINE d DECIMAL(6,2)
04    DEFINE c VARCHAR(200)
05    DATABASE test1
06    CREATE TABLE tab1 ( c1 INTEGER, c2 DECIMAL(6,2), c3 VARCHAR(200)
)
07    INSERT INTO tab1 VALUES ( 1, 123.45, 'aaaaaa' )
08    INSERT INTO tab1 VALUES ( 2, 123.66, 'bbbbbbbbb' )
09    INSERT INTO tab1 VALUES ( 3, 444.77, 'cccccc' )
10    EXECUTE IMMEDIATE "create procedure proc2 @key integer"
11                  || " as begin"
12                  || "    select * from tab1 where c1 > @key"
13                  || "  end"
14    DECLARE curs CURSOR FROM "call proc2(?)"
15    LET i = 1
16    FOREACH curs USING i INTO n, d, c
```

```
17        DISPLAY n, d, c
18    END FOREACH
19 END MAIN
```

Stored procedures with output parameters and result set

It is possible to execute Genero DB stored procedures with output parameters and a result set. The output parameter values are available after the OPEN cursor instruction:

```
01    OPEN curs USING n IN, d OUT, c INOUT
02    FETCH curs INTO rec.*
```

Stored procedure call with Oracle

Oracle supports *stored procedures* and *stored functions* as a group of PL/SQL statements that you can call by name. Oracle *stored functions* are very similar to *stored procedures*, except that a function returns a value to the environment in which it is called. Functions can be used in SQL expressions.

Stored procedures with output parameters

Oracle *stored procedures* or *stored functions* must be called with the input and output parameters specification in the `USING` clause of the EXECUTE, OPEN or FOREACH instruction. As in normal dynamic SQL, parameters must correspond by position, and the `IN/OUT/INOUT` options must match the parameter definition of the stored procedure.

To execute the *stored procedure*, you must include the procedure in an anonymous PL/SQL block with `BEGIN` and `END` keywords:

```
11   PREPARE stmt FROM "begin proc1(?,?,?); end;"
```

Remark: Oracle stored procedures do not specify the size of number and character parameters. The size of output values (especially character strings) are defined by the calling context (i.e. the data type of the variable used when calling the procedure). When you pass a CHAR(10) to the procedure, the returning value will be filled with blanks to reach a size of 10 bytes.

Warning: For technical reasons, the Oracle driver uses dynamic binding with OCIBindDynamic(). The Oracle Call Interface does not support stored procedures parameters with the CHAR data type when using dynamic binding. You must use VARCHAR2 instead of CHAR to define character string parameters for stored procedures.

Here is a complete example creating and calling a stored procedure with output parameters:

```
01 MAIN
02    DEFINE n INTEGER
```

```
03    DEFINE d DECIMAL(6,2)
04    DEFINE c VARCHAR(200)
05    DATABASE test1
06    EXECUTE IMMEDIATE "create procedure proc1( p1 in int, p2 in out
number, p3 in out varchar2 )"
07                  || " is begin"
08                  || "  p2 := p1 + 0.23;"
09                  || "  p3 := 'Value = ' || to_char(p1);"
10                  || "end;"
11    PREPARE stmt FROM "begin proc1(?,?,?); end;"
12    LET n = 111
13    EXECUTE stmt USING n IN, d INOUT, c INOUT
14    DISPLAY d
15    DISPLAY c
16 END MAIN
```

Stored functions with a return value

To execute the *stored function* returning a value, you must include the function in an anonymous PL/SQL block with `BEGIN` and `END` keywords, and use an assignment expression to specify the place holder for the returning value:

```
11    PREPARE stmt FROM "begin ? := func1(?,?,?); end;"
```

Stored procedures producing a result set

Oracle supports result set generation from stored procedures with the concept of cursor variables (`REF CURSOR`).

Warning: Genero does not support cursor references produced by Oracle stored procedures or functions.

Stored procedure call with IBM DB2

IBM DB2 implements *stored procedures* as a saved collection of SQL statements, which can accept and return user-supplied parameters. IBM DB2 *stored procedures* can also produce one or more result sets. Beside *stored procedures*, IBM DB2 supports *user defined functions*, typically used to define scalar functions returning a simple value which can be part of SQL expressions.

Stored procedures with output parameters

IBM DB2 *stored procedures* must be called with the input and output parameters specification in the `USING` clause of the EXECUTE, OPEN or FOREACH instruction. As in normal dynamic SQL, parameters must correspond by position and the `IN/OUT/INOUT` options must match the parameter definition of the stored procedure.

To execute the stored procedure, you must use the `CALL` SQL instruction:

```
11    PREPARE stmt FROM "call proc1(?,?,?)"
```

Here is a complete example creating and calling a stored procedure with output parameters:

```
01 MAIN
02    DEFINE n INTEGER
03    DEFINE d DECIMAL(6,2)
04    DEFINE c VARCHAR(200)
05    DATABASE test1
06    EXECUTE IMMEDIATE "create procedure proc1( in p1 int, out p2
decimal(6,2), inout p3 varchar(20) )"
07                  || " language sql begin"
08                  || "  set p2 = p1 + 0.23;"
09                  || "  set p3 = 'Value = ' || char(p1);"
10                  || "end"
11    PREPARE stmt FROM "call proc1(?,?,?)"
12    LET n = 111
13    EXECUTE stmt USING n IN, d OUT, c INOUT
14    DISPLAY d
15    DISPLAY c
16 END MAIN
```

Stored procedures producing a result set

With DB2 UDB, you can execute stored procedures returning a result set. To do so, you must declare a cursor and fetch the rows:

```
01 MAIN
02    DEFINE i, n INTEGER
03    DEFINE d DECIMAL(6,2)
04    DEFINE c VARCHAR(200)
05    DATABASE test1
06    CREATE TABLE tab1 ( c1 INTEGER, c2 DECIMAL(6,2), c3 VARCHAR(200)
)
07    INSERT INTO tab1 VALUES ( 1, 123.45, 'aaaaaa' )
08    INSERT INTO tab1 VALUES ( 2, 123.66, 'bbbbbbbbb' )
09    INSERT INTO tab1 VALUES ( 3, 444.77, 'cccccc' )
10    EXECUTE IMMEDIATE "create procedure proc2( in key integer )"
11                  || " result sets 1"
12                  || " language sql"
13                  || "  begin"
14                  || "  declare c1 cursor with return for"
15                  || "    select * from tab1 where c1 > key;"
16                  || "  open c1;"
17                  || "  end"
18    DECLARE curs CURSOR FROM "call proc2(?)"
19    LET i = 1
20    FOREACH curs USING i INTO n, d, c
21        DISPLAY n, d, c
22    END FOREACH
23 END MAIN
```

Stored procedures with output parameters and result set

It is possible to execute DB2 UDB stored procedures with output parameters and a result set. The output parameter values are available after the OPEN cursor instruction:

```
01    OPEN curs USING n IN, d OUT, c INOUT
02    FETCH curs INTO rec.*
```

Stored procedure call with Microsoft SQL Server

SQL Server implements *stored procedures*, which are a saved collection of Transact-SQL statements that can take and return user-supplied parameters. SQL Server *stored procedures* can also produce one or more result sets.

Stored procedures with output parameters

SQL Server *stored procedures* must be called with the input and output parameters specification in the `USING` clause of the EXECUTE, OPEN or FOREACH instruction. As in normal dynamic SQL, parameters must correspond by position and the `IN/OUT/INOUT` options must match the parameter definition of the stored procedure.

To execute the stored procedure, you must use an ODBC `call` escape sequence:

```
    PREPARE stmt FROM "{ call proc1(?,?,?) }"
```

Here is a complete example creating and calling a stored procedure with output parameters:

```
01 MAIN
02    DEFINE n INTEGER
03    DEFINE d DECIMAL(6,2)
04    DEFINE c VARCHAR(200)
05    DATABASE test1
06    EXECUTE IMMEDIATE "create procedure proc1 @v1 integer, @v2
decimal(6,2) output, @v3 varchar(20) output"
07                  || " as begin"
08                  || "  set @v2 = @v1 + 0.23"
09                  || "  set @v3 = 'Value = ' || cast(@v1 as varchar)"
10                  || "end"
11    PREPARE stmt FROM "{ call proc1(?,?,?) }"
12    LET n = 111
13    EXECUTE stmt USING n IN, d OUT, c OUT
14    DISPLAY d
15    DISPLAY c
16 END MAIN
```

Stored procedures producing a result set

With SQL Server, you can execute stored procedures returning a result set. To do so, you must declare a cursor and fetch the rows.

Warning: The following example uses a stored procedure with a simple SELECT statement. If the stored procedure contains additional Transat-SQL statements such as SET or IF (which is the case in complex stored procedures), SQL Server generates multiple result sets. By default the Genero MSV driver uses "Server Cursors" to support multiple active SQL statements. But SQL Server stored procedures generating multiple result sets cannot be used with Server Cursors: The Server Cursor is silently converted to a "Default Result Set" cursor by the ODBC driver. Since Default Result Set cursors do not support multiple active statements, you cannot use another SQL statement while processing the results of such stored procedure. You must CLOSE the cursor created for the stored procedure before continuing with other SQL statements.

```
01 MAIN
02    DEFINE i, n INTEGER
03    DEFINE d DECIMAL(6,2)
04    DEFINE c VARCHAR(200)
05    DATABASE test1
06    CREATE TABLE tab1 ( c1 INTEGER, c2 DECIMAL(6,2), c3 VARCHAR(200)
)
07    INSERT INTO tab1 VALUES ( 1, 123.45, 'aaaaaa' )
08    INSERT INTO tab1 VALUES ( 2, 123.66, 'bbbbbbbbb' )
09    INSERT INTO tab1 VALUES ( 3, 444.77, 'cccccc' )
10    EXECUTE IMMEDIATE "create procedure proc2 @key integer"
11                  || "  as select * from tab1 where c1 > @key"
12    DECLARE curs CURSOR FROM "{ call proc2(?) }"
13    LET i = 1
14    FOREACH curs USING i INTO n, d, c
15        DISPLAY n, d, c
16    END FOREACH
17 END MAIN
```

Stored procedures returning a cursor as output parameter

SQL Server supports "Cursor Output Parameters": A stored procedure can declare/open a cursor and return a reference of the cursor to the caller.

Warning: SQL Server stored procedures returning a cursor as output parameter are not supported. There are two reasons for this: The Genero language does not have a data type to store a server cursor reference, and the underlying ODBC driver does not support this anyway.

Stored procedures with return code

SQL Server stored procedures can return integer values. To get the return value of a stored procedure, you must use an assignment expression in the ODBC `call` escape sequence:

```
01    PREPARE stmt FROM "{ ? = call proc3(?,?,?) }"
```

Stored procedures with output parameters, return code and result set

With SQL Server stored procedures, you call stored procedures with a return code, output parameters and producing a result set.

Warning: Return codes and output parameters are the last items returned to the application by SQL Server; they are not returned until the last row of the result set has been fetched, after the SQLMoreResults() ODBC function is called. If output parameters are used, the SQL Server driver executes a SQLMoreResult() call when closing the cursor instead of SQLCloseCursor(), to get the return code and output parameter values from SQL Server.

```
01 MAIN
02    DEFINE r, i, n INTEGER
03    DEFINE d DECIMAL(6,2)
04    DEFINE c VARCHAR(200)
05    DATABASE test1
06    CREATE TABLE tab1 ( c1 INTEGER, c2 DECIMAL(6,2), c3 VARCHAR(200)
)
07    INSERT INTO tab1 VALUES ( 1, 123.45, 'aaaaaa' )
08    INSERT INTO tab1 VALUES ( 2, 123.66, 'bbbbbbbbb' )
09    INSERT INTO tab1 VALUES ( 3, 444.77, 'cccccc' )
10    EXECUTE IMMEDIATE "create procedure proc3 @key integer output"
11                   || "  as begin"
12                   || "     set @key = @key - 1"
13                   || "     select * from tab1 where c1 > @key"
14                   || "     return (@key * 3)"
15                   || "  end"
16    DECLARE curs CURSOR FROM "{ ? = call proc3(?) }"
17    LET i = 1
18    OPEN curs USING r INOUT, i INOUT
19    DISPLAY r, i
20    FETCH curs INTO n, d, c
21    FETCH curs INTO n, d, c
22    FETCH curs INTO n, d, c
23    DISPLAY r, i
24    CLOSE curs
25    DISPLAY r, i -- Now the returned values are available
26 END MAIN
```

Warning: Return code and output parameter variables must be defined as `INOUT` in the OPEN instruction.

Stored procedure call with PostgreSQL

PostgreSQL implements *stored functions* that can return values. If the function returns more that one value, you must specify the returning values as function parameters with the `OUT` keyword. If the function returns a unique value, you can use the `RETURNS` clause.

Warning: Pay attention to the function signature; PostgreSQL allows function overloading. For example, func(int) and func(char) are two different functions. To drop a function, you must specify the parameter type to identify the function signature properly.

Stored functions with output parameters

To execute a stored function with PostgreSQL, you must use `SELECT * FROM function`, as shown in the next line:

```
14  PREPARE stmt FROM "select * from proc1(?)"
```

In order to retrieve returning values into program variables, you must use an `INTO` clause in the EXECUTE instruction.

The following example shows how to call a stored function with PostgreSQL:

```
01 MAIN
02    DEFINE n INTEGER
03    DEFINE d DECIMAL(6,2)
04    DEFINE c VARCHAR(200)
05    DATABASE test1
06    EXECUTE IMMEDIATE "create function proc1(p1 integer, out p2
numeric(6,2), out p3 varchar(200))"
08                  || " as $$"
09                  || "  begin"
10                  || "    p2 := p1 + 0.23;"
11                  || "    p3 := 'Value = ' || cast(p1 as text);"
12                  || "  end;"
13                  || " $$ language plpgsql"
14    PREPARE stmt FROM "select * from proc1(?)"
15    LET n = 111
16    EXECUTE stmt USING n INTO d, c
17    DISPLAY d
18    DISPLAY c
19 END MAIN
```

Stored functions producing a result set

With PostgreSQL, you can execute stored procedures returning a result set. To do so, you must declare a cursor and fetch the rows:

```
01 MAIN
02    DEFINE i, n INTEGER
03    DEFINE d DECIMAL(6,2)
04    DEFINE c VARCHAR(200)
05    DATABASE test1
06    CREATE TABLE tab1 ( c1 INTEGER, c2 DECIMAL(6,2), c3 VARCHAR(200)
)
07    INSERT INTO tab1 VALUES ( 1, 123.45, 'aaaaaa' )
08    INSERT INTO tab1 VALUES ( 2, 123.66, 'bbbbbbbbb' )
09    INSERT INTO tab1 VALUES ( 3, 444.77, 'cccccc' )
10    EXECUTE IMMEDIATE "create function proc2(integer)"
```

```
11                 || " returns setof tab1"
12                 || " as $$"
13                 || "  select * from tab1 where c1 > $1;"
14                 || " $$ language sql"
15    DECLARE curs CURSOR FROM "select * from proc2(?)"
16    LET i = 1
17    FOREACH curs USING i INTO n, d, c
18        DISPLAY n, d, c
19    END FOREACH
20 END MAIN
```

Stored functions with output parameters and result set

Warning: With PostgreSQL you cannot return output parameters and a result set from the same stored procedure; both use the same technique to return values to the client, in the context of result columns to be fetched.

Stored procedure call with MySQL

MySQL implements *stored procedures* and *stored functions* as a collection of SQL statements that can take and return user-supplied parameters. Functions are very similar to procedures, except that they return a scalar value and can be used in SQL expressions.

Stored procedures with output parameters

Warning: Since MySQL C API (version 5.0) does not support an output parameter specification, the IN / OUT / INOUT technique cannot be used.

In order to return values from a MySQL *stored procedure* or *stored function*, you must use SQL variables. There are three steps to execute the procedure or function:

1. With the `SET` SQL statement, create and assign an SQL variables for each parameter.
2. `CALL` the stored procedure or stored function with the created SQL variables.
3. Perform a `SELECT` statement to return the SQL variables to the application.

In order to retrieve returning values into program variables, you must use an `INTO` clause in the EXECUTE instruction.

The following example shows how to call a stored procedure with output parameters:

Warning: MySQL version 5.0 does not allow you to prepare the CREATE PROCEDURE statement; you may need to execute this statement from the mysql command line tool.

Warning: MySQL version 5.0 cannot execute "SELECT @variable" with server-side cursors. Since the Genero MySQL driver uses server-side cursors to support

multiple active result sets, it is not possible to execute the SELECT statement to return output parameter values.

```
01 MAIN
02    DEFINE n INTEGER
03    DEFINE d DECIMAL(6,2)
04    DEFINE c VARCHAR(200)
05    DATABASE test1
06    EXECUTE IMMEDIATE "create procedure proc1(p1 integer, out p2
numeric(6,2), out p3 varchar(200))"
07                  || " no sql begin"
08                  || "    set p2 = p1 + 0.23;"
09                  || "    set p3 = concat( 'Value = ', p1 );"
10                  || "  end;"
11    LET n = 111
12    EXECUTE IMMEDIATE "set @p1 = ", n
13    EXECUTE IMMEDIATE "set @p2 = NULL"
14    EXECUTE IMMEDIATE "set @p3 = NULL"
15    EXECUTE IMMEDIATE "call proc1(@p1, @p2, @p3)"
16    PREPARE stmt FROM "select @p2, @p3"
17    EXECUTE stmt INTO d, c
18    DISPLAY d
19    DISPLAY c
20 END MAIN
```

Stored functions returning values

The following example shows how to retrieve the return value of a stored function with MySQL:

Warning: MySQL version 5.0 does not allow you to prepare the CREATE FUNCTION statement; you may need to execute this statement from the mysql command line tool.

```
01 MAIN
02    DEFINE n INTEGER
03    DEFINE c VARCHAR(200)
04    DATABASE test1
05    EXECUTE IMMEDIATE "create function func1(p1 integer)"
06                  || " no sql begin"
07                  || "    return concat( 'Value = ', p1 );"
08                  || "  end;"
09    PREPARE stmt FROM "select func1(?)"
10    LET n = 111
11    EXECUTE stmt USING n INTO c
12    DISPLAY c
13 END MAIN
```

Stored procedures producing a result set

Warning: The MySQL version 5.0 stored procedures and stored functions cannot return a result set.

1.5 Cursors and Connections

With Genero you can connect to several database sources from the same program by using the CONNECT instruction. When connected, you can DECLARE cursors or PREPARE statements, which can be used in parallel as long as you follow the rules. This section describes how to use SQL cursors and SQL statements in a multiple-connection program.

For convenience, the term *Prepared SQL Statement* and *Declared Cursor* will be grouped as *SQL handle;* from an internal point of view, both concepts merge into a unique *SQL Handle,* an object provided to manipulate SQL statements.

When you DECLARE a cursor or when you PREPARE a statement, you actually create an *SQL Handle*; the runtime system allocates resources for that SQL Handle before sending the SQL text to the database server via the database driver.

The SQL Handle is created in the context of the current connection, and must be used in that context, until it is freed or re-created with another DECLARE or PREPARE. If you try to use an SQL Handle in a different connection context than the one for which it was created, you will get a runtime error.

To change the current connection context, you must use the SET CONNECTION instruction. To set a specific connection, you must identify it by a name. To identify a connection, you typically use the AS clause of the CONNECT instruction. If you don't use the AS clause, the connection gets a default name based on the data source name. Since this might change as the database name changes, it is best to use an explicit name with the AS clause.

This small program example illustrates the use of two cursors with two different connections:

```
01 MAIN
02    CONNECT TO "db1" AS "s1"
03    CONNECT TO "db2" AS "s2"
04    SET CONNECTION "s1"
05    DECLARE c1 CURSOR FOR SELECT tab1.* FROM tab1
06    SET CONNECTION "s2"
07    DECLARE c2 CURSOR FOR SELECT tab1.* FROM tab1
08    SET CONNECTION "s1"
09    OPEN c1
10    SET CONNECTION "s2"
11    OPEN c2
12    ...
13 END MAIN
```

The DECLARE and PREPARE instructions are a type of creator instructions; if an SQL Handle is re-created in a connection other than the original connection for which it was created, old resources are freed and new resources are allocated in the current connection context.

This allows you to re-execute the same cursor code in different connection contexts, as in the following example:

```
01 MAIN
02    CONNECT TO "db1" AS "s1"
03    CONNECT TO "db2" AS "s2"
04    SET CONNECTION "s1"
05    IF checkForOrders() > 0 ...
06    SET CONNECTION "s2"
05    IF checkForOrders() > 0 ...
08    ...
09 END MAIN
10
11 FUNCTION checkForOrders(d)
12    DEFINE d DATE, i INTEGER
13    DECLARE c1 CURSOR FOR SELECT COUNT(*) FROM orders WHERE ord_date
= d
14    OPEN c1
15    FETCH c1 INTO i
16    CLOSE c1
17    FREE c1
18    RETURN i
19 END FUNCTION
```

If the SQL handle was created in a different connection, the resources used in the old connection context are freed automatically, and new SQL Handle resources are allocated in the current connection context.

1.6 SQL Error identification

You can centralize SQL error identification in a BDL function:

```
01 CONSTANT SQLERR_FATAL = -1
02 CONSTANT SQLERR_LOCK  = -2
03 CONSTANT SQLERR_CONN  = -3
```

(constants must be defined in GLOBALS)

```
04 FUNCTION identifySqlError()
05    CASE
06      WHEN SQLCA.SQLCODE == -201 OR SQLCA.SQLERRD[2] == ...
07         RETURN SQLERR_FATAL
08      WHEN SQLCA.SQLCODE == -263 OR SQLCA.SQLERRD[2] == ...
09         RETURN SQLERR_LOCK
10      ...
11    END CASE
12 END FUNCTION
```

-

```
The generic Informix error code is stored in
  SQLCA.SQLCODE register.
```

-

```
The native Database Provider error code is stored in
SQLCA.SQLERRD[2]
  register.
```

If really needed, this would also allow adding a database specific test.

---

2. Performance

2.1 Using Dynamic SQL

Although BDL allows you to write SQL statements directly in the program source as a part of the language (Static SQL), it is strongly recommended that you use Dynamic SQL instead when you are executing SQL statements within large program loops. Dynamic SQL allows you to PREPARE the SQL statements once and EXECUTE N times, improving performance.

To perform Static SQL statement execution, the database interface must use the basic API functions provided by the database vendor. These are usually equivalent to the `PREPARE` and `EXECUTE` instructions. So when you write a Static SQL statement in your BDL program, it is actually converted to a `PREPARE` + `EXECUTE`.

For example, the following BDL code:

```
01 FOR n=1 TO 100
02    INSERT INTO tab VALUES ( n, c )
03 END FOR
```

is actually equivalent to:

```
01 FOR n=1 TO 100
02    PREPARE s FROM "INSERT INTO tab VALUES ( ?, ? )"
03    EXECUTE s USING n, c
04 END FOR
```

To improve the performance of the preceding code, use a PREPARE instruction before the loop and put an EXECUTE instruction inside the loop:

```
01 PREPARE s FROM "INSERT INTO tab VALUES ( ?, ? )"
02 FOR n=1 TO 100
03    EXECUTE s USING n, c
```

```
04 END FOR
```

2.2 Using transactions

When you use an ANSI compliant RDBMS like **Oracle** or **DB2**, the database interface must perform a COMMIT after each statement execution. This generates unnecessary database operations and can slow down big loops. To avoid this implicit COMMIT, you can control the transaction with BEGIN WORK / COMMIT WORK around the code containing a lot of SQL statement execution.

For example, the following loop will generate 2000 basic SQL operations (1000 INSERTs plus 1000 COMMITs):

```
01 PREPARE s FROM "INSERT INTO tab VALUES ( ?, ? )"
01 FOR n=1 TO 100
03    EXECUTE s USING n, c   -- Generates implicit COMMIT
04 END FOR
```

You can improve performance if you put a transaction block around the loop:

```
01 PREPARE s FROM "INSERT INTO tab VALUES ( ?, ? )"
02 BEGIN WORK
03 FOR n=1 TO 100
04    EXECUTE s USING n, c   -- In transaction -> no implicit COMMIT
05 END FOR
06 COMMIT WORK
```

With this code, only 1001 basic SQL operations will be executed (1000 INSERTs plus 1 COMMIT).

However, you must take care when generating large transactions because all modifications are registered in transaction logs. This can result in a lack of database server resources (INFORMIX "transaction too long" error, for example) when the number of operations is very big. If the SQL operation does not require a unique transaction for database consistency reasons, you can split the operation into several transactions, as in the following example:

```
01 PREPARE s FROM "INSERT INTO tab VALUES ( ?, ? )"
02 BEGIN WORK
03 FOR n=1 TO 100
04    IF n MOD 10 == 0 THEN
05       COMMIT WORK
06       BEGIN WORK
07    END IF
08    EXECUTE s USING n, c   -- In transaction -> no implicit COMMIT
09 END FOR
10 COMMIT WORK
```

2.3 Avoiding long transactions

Some BDL applications do not care about long transactions because they use an Informix database without transaction logging (transactions are not stored in log files for potential rollbacks). However, if a failure occurs, no rollback can be made, and only some of the rows of a query might be updated. This could result in data inconsistency !

With many providers (Genero DB, SQL Server, IBM DB2, Oracle…), using transactions is mandatory. Every database modification is stored in a log file.

BDL applications must prevent long transactions when connected to a database using logging. If a table holds hundreds of thousands of rows, a "DELETE FROM table", for example, might cause problems. If the transaction log is full, no other insert, update or delete could be made on the database. The activity could be stopped until a backup or truncation of the log !

For example, if a table holds hundreds of thousands of rows, a "DELETE FROM table" might produce a "snapshot too old" error in ORACLE if the rollback segments are too small.

Solution :

You must review the program logic in order to avoid long transactions:

- keep transactions as short as possible.
- access the least amount of data possible while in a transaction. If possible, avoid using a big SELECT in transaction.
- split a long transaction into many short transactions. Use a loop to handle each block (see the last example below : 2.2 Using transactions).
- to delete all rows from a table use the "TRUNCATE TABLE" instruction instead of "DELETE FROM" (Not for all vendors)

In the end, increase the size of the transaction log to avoid it filling up.

2.4 Declaring prepared statements

Line 2 of the following example shows a cursor declared with a prepared statement:

```
01 PREPARE s FROM "SELECT * FROM table WHERE ", condition
02 DECLARE c CURSOR FOR s
```

While this has no performance impact with Informix database drivers, it can become a bottleneck when using non-Informix databases:

Statement preparation consumes a lot of memory and processor resources. Declaring a cursor with a prepared statement is a native Informix feature, which does consume only one real statement preparation. But non-Informix databases do not support this feature.

So the statement is prepared twice (once for the PREPARE, and once for the DECLARE). When used in a big loop, such code can cause performance problems.

To optimize such code, you can use the FROM clause in the DECLARE statement:

```
01 DECLARE c CURSOR FROM "SELECT * FROM table WHERE " || condition
```

By using this solution only one statement preparation will be done by the database server.

Remark: This performance problem does not appear with DECLARE statements using static SQL.

2.5 Saving SQL resources

To write efficient SQL in a Genero program, you should use Dynamic SQL as described in 2.1 of this performance section. However, when using Dynamic SQL, you allocate an SQL statement handle on the client and server side, consuming resources. According to the database type, this can be a few bytes or a significant amount of memory. For example, on a Linux 32b platform, a prepared statement costs about 5 Kbytes with an Informix CSDK 2.80 client, while it costs about 20 Kbytes with an Oracle 10g client. That can be a lot of memory if you have programs declaring a dozen or more cursors, multiplied by hundreds of user processes. When executing several Static SQL statements, the same statement handle is reused and thus less memory is needed.

Genero allows you to use either Static or Dynamic SQL, so it's in your hands to choose memory or performance. However, in some cases the same code will be used by different kinds of programs, needing either low resource usage or good performance. In many OLTP applications you can actually distinguish two type of programs:

- Programs where memory usage is not a problem but needing good performance (typically, batch programs executed as a unique instance during the night).
- Programs where performance is less important but where memory usage must be limited (typically, interactive programs executed as multiple instances for each application user).

To reuse the same code for interactive programs and batch programs, you can do the following:

1. Define a local module variable as an indicator for the prepared statement.
2. Write a function returning the type of program (for example, 'interactive' or 'batch' mode).
3. Then, in a reusable function using SQL statements, you prepare and free the statement according to the indicators, as shown in the next example.

```
01 DEFINE up_prepared INTEGER
02
03 FUNCTION getUserPermissions( username )
```

```
04   DEFINE username VARCHAR(20)
05   DEFINE cre, upd, del CHAR(1)
06
07   IF NOT up_prepared THEN
08      PREPARE up_stmt FROM "SELECT can_create, can_update,
cab_delete"
09                           || " FROM user_perms WHERE name = ?"
10      LET up_prepared = TRUE
11   END IF
12
13   EXECUTE up_stmt USING username INTO cre, upd, del
14
15   IF isInteractive() THEN
16      FREE up_stmt
17      LET up_prepared = FALSE
18   END IF
19
20   RETURN cre, upd, del
21
22 END FUNCTION
```

The first time this function is called, the `up_prepared` value will be FALSE, so the statement will be prepared in line 08. The next time the function is called, the statement will be re-prepared only if `up_prepared` is TRUE. The statement is executed in line 13 and values are fetch into the variables returned in line 20. If the program is interactive, lines 15 to 18 free the statement and set the `up_prepared` module variable back to FALSE, forcing statement preparation in the next call of this function.

---

3. Portability

Writing portable SQL is mandatory if you want to succeed with different kind of database servers. This section gives you some hints to solve SQL incompatibility problems in your programs. Read this section carefully and review your program source code if needed. You should also read carefully the ODI Adaptation Guides which contain detailed information about SQL compatibility issues.

To easily detect SQL statements with specific syntax, you can use the -W stdsql option of fglcomp:

```
$ fglcomp -W stdsql orders.4gl
module.4gl:15: SQL Statement or language instruction with specific SQL
syntax.
```

Remark: This compiler option can only detect non-portable SQL syntax in Static SQL statements.

3.1 Database entities

Most database servers can handle multiple database entities (you can create multiple 'databases'), but this is not possible with all engines:

Database Server Type	Multiple Database Entities
GeneroDB	No
IBM DB2 UDB (Unix)	Yes
Informix	Yes
Microsoft SQL Server	Yes
MySQL	Yes
Oracle Database Server	No
PostgreSQL	Yes
Sybase ASA	Yes

When using a database server that does not support multiple database entities, you can emulate different databases with schemas, but this requires you to check for the database user definition. Each database user must have privileges to access any schema, and to see any table of any schema without needing to set a schema prefix before table names in SQL statements.

You can force the database driver to set a specific schema at connection with the following FGLPROFILE entry:

```
dbi.database.<dbname>.schema = "<schema-name>"
```

Some databases like **GeneroDB** also allow you to define a default schema for each database user. When the user connects to the database, the default schema is automatically selected.

3.2 Database users and security

To get the benefit of the database server security features, you should identify each physical user as a database user.

According to the type of server, you must do the following steps to create a database user:

1. Define the user as an operating system user.
2. Declare the user in the database server.
3. Grant database access privileges.

Each database server has its specific users management and data access privilege mechanisms. Check the vendor documentation for security features and make sure you can define the users, groups, and privileges in all database servers you want to use.

3.3 Database creation statements

Database creation statements like CREATE DATABASE, CREATE DBSPACE, DROP DATABASE cannot be executed with **ODI** database drivers. Such high-level database management statements are Informix-specific and require a connection to the server with no current database selected, which is not supported by the ODI architecture and drivers. However, for compatibility, the standard Informix drivers (with **ix** prefix) allow the BDL programs to execute such statements.

3.4 Data definition statements

When using Data Definition Statements like CREATE TABLE, ALTER TABLE, DROP TABLE, only a limited SQL syntax works on all database servers. Most databases support NOT NULL, CHECK, PRIMARY KEY, UNIQUE, FOREIGN KEY constraints, but the syntax for naming constraints is different.

The following statement works with most database servers and creates a table with equivalent properties in all cases:

```
CREATE TABLE customer
(
 cust_id INTEGER NOT NULL,
 cust_name CHAR(50) NOT NULL,
 cust_lastorder DATE NOT NULL,
 cust_group INTEGER,
   PRIMARY KEY (cust_id),
   UNIQUE (cust_name),
   FOREIGN KEY (cust_group) REFERENCES group (group_id)
)
```

Warning: Some engines like SQL Server have a different default behavior for NULL columns when you create a table. You may need to set up database properties to make sure that a column allows NULLs if the NOT NULL constraint is not specified.

When you want to create tables in programs using non-standard clauses (for example to define storage options), you must use Dynamic SQL and adapt the statement to the target database server.

3.5 Using portable data types

The ANSI SQL specification defines standard data types, but for historical reasons most databases vendors have implemented native (non-standard) data types. You can usually use a synonym for ANSI types, but the database server always uses the native types behind the scenes. For example, when you create a table with an INTEGER column in **Oracle**, the native NUMBER data type is used. In your programs, avoid BDL data types that do not have a native equivalent in the target database. This includes simple types like floating point numbers, as well as complex data types like INTERVAL. Numbers may cause rounding or overflow problems, because the values stored in the database have different limits. For DECIMALs, always use the same precision and scale for the BDL variables and the database columns.

To write portable applications, we strongly recommend using the following BDL data types only:

BDL Data Type
CHAR(n)
VARCHAR(n)
INTEGER
SMALLINT
DECIMAL(p,s)
DATE
DATETIME HOUR TO SECOND
DATETIME YEAR TO FRACTION(n)

See the ODI Adaptation Guides for more details about data type compatibility.

3.6 CHAR and VARCHAR types

The CHAR and VARCHAR types are designed to store character strings, but all database servers do not have the same behavior when comparing two CHAR or VARCHAR values having trailing spaces.

CHAR and VARCHAR columns in databases

With all kinds of databases servers, CHAR columns are always filled with blanks up to the size of the column, but trailing blanks are not significant in comparisons:

```
CHAR("abc ") = CHAR("abc")
```

With all database servers except Informix, trailing blanks are significant when comparing VARCHAR values:

```
VARCHAR("abc ") <> VARCHAR("abc")
```

This is a major issue if you mix CHAR and VARCHAR columns and variables in your SQL statements, because the result of an SQL query can be different depending on whether you are using **Informix** or another database server.

CHAR and VARCHAR variables in BDL

In BDL, CHAR variables are filled with blanks, even if the value used does not contain all spaces.

The following example:

```
01 DEFINE c CHAR(5)
02 LET c = "abc"
03 DISPLAY c || "."
```

shows the value `"abc  ."` (5 chars + dot).

In BDL, VARCHAR variables are assigned with the exact value specified, with significant trailing blanks.

For example, this code:

```
01 DEFINE v VARCHAR(5)
02 LET v = "abc "
03 DISPLAY v || "."
```

shows the value `"abc ."` (4 chars + dot).

When comparing CHAR or VARCHAR variables in a BDL expression, the trailing blanks are not significant:

```
01 DEFINE c CHAR(5)
02 DEFINE v1, v2 VARCHAR(5)
03 LET c = "abc"
04 LET v1 = "abc "
05 LET v2 = "abc  "
```

```
06 IF c == v1 THEN DISPLAY "c==v1"
07 END IF
08 IF c == v2 THEN DISPLAY "c==v2"
09 END IF
10 IF v1 == v2 THEN DISPLAY "v1==v2"
11 END IF
```

shows all three messages.

Additionally, when you assign a VARCHAR variable from a CHAR, the target variable gets the trailing blanks of the CHAR variable:

```
01 DEFINE pc CHAR(50)
02 DEFINE pv VARCHAR(50)
03 LET pc = "abc"
04 LET pv = pc
05 DISPLAY pv || "."
```

shows `"abc <47 spaces>. "` (50 chars + dot).

To avoid this, you can use the CLIPPED operator:

```
LET pv = pc CLIPPED
```

CHAR and VARCHAR variables and columns

When you insert a row containing a CHAR variable into a CHAR or VARCHAR column, the database interface removes the trailing blanks to avoid overflow problems, (insert CHAR(100) into CHAR(20) when value is `"abc"` must work).

In the following example:

```
01 DEFINE c CHAR(5)
02 LET c = "abc"
03 CREATE TABLE t ( v1 CHAR(10), v2 VARCHAR(10) )
04 INSERT INTO tab VALUES ( c, c )
```

The value in column v1 and v2 would be `"abc"` (3 chars in both columns).

When you insert a row containing a VARCHAR variable into a VARCHAR column, the VARCHAR value in the database gets the trailing blanks as set in the variable. When the column is a CHAR(N), the database server fills the value with blanks so that the size of the string is N characters.

In the following example:

```
DEFINE c VARCHAR(5)
LET c = "abc  "
```

```
CREATE TABLE t ( v1 CHAR(10), v2 VARCHAR(10) )
INSERT INTO tab VALUES ( c, c )
```

The value in column v1 would be `"abc       "` (10 chars) and v2 would be `"abc  "` (5 chars).

What should you do?

Use VARCHAR variables for VARCHAR columns, and CHAR variables for CHAR columns to achieve portability across all kinds of database servers.

See also: LENGTH() function.

3.7 Concurrent data access

Data Concurrency is the simultaneous access of the same data by many users. Data Consistency means that each user sees a consistent view of the database. Without adequate concurrency and consistency controls, data could be changed improperly, compromising data integrity. To write interoperable BDL applications, you must adapt the program logic to the behavior of the database server regarding concurrency and consistency management. This issue requires good knowledge of multi-user application programming, transactions, locking mechanisms, isolation levels and wait mode. If you are not familiar with these concepts, carefully read the documentation of each database server which covers this subject.

Processes accessing the database can change transaction parameters such as the isolation level. The main problem is to find a configuration which results in similar behavior on every database engine. Existing BDL programs must be adapted to work with this new behavior. ALL programs accessing the same database must be changed.

The following is the best configuration to get common behavior with all types of database engines :

- The database must support transactions; this is usually the case.
- Transactions must be as short as possible (a few seconds).
- The Isolation Level must be at least "Read Committed" (= "Cursor Stability").
- The Wait Mode for locks must be "WAIT" or "WAIT n" (timeout).

Remarks: With this configuration, the locking granularity does not have to be at the row level. To improve performance with **Informix** databases, you can use the "LOCK MODE PAGE" locking level, which is the default.

3.8 The SQLCA register

SQLCA is the *SQL Communication Area* variable. SQLCA is a global record predefined by the runtime system, that can be queried to get SQL status information. After executing an SQL statement, members of this record contain execution or error data, but it is specific to Informix databases. For example, after inserting a row in a table with a SERIAL column, SQLCA.SQLERRD[2] contains the new generated serial number.

SQLCA.SQLCODE will be set to a specific Informix SQL error code, if the database driver can convert the native SQL error to an Informix SQL error. Additional members may be set, but that depends on the database server and database driver.

To identify SQL errors, you can also use SQLSTATE; this variable holds normalized ISO SQL error codes. However, even if Genero database drivers are prepared to support this register, not all RDBMS support standard SQLSTATE error codes:

Database Server Type	Supports SQLSTATE errors
GeneroDB	No
IBM DB2 UDB (Unix)	Yes, since version 7.1
Informix	Yes, since IDS 10
Microsoft SQL Server	Yes, since version 8 (2000)
MySQL	Not in version 5.x
Oracle Database Server	Not in version 10.2
PostgreSQL	Yes, since version 7.4
Sybase ASA	Not in version 8.x

According to the above, SQL error identification requires quite complex code, which can be RDBMS-specific in some cases. Therefore, it is strongly recommended that you centralize SQL error identification in a function. This will allow you to write RDBMS-specific code, when needed, only once.

3.9 Optimistic Locking

This section describes how to implement *optimistic locking* in BDL applications. Optimistic locking is a portable solution to control simultaneous modification of the same record by multiple users.

Traditional Informix-based applications use a SELECT FOR UPDATE to set a lock on the row to be edited by the user. This is called *pessimistic locking*. The SELECT FOR UPDATE is executed before the interactive part of the code, as described in here:

1. When the end user chooses to modify a record, the program declares and opens a cursor with a SELECT FOR UPDATE.
   At this point, an SQL error might be raised if the record is already locked by

another process.
Otherwise, the lock is acquired and user can modify the record.
2. The user edits the current record in the input form.
3. The user validates the dialog.
4. The UPDATE SQL instruction is executed.
5. The transaction is committed or the SELECT FOR UPDATE cursor is closed. The lock is released.

Note that if the Informix database was created with transaction logging, you must either start a transaction or define the SELECT FOR UPDATE cursor WITH HOLD option.

Unfortunately, this is not a portable solution. The lock wait mode should preferably be "WAIT" for portability reasons. Pessimistic locking is based on a "NOT WAIT" mode to return control to the program if a record is already locked by another process. Therefore, following the portable concurrency model, the *pessimistic locking* mechanisms must be replaced by the *optimistic locking* technique.

Basically, instead of locking the row before the user starts to modify the record data, the *optimistic locking* technique makes a copy of the current values(i.e. Before Modification Values), lets the user edit the record, and when it's time to write data into the database, checks if the BMVs still correspond to the current values in the database:

1. A SELECT is executed to fill the record variable used by the interactive instruction for modifications.
2. The record variable is copied into a backup record to keep Before Modification Values.
3. The user enters modifications in the input form; this updates the values in the modification record.
4. The user validates the dialog.
5. A transaction is started with BEGIN WORK.
6. A SELECT FOR UPDATE is executed to put the current database values into a temporary record.
7. If the SQL status is NOTFOUND, the row has been deleted by another process, and the transaction is rolled back.
8. Otherwise, the program compares the temporary record values with the backup record values `(rec1.*==rec2.*)`
9. If these values have changed, the row has been modified by another process, and the transaction is rolled back.
10. Otherwise, the UPDATE statement is executed.
11. The transaction is committed.

To compare 2 records, simply write:

```
01 IF new_record.* != bmv_record.* THEN
02    LET values_have_changed = TRUE
03 END IF
```

The optimistic locking technique could be implemented with a unique SQL instruction: an UPDATE could compare the column values to the BMVs directly (UPDATE ... WHERE kcol = kvar AND col1 = bmv.var1 AND ...). But, this is not possible when BMVs can be

NULL. The database engine always evaluates conditional expressions such as "col=NULL" to FALSE. Therefore, you must use "col IS NULL" when the BMV is NULL. This means dynamic SQL statement generation based on the DMV values. Additionally, to use the same number of SQL parameters (? markers), you would have to use "col=?" when the BMV is not null and "col IS NULL and ? IS NULL" when the BMV is null. Unfortunately, the expression " ? IS [NOT] NULL " is not supported by all database servers (DB2 raises error SQL0418N).

If you are designing a new database application from scratch, you can also use the row versioning method. Each tables of the database must have a column that identifies the current version of the row. The column can be a simple INTEGER (to hold a row version number) or it can be a timestamp (DATETIME YEAR TO FRACTION(5) for example). To guaranty that the version or timestamp column is updated each time the row is updated, you should implement a trigger to increment the version or set the timestamp when an UPDATE statement is issued. If this is in place, you just need to check that the row version or timestamp has not changed since the user modifications started, instead of testing all field of the BMV record.

3.10 Auto-incremented columns

This section describes how to implement **auto-incremented fields** for portability.

INFORMIX provides the SERIAL data type which can be emulated by the database interface with most non-INFORMIX database engines. But, this requires additional configuration and maintenance tasks. If you plan to review the architecture of your programs, you should use this portable implementation instead of SERIALs emulated by the connectors when "ifxemul.serial" is true.

Solution 1: Use database specific serial generators.

Principle:

In accordance with the target database, you must use the appropriate native serial generation method. Get the database type with the **db_get_database_type()** function of **fgldbutl.4gl** and use the appropriate SQL statements to insert rows with serial generation.

Warning : Not all database engines provide a sequence generator. Check the documentation of your target database.

Implementation:

1. Create the database objects required for serial generation in the target database (for example, create tables with SERIAL columns in Informix, tables with IDENTITY columns in **SQL Server** and sequence generators in **Oracle**).
2. Adapt your BDL programs to use the native sequence generators in accordance with the database type.

BDL example:

```
01 DEFINE dbtype CHAR(3)
02 DEFINE t1rec RECORD
03           id    INTEGER,
04           name  CHAR(50),
05           cdate DATE
06     END RECORD
07
08 LET dbtype = db_get_database_type()
09
10 IF dbtype = "IFX" THEN
11    INSERT INTO t1 ( id, name, cdate )
12           VALUES ( 0, t1rec.name, t1rec.cdate )
13    LET t1rec.id = SQLCA.SQLERRD[2]
14 END IF
15 IF dbtype = "ORA" THEN
16    INSERT INTO t1 ( id, name, cdate )
17            VALUES ( t1seq.nextval, t1rec.name, t1rec.cdate )
18    SELECT t1seq.currval INTO t1rec.id FROM dual
19 END IF
20 IF dbtype = "MSV" THEN
21    INSERT INTO t1 ( name, cdate )
22            VALUES ( t1rec.name, t1rec.cdate )
23    PREPARE s FROM "SELECT convert(integer,@@identity)"
24    EXECUTE s INTO t1rec.id
25 END IF
```

Solution 2: Generate serial numbers by hand.

Purpose:

The goal is to generate unique INTEGER numbers. These numbers will usually be used for primary keys.

Prerequisites:

1. The database must use transactions. This is usually the case with non-INFORMIX databases, but INFORMIX databases default to auto commit mode. Make sure your INFORMIX database allows transactions.
2. The sequence generation must be called inside a transaction (BEGIN WORK / COMMIT WORK).
3. The lock wait mode must be WAIT. This is usually the case in non-INFORMIX databases, but INFORMIX defaults to NOT WAIT. You must change the lock wait mode with "SET LOCK MODE TO WAIT" or "WAIT seconds" when using INFORMIX.
4. Non-BDL applications or stored procedures must implement the same technique when inserting records in the table having auto-incremented columns.

Principle:

A dedicated table named "**SEQREG**" is used to register sequence numbers. The key is the name of the sequence. This name will usually be the table name the sequence is generated for. In short, this table contains a primary key that identifies the sequence and a column containing the last generated number.

The uniqueness is granted by the concurrency management of the database server. The first executed instruction is an UPDATE that sets an exclusive lock on the SEQREG record. When two processes try to get a sequence at the same time, one will wait for the other until its transaction is finished.

Implementation:

The "**fgldbutl.4gl**" utility library implements a function called "**db_get_sequence**" which generates a new sequence. You must create the SEQREG table as described in the **fgldbutl.4gl** source and make sure that every user has the privileges to access and modify this table.

BDL example:

```
01 DEFINE rec RECORD
02            id    INTEGER,
03            name  CHAR(100)
04      END RECORD
05 BEGIN WORK
06 LET rec.id = db_get_sequence( "CUSTID" )
07 IF rec.id>0 THEN
08   INSERT INTO CUSTOMER ( CUSTID, CUSTNAME )
09          VALUES ( rec.id, rec.name )
10 ELSE
11   ERROR "cannot get new sequence number"
12 END
13 COMMIT WORK
```

3.11 Informix SQL ANSI Mode

INFORMIX allows you to create databases in ANSI mode, which is supposed to be closer to ANSI standard behavior. Other databases like **ORACLE** and **DB2** are 'ANSI' by default.

If you are not using the ANSI mode with **Informix**, we suggest you keep the database as is, because turning an Informix database into ANSI mode can result in unexpected behavior of the programs.

Here are some ANSI mode issues extracted from the Informix books:

- Some actions, like CREATE INDEX will generate a warning but will not be forbidden.
- Buffered logging is not allowed to enforce data recovery. (Buffered logging provides better performance)

- The table-naming scheme allows different users to create tables without having to worry about name conflicts.
- Owner specification is required in database object names (SELECT ... FROM "owner".table). You must quote the owner name to prevent automatic translation of the owner name into uppercase : SELECT .. FROM owner.table -> SELECT .. FROM OWNER.table => table not found in database!
- Default privileges differ : When creating a table, the server grants privileges to the table owner and the DBA only. The same thing happens for the 'Execute' privilege when creating stored procedures.
- Default isolation level is REPEATABLE READ.
- An error is generated if any character field is filled with a value that is longer than the field width.
- DECIMAL(p) (floating point decimals) are automatically converted to DECIMAL(p,0) (fixed point decimals).
- Closing a closed cursor generates an SQL error.

It will take more time to adapt the programs to the INFORMIX ANSI mode than using the database interface to simulate the native mode of INFORMIX.

3.12 WITH HOLD and FOR UPDATE

Informix supports WITH HOLD cursors using the FOR UPDATE clause. Such cursors can remain open across transactions (when using FOR UPDATE, locks are released at the end of a transaction, but the WITH HOLD cursor is not closed). This kind of cursor is Informix-specific and not portable. The SQL standards recommend closing FOR UPDATE cursors and release locks at the end of a transaction. Most database servers close FOR UPDATE cursors when a COMMIT WORK or ROLLBACK WORK is done. All database servers release locks when a transaction ends.

Database Server Type	WITH HOLD FOR UPDATE supported?
GeneroDB	Yes (if primary key or unique index)
IBM DB2 UDB (Unix)	No
Informix	Yes
Microsoft SQL Server	No
MySQL	No
Oracle Database Server	No
PostgreSQL	No
Sybase ASA	No

It is mandatory to review code using WITH HOLD cursors with a SELECT statement having the FOR UPDATE clause.

The standard SQL solution is to declare a simple FOR UPDATE cursor outside the transaction and open the cursor inside the transaction:

```
01 DECLARE c1 CURSOR FOR SELECT ... FOR UPDATE
02 BEGIN WORK
03   OPEN c1
04   FETCH c1 INTO ...
05   UPDATE ...
06 COMMIT WORK
```

If you need to process a complete result set with many rows including updates of master and detail rows, you can declare a normal cursor and do a SELECT FOR UPDATE inside each transaction, as in the following example:

```
01 DECLARE c1 CURSOR FOR SELECT key FROM master ...
02 DECLARE c2 CURSOR FOR SELECT * FROM master WHERE key=? FOR UPDATE
03 FOREACH c1 INTO mrec.key
04   BEGIN WORK
05   OPEN c2 INTO mrec.key
06   FETCH c2 INTO rec.*
07   IF STATUS==NOTFOUND THEN
08      ROLLBACK WORK
09      CONTINUE FOREACH
10   END IF
11   UPDATE master SET ... WHERE CURRENT OF c2
12   UPDATE detail SET ... WHERE mkey=mrec.key
13   COMMIT WORK
14 END FOREACH
```

3.13 Positioned Updates/Deletes

The "WHERE CURRENT OF *cursor-name*" clause in UPDATE and DELETE statements is not supported by all database engines.

Database Server Type	WHERE CURRENT OF supported?
GeneroDB	Yes
IBM DB2 UDB (Unix)	Yes
Informix	Yes
Microsoft SQL Server	Yes
MySQL	Yes
Oracle Database Server	No, emulated by driver with ROWIDs
PostgreSQL	No, emulated by driver with OIDs
Sybase ASA	Yes

Some database drivers can emulate WHERE CURRENT OF mechanisms by using rowids, but this requires additional processing. You should review the code to disable this option.

The standard SQL solution is to use primary keys in all tables and write UPDATE / DELETE statements with a WHERE clause based on the primary key:

```
01 DEFINE rec RECORD
02             id    INTEGER,
03             name  CHAR(100)
04       END RECORD
05 BEGIN WORK
06   SELECT CUSTID FROM CUSTOMER
07          WHERE CUSTID=rec.id FOR UPDATE
08   UPDATE CUSTOMER SET CUSTNAME = rec.name
09          WHERE CUSTID = rec.id
10 COMMIT WORK
```

3.14 String literals in SQL statements

Some database servers like INFORMIX allow single and double quoted string literals in SQL statements, both are equivalent:

```
SELECT COUNT(*) FROM table
 WHERE col1 = "abc'def""ghi"
   AND col1 = 'abc''def"ghi'
```

Most database servers do not support this specific feature:

Database Server Type	Double quoted string literals
GeneroDB	No
IBM DB2 UDB (Unix)	No
Informix	Yes
Microsoft SQL Server	Yes
MySQL	No
Oracle Database Server	No
PostgreSQL	No
Sybase ASA	No

The ANSI SQL standards define doubles quotes as database object names delimiters, while single quotes are dedicated to string literals:

```
CREATE TABLE "my table" ( "column 1" CHAR(10) )
SELECT COUNT(*) FROM "my table" WHERE "column 1" = 'abc'
```

If you want to write a single quote character inside a string literal, you must write 2 single quotes:

```
... WHERE comment = 'John''s house'
```

When writing Static SQL in your programs, the double quoted string literals as converted to ANSI single quoted string literals by the fglcomp compiler. However, Dynamic SQL statements are not parsed by the compiler and therefore need to use single quoted string literals.

We recommend that you ALWAYS use single quotes for string literals and, if needed, double quotes for database object names.

3.15 Date and Time literals in SQL statements

INFORMIX allows you to specify date and time literals as a quoted character string in a specific format, depending upon DBDATE and GLS environment variables. For example, if DBDATE=DMY4, the following statement specifies a valid DATE literal:

```
SELECT COUNT(*) FROM table WHERE date_col = '24/12/2005'
```

Other database servers do support date/time literals as quoted character strings, but the date/time format specification is quite different. The parameter to specify the date/time format can be a database parameter, an environment variable, or a session option...

In order to write portable SQL, just use SQL parameters instead of literals:

```
01 DEFINE cnt INTEGER
02 DEFINE adate DATE
03 LET adate = '24/12/2005'  -- DBDATE applies because this is BDL, not
SQL!
04 SELECT COUNT(*) INTO cnt FROM table
05   WHERE date_col = adate
```

Or, when using dynamic SQL:

```
01 DEFINE cnt INTEGER
02 DEFINE adate DATE
03 LET adate = '24/12/2005'
04 PREPARE s1 FROM "SELECT COUNT(*) FROM table WHERE date_col = ?"
05 EXECUTE s1 USING adate INTO cnt
```

3.16 Naming database objects

Name syntax

Each type of database server has its own naming conventions for database objects (i.e. tables and columns):

Database Server Type	Naming Syntax
GeneroDB	[*schema.*]*identifier*
IBM DB2 UDB (Unix)	[[*database.*]*owner.*]*identifier*
Informix	[*database*[@*dbservername*]:][*owner.*]*identifier*
Microsoft SQL Server	[[[*server.*][*database*].][*owner_name*].]*object_name*
MySQL	[*database.*]*identifier*
Oracle Database Server	[*schema.*]*identifier*[@*database-link*]
PostgreSQL	[*owner.*]*identifier*
Sybase ASA	[*database.*]*identifier*

Case-sensitivity

Most database engines have case-sensitive object identifiers. In most cases, when you do not specify identifiers in double quotes, the SQL parser automatically converts names to uppercase or lowercase, so that the identifiers match if the objects are also created without double quoted identifiers.

```
CREATE TABLE Customer ( cust_ID INTEGER )
```

In **Oracle**, the above statement would create a table named "`CUSTOMER`" with a "`CUST_ID`" column.

The following table shows the case sensitivity features of each kind of database engine:

Database Server Type	Case sensitive names?	Not-quoted names converted to ...
GeneroDB	Yes	Uppercase
IBM DB2 UDB (Unix)	Yes	Uppercase
Informix (1)	No	No
Microsoft SQL Server (2)	Yes	Not converted

MySQL	Yes	Not converted
Oracle Database Server	Yes	Uppercase
PostgreSQL	No	Lowercase
Sybase ASA	No	Lowercase

(1) If not ANSI database mode.
(2) Global parameter set at installation.

Warning: You must take care with database servers marked in red, because object identifiers are case sensitive and are not converted to uppercase or lowercase if not delimited by double-quotes. This means that, by error, you can create two tables with a similar name:

```
CREATE TABLE customer ( cust_id INTEGER )  -- first table
CREATE TABLE Customer ( cust_id INTEGER )  -- second table
```

Size of identifiers

The maximum size of a table or column name depends on the database server type. Some database engines allow very large names (256c), while others support only short names (30c max). Therefore, using short names is required for writing portable SQL. Short names also simplify SQL programs.

How to write SQL with portable object identifiers

We recommend that you use simple and short (<30c) database object names, without double quotes and without a schema/owner prefix:

```
CREATE TABLE customer ( cust_id INTEGER )
SELECT customer.cust_id FROM table
```

You may need to set the database schema after connection, so that the current database user can see the application tables without specifying the owner/schema prefix each time.

Tip: Even if all database engines do not required unique column names for all tables, we recommend that you define column names with a small table prefix (for example, CUST_ID in CUSTOMER table).

3.17 Temporary tables

Not all database servers support temporary tables. The engines supporting this feature often provide it with a specific table creation statement:

Database	Temp table creation syntax	Local to

Server Type		SQL session?
GeneroDB	`CREATE TEMP TABLE tablename ( column-defs )` `SELECT ... INTO TEMP tablename`	Yes
IBM DB2 UDB (Unix)	`DECLARE GLOBAL TEMPORARY TABLE tablename ( column-defs )` `DECLARE GLOBAL TEMPORARY TABLE tablename AS ( SELECT ... )`	Yes
Informix	`CREATE TEMP TABLE tablename ( column-defs )` `SELECT ... INTO TEMP tablename`	Yes
Microsoft SQL Server	`CREATE TABLE #tablename ( column-defs )` `SELECT select-list INTO #tablename FROM ...`	Yes
MySQL	`CREATE TEMPORARY TABLE tablename ( column-defs )` `CREATE TEMPORARY TABLE tablename LIKE other-table`	Yes
Oracle Database Server	`CREATE GLOBAL TEMPORARY TABLE tablename ( column-defs )` `CREATE GLOBAL TEMPORARY TABLE tablename AS SELECT ...`	No: only data is local to session
PostgreSQL	`CREATE TEMP TABLE tablename ( column-defs )` `SELECT select-list INTO TEMP tablename FROM ...`	Yes
Sybase ASA	`CREATE GLOBAL TEMPORARY TABLE tablename ( column-defs )` `CREATE TABLE #tablename ( column-defs )`	Yes

Some databases even have a different behavior when using temporary tables. For example, **Oracle 9i** supports a kind of temporary table, but it must be created as a permanent table. The table is not specific to an SQL session: it is shared by all processes - only the data is local to a database session.

You must review the programs using temporary tables, and adapt the code to use database-specific temporary tables.

3.18 Outer joins

Old INFORMIX SQL outer joins specified with the OUTER keyword in the FROM part are not standard:

```
SELECT * FROM master, OUTER ( detail )
  WHERE master.mid = detail.mid
    AND master.cdate IS NOT NULL
```

Database Server Type	Supports Informix OUTER join syntax
GeneroDB	Yes
IBM DB2 UDB (Unix)	No (but translated by driver)
Informix (1)	Yes
Microsoft SQL Server (2)	No (but translated by driver)
MySQL	No (but translated by driver)
Oracle Database Server	No (but translated by driver)
PostgreSQL	No (but translated by driver)
Sybase ASA	No (but translated by driver)

Most recent database servers now support the standard ANSI outer join specification:

```
SELECT * FROM master LEFT OUTER JOIN detail ON (master.mid =
detail.mid)
  WHERE master.cdate IS NOT NULL
```

You should use recent database servers and use ANSI outer joins only.

3.19 Sub-string expressions

Only INFORMIX supports sub-string specification with square brackets:

```
SELECT * FROM item WHERE item_code[1,4] = "XBFG"
```

However, most database servers support a function that extracts sub-strings from a character string:

Database Server Type	Supports col[x,y] sub-strings?	Provides sub-string function?
GeneroDB	Yes	`SUBSTR(expr,start,length)`
IBM DB2 UDB (Unix)	No	`SUBSTR(expr,start,length)`
Informix (1)	Yes	`SUBSTR(expr,start,length)`
Microsoft SQL Server (2)	No	`SUBSTRING(expr,start,length)`
MySQL	No	`SUBSTR(expr,start,length)`
Oracle Database Server	No	`SUBSTRING(expr,start,length)`
PostgreSQL	No	`SUBSTRING(expr FROM start FOR length )`
Sybase ASA	No	`SUBSTR(expr,start,length)`

Warning: INFORMIX allows you to update some parts of a [VAR]CHAR column by using the sub-string specification (`UPDATE tab SET col[1,2] ='ab'`). This is not possible with other databases.

Review the SQL statements using sub-string expressions and use the database specific sub-string function.

You could also create your own SUBSTRING() user function in all databases that do not support this function, to have a common way to extract sub-strings. In Microsoft **SQL Server**, when you create a user function, you must specify the owner as prefix when using the function. Therefore, you should create a SUBSTRING() user function instead of SUBSTR().

3.20 Using ROWIDs

Rowids are implicit primary keys generated by the database engine. Not all database servers support rowids:

Database Server Type	Rowid keyword?	Rowid type?
GeneroDB	`ROWID`	`INTEGER`
IBM DB2 UDB (Unix)	`none`	`none`
Informix (1)	`ROWID`	`INTEGER`
Microsoft SQL Server (2)	`none`	`none`
MySQL	`none`	`none`
Oracle Database Server	`ROWID`	`CHAR(18)`
PostgreSQL	`OID`	`internal type`
Sybase ASA	`none`	`none`

Warning: INFORMIX fills the SQLCA.SQLERRD[3] register with the ROWID of the last updated row. This register is an INTEGER and cannot be filled with rowids having CHAR(*) type.

Search for ROWID and SQLCA.SQLERRD[3] in your code and review the code to remove the usage of rowids.

3.21 MATCHES operator

The MATCHES operator allows you to scan a string expression:

```
SELECT * FROM customer WHERE customer_name MATCHES "A*[0-9]"
```

Here is a table listing the database servers which support the MATCHES operator:

Database Server Type	Support for MATCHES operator?
GeneroDB	Yes
IBM DB2 UDB (Unix)	No
Informix (1)	Yes
Microsoft SQL Server (2)	No
MySQL	No
Oracle Database Server	No
PostgreSQL	No
Sybase ASA	No

The MATCHES operator is specific to INFORMIX SQL and Genero db. There is an equivalent standard operator: LIKE. We recommend to replace MATCHES expressions in your SQL statements with a standard LIKE expression. MATCHES uses `*` and `?` as wildcards. The equivalent wildcards in the LIKE operator are `%` and `_`. Character ranges `[a-z]` are not supported by the LIKE operator.

Remark: The BDL language provides a MATCHES operator which is part of the runtime system. Do not confuse this with the SQL MATCHES operator, used in SQL statements. There is no problem in using the MATCHES operator of the BDL language.

Warning: A program variable can be used as parameter for the MATCHES or LIKE operator, but you must pay attention to blank padding semantics of the target database. If the program variable is defined as a CHAR(N), it is filled by the runtime system with trailing blanks, in order to have a size of N. For example, when a CHAR(10) variable is assigned with "ABC%", it contains actually "ABC%<6 blanks>". If this variable is used as LIKE parameter, the database server will search for column values matching "ABC" + some characters + 6 blanks. To avoid automatic blanks, use a VARCHAR(N) data type instead of CHAR(N).

3.22 GROUP BY clause

Some databases allow you to specify a column index in the GROUP BY clause:

```
SELECT a, b, sum(c) FROM table GROUP BY 1,2
```

This is not possible with all database servers:

Database Server Type	GROUP BY colindex, ... ?

GeneroDB	No
IBM DB2 UDB (Unix)	No
Informix (1)	Yes
Microsoft SQL Server (2)	No
MySQL	Yes
Oracle Database Server	No
PostgreSQL	Yes
Sybase ASA	No

Search for GROUP BY in your SQL statements and use explicit column names.

3.23 LENGTH() function

Not all database servers support the LENGTH() function, and some have specific behavior:

Database Server Type	Length function?	Counts trailing blanks for CHAR() columns?	Return value when NULL
GeneroDB	LENGTH(expr)	No	NULL
IBM DB2 UDB (Unix)	LENGTH(expr)	Yes	NULL
Informix (1)	LENGTH(expr)	No	NULL
Microsoft SQL Server (2)	LEN(expr)	No	NULL
MySQL	LENGTH(expr)	No	NULL
Oracle Database Server	LENGTH(expr)	Yes	NULL
PostgreSQL	LENGTH(expr)	Yes	NULL
Sybase ASA	LENGTH(expr)	No	NULL

Search for LENGTH in your SQL statements and review the code of the database-specific function. You could also define your own LEN() user function to have a common function in all databases. In Microsoft SQL Server, when you create a user function, you must specify the owner as prefix when using the function. Therefore, you should create a LEN() user function instead of LENGTH().

Remark: The BDL language provides a LENGTH built-in function which is part of the runtime system. Do not confuse this with the SQL LENGTH() function, used in SQL statements. There is no problem in using the LENGTH() function of the BDL language. However, the LENGTH() function of the language returns zero when the string expression is NULL.

3.24 SQL Interruption

With Informix, it is possible to interrupt a long-running query if the SQL INTERRUPT ON option is set by the Genero program. The database server returns SQLCODE -213, which can be trapped to detect a user interruption.

```
01 MAIN
02   DEFINE n INTEGER
03   DEFER INTERRUPT
04   OPTIONS SQL INTERRUPT ON
05   DATABASE test1
06   WHENEVER ERROR CONTINUE
07   -- Start long query (self join takes time)
08   -- From now on, user can hit CTRL-C in TUI mode to stop the query
09   SELECT COUNT(*) INTO n FROM customers a, customers b
10        WHERE a.cust_id <> b.cust_id
11   IF SQLCA.SQLCODE == -213 THEN
12    DISPLAY "Statement was interrupted by user..."
13      EXIT PROGRAM 1
14   END IF
15   WHENEVER ERROR STOP
16   ...
17 END MAIN
```

When SQL Interruption is available for a database server type, Genero database drivers implement it to behave as in Informix, converting the native error to the code -213. Not all database servers support SQL interruption:

Database Server Type	SQL Interruption API	SQL error code for interrupted query
GeneroDB 3.80	SQLCancel()	Native error -30005
IBM DB2 UDB 9.x	SQLCancel()	Native error -952
Informix	sqlbreak()	Native error -213
Microsoft SQL Server 2005 (SNC driver only)	SQLCancel()	SQLSTATE HY008
MySQL	N/A	?
Oracle Database Server 8.x, 9.x, 10.x	OCIBreak()	Native error -1013
PostgreSQL 8.x	PQCancel()	SQLSTATE 57014
Sybase ASA	N/A	?

The Interaction Model

Summary:

- The Model-View-Controller Paradigm
- Controlling User Actions
  - Binding Action views to Action Controllers
  - Decorating Action Views
  - Enabling and Disabling Actions
  - Default Action Views
- Predefined Actions
  - Definition
  - Lists of predefined actions
  - Overwriting Predefined Actions in Interactive Instructions
- Keyboard Accelerator Names
  - Accelerator keys
  - List of key names
  - Accelerator key modifiers
- Interruption Handling
- Detecting data changes immediately
- Controlling data validation when an action is fired
- Windows closed by the user

See also: Dynamic User Interface, Form Files, Windows and Forms.

The Model-View-Controller Paradigm

The Dynamic User Interface architecture is based on the Model-View-Controller (MVC) paradigm. The *Model* defines the object to be displayed (typically the application data that is stored in program variables). The *View* defines the decoration of the Model (how the Model must be displayed to the screen, this is typically the form). The *Controller* is the program code that implements the processing that manages the Model. Multiple *Views* can be associated to a *Model* and a *Controller*.

With BDL, you define the Views in the Abstract User Interface tree or through built-in classes designed for this (such as Window or Form). You store Models in the program variables, and you implement the Controllers with interactive instructions, such as DIALOG or INPUT.

Normally the Controllers should not provide any decoration information, as that is the purpose of Views. Because of the history of the language, however, some interactive instructions such as MENU define both the Controller and some presentation information such as menu title, command labels, and comments. In this case, the runtime system automatically creates the View with that information; you can still associate other Views to the same controller.

Controlling User Actions

Binding Action Views to Action Controllers

In the user interface of the application, you can have interactive elements (such as buttons) that can trigger an event transmitted to the Runtime System for interpretation. To manage such events, the *Action Views* can produce *Actions Events* that will execute the code of the corresponding *Action Handler* in the current interactive instruction of the program.

Action Views (like buttons) are bound to an action by the 'name' attribute. For example, a Toolbar button with the name 'cancel' is bound to the Action Handler using the name 'cancel'. Action Handlers are defined in interactive instructions with `ON ACTION` clause or `COMMAND` / `ON KEY` clauses:

```
01 INPUT ARRAY custarr WITHOUT DEFAULTS FROM sr_cust.*
02   ON ACTION printrec
03     CALL PrintRecord(custarr[arr_curr()].*)
04   ON ACTION showhelp
05     CALL ShowHelp()
06 END INPUT
```

The name of the action must be a valid identifier in the program.

Warning: In the Abstract User Interface tree (where the action views are defined), action names are case-sensitive (as they are standard DOM attribute values). In BDL, however, identifiers are not case-sensitive. To avoid any confusion, the compiler automatically converts action identifiers to lowercase.

For singular dialogs such as INPUT or DISPLAY ARRAY, actions only have a simple name. Action names in a DIALOG instruction, however, have both simple names and prefixed names. When using a DIALOG instruction, one can define the same action (or have predefined actions such as *insert*, *append*, *delete*) in different sub-dialogs. These action names automatically get a prefix to identify actions properly across sub-dialogs. These actions are prefixed with the sub-dialog identifier. For example, when defining an `ON ACTION save` handler in a DISPLAY ARRAY sub-dialog using a screen-array named "sa", the runtime system will identify that action with both the "save" name and the "**sa.save**" prefixed name. In the form files, you can then bind Action Views to sub-dialog Action Handlers by using the sub-dialog prefix.

For backward compatibility, the `COMMAND` / `ON KEY` clauses are still supported. However, it is strongly recommended that you use `ON ACTION` clauses instead, as the `ON ACTION` clauses identify user actions with an abstract name.

Decorating Action Views

A default decoration for action views can be centralized in an external file. This is strongly recommended, to separate the decoration of the action view from action usage (the action handler) as much as possible. See Action Defaults for more details.

Enabling and Disabling Actions

During an interactive instruction, you can enable or disable an action with the `setActionActive()` method of the ui.Dialog built-in class. This method takes the name of the action (in lowercase letters) and an boolean expression (`0` or `FALSE`, `1` or `TRUE`) as arguments.

```
01 ...
02    BEFORE INPUT
03       CALL DIALOG.setActionActive("zoom",FALSE)
04 ...
```

Default Views for Actions

If no explicit action view is defined, such as a toolbar button or a topmenu command, the front end creates a default action view for each `MENU COMMAND`, `ON KEY` or `ON ACTION` clause used in the current interactive instruction (typically, the default action views appear as buttons in the action frame). You can control the presentation of these default views with Window Styles.

If one or more action views are defined explicitly for a given action, the front end considers that the default view is no longer needed, and hides the corresponding button. Typically, if you define in the form a BUTTONEDIT field or a BUTTON that triggers an action, you do not need an additional button in the action frame.

Predefined Actions

Definition

The BDL language predefines some action names for common operations of interactive instructions. Predefined actions can be:

- Automatic actions: actions that are automatically created and handled by the runtime system.
- Special actions: actions with a special usage.
- Local actions: actions that are handled on the front end only.

If you define your own `ON ACTION` handler with the name of an *Automatic* action, the default action processing is bypassed and the program code is executed instead. For more details, see Overwriting Automatic Actions in Interactive Instructions.

As for user-defined actions, if you design action views using predefined action names, they will automatically attach themselves to the actions of the interactive instructions. It is also possible to define default images, texts, comments and accelerator keys in the Action Defaults resource file for these predefined actions.

List of predefined actions

Action Name	Description	ON ACTION block	Context
Runtime Controlled	**Automatically created by the runtime system (except for DIALOG)**		
`accept`	Validates the current interactive instruction	possible	(1)
`cancel`	Cancels the current interactive instruction	possible	(1)
`close`	Triggers a cancel key in the current interactive instruction (by default)	possible	(7)
`insert`	Inserts a new row before current row	possible	(2)
`append`	Appends a new row at the end of the list	possible	(2)
`delete`	Deletes the current row	possible	(2)
`nextrow`	Moves to the next row in a list	possible	(4)
`prevrow`	Moves to the previous row in a list	possible	(4)
`firstrow`	Moves to the first row in a list	possible	(4)
`lastrow`	Moves to the last row in a list	possible	(4)
`help`	Shows the help topic defined by the `HELP` clause	possible	(1)
Special	**Special behavior**		
`interrupt`	Sends an interruption request to the program when processing	no	(5)
`dialogtouched`	Sent by the front end each time the user modifies the value of a field	yes	(7)
Front-end controlled	**Handled by the front end (the runtime does not know about these actions)**		
`editcopy`	Copies to the clipboard the current selected text	no	(1)
`editcut`	Copies to the clipboard and removes the current selected text	no	(1)
`editpaste`	Pastes the clipboard content to the current input widget	no	(1)
`nextfield`	Moves to the next field in the form	no	(3)
`prevfield`	Moves to the previous field in the form	no	(3)
`nextrow`	Moves to the next row in the list (supported for backward compatibility - is a runtime action)	no	(4)
`prevrow`	Moves to the previous row in the list (supported for backward compatibility - is a runtime action)	no	(4)
`firstrow`	Moves to the first row in the list (supported for backward compatibility - is a runtime action)	no	(4)

lastrow	Moves to the last row in the list (supported for backward compatibility - is a runtime action)	no	(4)
nextpage	Moves to the next page in the list	no	(4)
prevpage	Moves to the previous page in the list	no	(4)
nexttab	Moves to the next page in the folder	no	(6)
prevtab	Moves to the previous page in the folder	no	(6)

1. CONSTRUCT, INPUT, PROMPT, INPUT ARRAY and DISPLAY ARRAY.
2. INPUT ARRAY only.
3. CONSTRUCT, INPUT and INPUT ARRAY.
4. INPUT ARRAY and DISPLAY ARRAY.
5. Only possible when no interactive instruction is active.
6. Possible in any kind of interactive instruction (MENU included).
7. DIALOG, CONSTRUCT, INPUT, PROMPT, INPUT ARRAY and DISPLAY ARRAY.

Overwriting Automatic Actions in Interactive Instructions

The `ON ACTION` clause can be used with a predefined action name in an interactive instruction :

```
01 INPUT BY NAME customer.*
02  ON ACTION accept
03     ...
04 END INPUT
```

In this case, the default behavior is not performed; the user code is executed instead.

Keyboard Accelerator Names

Accelerator keys

Some parts of the user interface can define accelerators keys. With Action Defaults, you can define up to four accelerator keys for the same action, by setting the *acceleratorName*, *acceleratorName2*, *acceleratorName3* and *acceleratorName4* attributes.

If no accelerators are defined in the Action Defaults, the runtime system sets default accelerators for predefined actions, according to the user interface mode. For example, the *accept* action will get the `Return` and `Enter` keys in GUI mode; the `Escape` key would be used in TUI mode.

If you want to force an action to have no accelerator, specify "`none`" as the accelerator name.

If one of the user-defined actions uses an accelerator that would normally be used for a predefined action, the runtime system does not set that accelerator for the predefined action. For example (in GUI mode), if you define an `ON ACTION quit` with an action default using the accelerator "`Escape`", the *cancel* predefined action will not get the "`Escape`" default accelerator. User settings take precedence over defaults.

Note that text edition and navigation accelerators such as `Home` and `End` are usually local to the widget. According to the context, such accelerators might be eaten by the widget and will not fire the action bound to the corresponding accelerator defined in the Action Defaults. For example, even if the Action Defaults for *firstrow* action defines the `Home` accelerator, when using an INPUT ARRAY, the Home keystroke will jump to the beginning of the edit field, not the the first row of the list.

The following table lists all the keyboard accelerator names recognized by the runtime system:

Accelerator Name	Description
`none`	Special name indicating the the runtime system must not set any default accelerator for the action.
`0-9`	Decimal digits from 0 to 9
`A-Z`	Letters from A to Z
`F1-F35`	The functions keys
`Return`	The RETURN key (alphanumeric keypad, see Note)
`Enter`	The ENTER key (numeric keypad, see Note)
`Space`	The SPACE-BAR key
`Escape`	The ESCAPE key
`Tab`	The TABULATION Key
`BackSpace`	The BACKSPACE key (do not confuse with DELETE key)
`Up`	The UP key (arrow keyboard group)
`Down`	The DOWN key (arrow keyboard group)
`Left`	The LEFT key (arrow keyboard group)
`Right`	The RIGHT key (arrow keyboard group)
`Insert`	The INSERT key (navigation keyboard group)
`Delete`	The DELETE key (navigation keyboard group)
`Home`	The HOME key (navigation keyboard group)
`End`	The END key (navigation keyboard group)
`Next`	The NEXT PAGE key (navigation keyboard group)
`Prior`	The PRIOR PAGE key (navigation keyboard group)

Note:

The "`Enter`" key represents the ENTER key available on the numeric keypad of standard keyboards, while "`Return`" represents the RETURN key of the alphanumeric keyboard. By default, the validation action is configured to accept both "`Enter`" and "`Return`" keys. See the Action Defaults file.

Accelerator key modifiers

All of the key names listed in the previous table can be combined with CONTROL / SHIFT / ALT modifiers, by adding "`Control-`", "`Shift-`", or "`Alt-`" to the name of the accelerator.

For example:

```
Control-P
Shift-Alt-F12
Control-Shift-Alt-Z
```

Interruption Handling

Why do we need interruption handling?

When the BDL program executes an interactive instruction, the front end can send action events based on user actions. When the program performs a long process like a loop, a report, or a database query, the front end has no control. You might want to permit the user to stop a long-running process.

How to program the detection of user interruptions

To detect user interruptions, you define an action view with the name 'interrupt'. When the runtime system takes control to process program code, the front end automatically enables the local 'interrupt' action to let the user send an asynchronous interruption request to the program.

Warning: The front end can not handle interruption requests properly if the display generates a lot of network traffic. In this case, the front end has to process a lot of user interface modifications and has no time to detect a mouse click on the 'interrupt' action view. A typical example is a program doing a loop from 1 to 10000, just displaying the value of the counter to a field and doing a refresh. This would generate hundreds of AUI tree modifications in a short period of time. In such a case, we recommended that you calculate a modulo and display steps 10 by 10 or 100 by 100.

Interruption handling example:

Form file "f1.per":

```
01 LAYOUT
02  GRID
03  {
04   Step: [pb                    ]
05         [sb                    ]
06  }
07  END
08 END
09 ATTRIBUTES
10 PROGRESSBAR pb = FORMONLY.progress, VALUEMIN=0, VALUEMAX=100;
11 BUTTON sb : interrupt, TEXT="Stop";
12 END
```

Program:

```
01 MAIN
02  DEFINE i,j INTEGER
03  DEFER INTERRUPT
04  OPEN FORM f1 FROM "f1"
05  DISPLAY FORM f1
06  LET int_flag=FALSE
07  FOR i=1 TO 100
08    DISPLAY i TO progress
09    CALL ui.Interface.refresh()
10    FOR j=1 TO 1000 -- Loop to emulate processing
11      DISPLAY j
12      IF int_flag THEN EXIT FOR END IF
13    END FOR
14    IF int_flag THEN EXIT FOR END IF
15  END FOR
16 END MAIN
```

Detecting data changes immediately

You can use a special predefined action to detect user changes immediately and execute code in the program to set up your interactive instruction. This special action has the name "*dialogtouched*" and must be declared with an `ON ACTION` clause to be enabled:

```
01  DIALOG
02    ...
03    ON ACTION dialogtouched
04       LET changing = TRUE
05       CALL DIALOG.setActionActive("dialogtouched", FALSE)
06    ...
07  END DIALOG
```

Each time the user modifies the value of a field (without leaving the field), the `ON ACTION dialogtouched` block will be executed; This can be triggered by typing characters in a text editor field, clicking a checkbox / radiogroup, or modifying a slider.

The *dialogtouched* action works for any field controlled by the current interactive instruction, and with any type of form field.

Note that the current field may contain some text that does not represent a value of the underlying field data type. For this reason, the target variable cannot hold the current text displayed on the screen when the `ON ACTION dialogtouched` code is executed, even when using the `UNBUFFERED` mode.

You may only want to detect the beginning of record modification, to enable a "save" action for example. To prevent further *dialogtouched* action events, just disable that action with a setActionActive() call. If this action is enabled, the `ON ACTION` block will be fired each time the user modifies the value in the current field.

To avoid data validation, the *dialogtouched* action is defined with `validate="no"` in the default Action Defaults file. This is mandatory when using the `UNBUFFERED` mode; otherwise the runtime would try to copy the input buffer into the program variable when a *dialogtouched* action is fired.

Controlling data validation when an action is fired

When using the `UNBUFFERED` mode of interactive instructions such as INPUT or DIALOG, if the user triggers an action, the current field data is checked and loaded in the target variable bound to the form field. For example, if the user types a wrong date (or only a part of a date) in a field using a DATE variable and then clicks on a button to fire an action, the runtime system will throw an error and will not execute the `ON ACTION` block corresponding to the button.

If you want to prevent data validation for some actions, you can use the `validate` Action Default attribute. This attribute instructs the runtime not to copy the input buffer text into the program variable (requiring input buffer text to match the target data type).

The `validate` Action Default attribute can be set in the global action default file, or at the form level, in a line of the ACTION DEFAULTS section.

Windows closed by the user

In graphical applications, windows can be closed by the user, for example by pressing `ALT+F4` or by clicking the cross button in the upper-left corner of the window. A Predefined Action is dedicated to this specific event. When the user closes a graphical window, the program gets a *close* action.

By default, when the program is in a MENU instruction, the *close* action is converted to an INTERRUPT keypress (the key that cancels an interactive instruction). This means that a `COMMAND KEY(INTERRUPT)` block is invoked if it is defined in the `MENU` statement. If there is no `COMMAND KEY(INTERRUPT)`, nothing happens.

When executing an INPUT, INPUT ARRAY, CONSTRUCT or DISPLAY ARRAY, the *close* action acts by default the same as the *cancel* predefined action. So when the user clicks the X cross button, the interactive instruction stops and int_flag is set to 1. Note that if the `CANCEL=FALSE` option is set, no *close* action will be created, and you must write an `ON ACTION close` control block to create an explicit action. In this case, you define an action handler for *close* and the int_flag register will not be set.

When executing a DIALOG instruction, the *close* action executes the `ON ACTION close` block if defined. Otherwise, the *close* action is mapped to the *cancel* action and the `ON ACTION cancel` block is fired is one is defined. If neither `ON ACTION close`, nor `ON ACTION cancel` is defined, the window cannot be closed with the X cross button or an `ALT+F4` keystroke.

You typically implement a *close* action handler as in the following example:

```
01 INPUT BY NAME cust_rec.*
02    ...
03    ON ACTION close
04      IF msg_box_yn("Are you sure you want to close this window?") ==
"y" THEN
05        EXIT INPUT
06      END IF
07    ...
08 END INPUT
```

Using Windows and Forms

Summary:

- Windows and Forms Concepts
- Opening a Window (`OPEN WINDOW`)
  - Window Dimensions
  - Open With Form
  - Window Styles
  - Window Title
  - OPEN WINDOW attributes
- Closing a Window (`CLOSE WINDOW`)
- Selecting a Window (`CURRENT WINDOW IS`)
- Opening a Form (`OPEN FORM`)
- Displaying a Form (`DISPLAY FORM`)
- Closing a Form (`CLOSE FORM`)
- Clearing a Window (`CLEAR WINDOW`) **TUI Only!**
- Clearing the screen (`CLEAR SCREEN`) **TUI Only!**
- Displaying text by position (`DISPLAY AT`) **TUI Only!**

See also: Presentation Styles, Window class, Form class, Flow Control, Forms, Input Array, Display Array, Record Input, Construct.

Windows and Forms Concepts

Programs manipulate "Window" and "Form" objects to define display areas for interactive instructions like INPUT ARRAY, DISPLAY ARRAY, DIALOG, INPUT and CONSTRUCT. When an interactive instruction takes control, it uses the Form associated with the current window.

Windows

Windows are created from programs; they define a display context for sub-elements like forms, ring menus, message and error lines. A window can contain only one form at a time.

When using a character terminal, windows are displayed as fixed-size boxes, at a given line and column position, with a given width and height. When using a graphical front end, windows are displayed as independent resizable windows by default. This behavior is needed to create real graphical applications, but it breaks the old-mode layout implementations.

When the runtime system starts a program, it creates a default window named SCREEN. This default window can be used as another window (it can hold a Ring Menu and a Form), but it can also be closed, with `CLOSE WINDOW SCREEN`.

A program creates a window with the OPEN WINDOW instruction, which also defines the window identifier. The program destroys a Window with the CLOSE WINDOW instruction. One or more windows can be displayed concurrently, but there can be only one current Window. You can use the CURRENT WINDOW instruction to make a specific window current.

When opening a window, the window style is used to specify the type and the decoration of the window.

You can also use the ui.Window class to manipulate windows as objects.

Forms

Forms define the layout and presentation of areas used by the program to display or input data. Typically, Forms are loaded by programs from external files with the **42f** extension, the compiled version of Form Specification Files.

Forms files are identified by the file name, but you can also specify a form version with the VERSION attribute. The form version attribute is typically used to indicate that the form content has changed. The front-end is then able to distinguish different form versions and avoid saved settings being reloaded for new form versions.

A program can load a Form file with the OPEN FORM instruction, then display the Form with DISPLAY FORM into the current window, and release resources with CLOSE FORM. For temporary popup windows (typical record list where the user can select a row), you must dedicate a new window for the form. This can be done wit a unique instruction: OPEN WINDOW WITH FORM.

When a Form is displayed, it is attached to the current window and a ui.Form object is created internally. You can get this object with the ui.Window.getForm() method. The ui.Form built-in class is provided to handle form elements. You can, for example, hide some parts of a form.

The Form that is used by interactive instructions like INPUT is defined by the current window.

Windows in MDI mode

Windows can be displayed in an MDI container application; see Dynamic User Interface for more details.

OPEN WINDOW

Purpose:

Creates and displays a new Window.

Syntax:

```
OPEN WINDOW identifier
  [ AT line, column ]
  WITH [ FORM form-file | height ROWS, width COLUMNS ]
  [ ATTRIBUTE ( window-attributes ) ]
```

Notes:

1. *identifier* is the name of the window. It is always converted to lowercase by the compiler.
2. The `AT` clause is optional.
3. *line* is the integer defining the top position of the window.
   The first line in the screen is 1, while the relative line number inside the window is zero.
4. *column* is the integer defining the position of the left margin.
   The first column in the screen is 1, while the relative column number inside the window is zero.
5. *form-file* is a string literal or variable defining the form specification file to be used, without the file extension.
6. *height* defines the number of lines of the window in character units; includes the borders in character mode.
7. *width* defines the number of lines of the window in character units; includes the borders in character mode.
8. *window-attributes* defines the window attributes. See below for more details.

Tips:

1. For graphical applications, use this instruction without the `AT` clause, and with the `WITH FORM` clause.

Warnings:

1. The compiler converts the window identifier to lowercase for internal storage. When using functions or methods receiving the window identifier as a string parameter, the window name is case-sensitive. We recommend that you always specify the window identifier in lowercase letters.

Usage:

An `OPEN WINDOW` statement can have the following effects:

- Declares a name (the *identifier*) for the window.
- Indicates which form has to be used in that window.
- Specifies the display attributes of the window.
- When using character mode, specifies the position and dimensions of the window, in character units.

The window identifier must follow the rules for identifiers and be unique among all windows defined in the program. Its scope is the entire program. You can use this

identifier to reference the same Window in other modules with other statements (for example, CURRENT WINDOW and CLOSE WINDOW).

The runtime system maintains a stack of all open windows. If you execute `OPEN WINDOW` to open a new window, it is added to the window stack and becomes the current window. Other statements that can modify the window stack are CURRENT WINDOW and CLOSE WINDOW.

Window Dimensions

When using GUI mode, the `WITH lines ROWS, characters COLUMNS` clause is ignored, because the size of the window is automatically calculated according to its contents.

When using character mode, the `WITH lines ROWS, characters COLUMNS` clause specifies explicit vertical and horizontal dimensions for the window. The expression at the left of the `ROWS` keyword specifies the height of the window, in character unit lines. This must be an integer between 1 and max, where max is the maximum number of lines that the screen can display. The integer expression after the comma at the left of the `COLUMNS` keyword specifies the width of the window, in character unit columns. This must return a whole number between 1 and length, where length is the number of characters that your monitor can display on one line. In addition to the lines needed for a form, allow room for the Comment line, the Menu line, the Menu comment line and the Error line. The runtime system issues a runtime error if the window does not include sufficient lines to display both the form and these additional reserved lines. The minimum number of lines required to display a form in a window is the number of lines in the form, plus an additional line below the form for prompts, messages, and comments.

Open With Form

As an alternative to specifying explicit dimensions, the `WITH FORM` clause can specify a quoted string or a character variable that specifies the name of a file that contains the compiled screen form. The runtime system expects the compiled version of the form, but the file name should not include the **.42f** file extension. A window is automatically opened and sized to the screen layout of the form. When using character mode, the width of the window is from the left-most character on the screen form (including leading blank spaces) to the right-most character on the screen form (truncating trailing blank spaces). The length of the window is calculated as (form line) + (form length).

It is recommended that you use the `WITH FORM` clause, especially in the standard GUI mode, because the window is created in accordance with the form. If you use this clause, you do not need the OPEN FORM, DISPLAY FORM, or CLOSE FORM statement to open and close the form. The CLOSE WINDOW statement closes the window and the form.

Window Styles

By default windows are displayed as normal application windows, but you can use the window style to show a window at the top of all other windows, as a "modal window".

The window style defines the type of the window (normal, modal) and its decoration, via a Presentation Style. The Presentation Style specifies a set of attributes in an external file (.4st).

The `STYLE` attribute can be used in the OPEN WINDOW instruction to define the default style for a Window, but it is better to specify the window style in the form file, with the `WINDOWSTYLE` attribute of the LAYOUT section. This avoids decoration-specific code in the programs.

Warnings:

1. If you open and display a second form in an existing window, the window style of the second form is not applied.

The following standard window styles are defined in the default presentation style file (FGLDIR/lib/default.4st):

STYLE attribute	Style name in 4st file	Description
none	`Window`	Defines presentation attributes for common application windows. When using MDI containers, normal windows are displayed as MDI children.
`main`	`Window.main` `Window.main2`	Defines presentation attributes for starter applications, where the main window shows a startmenu if one is defined by the application.

`dialog`	`Window.dialog`
	`Window.dialog2`
	`Window.dialog3`
	`Window.dialog4`
		Defines presentation attributes for typical OK/Cancel modal windows.

`naked`	`Window.naked`	Defines presentation attributes for windows that should not show the default view for ring menus and action buttons (OK/Cancel).
`viewer`	`Window.viewer`	Defines presentation attributes for viewers as the report pager (fglreport.per).

Warnings:

1. It is recommended that you NOT change the default settings of windows styles in the **default.4st** file.
2. If you create your own style file, copy the default styles into your file.
3. It is not possible to change the presentation attributes of windows in the AUI tree.

For more details about the attributes you can set for Windows, see Presentation Styles.

Window Title

The TEXT attribute in the ATTRIBUTE clause defines the default title of the window. If the window is opened with a form (WITH FORM) that defines a TEXT attribute in the LAYOUT section, the default is ignored. Subsequent OPEN FORM instructions may change the window title if the form defines a new title in the LAYOUT section.

It is recommended that you specify the window title in the form file, not with the TEXT attribute of the OPEN WINDOW instruction.

If you want to set a window title dynamically, you can use the Window built-in class.

OPEN WINDOW attributes

The following table shows the *window-attributes* supported by the OPEN WINDOW statement:

Attribute	Description
TEXT = *string*	Defines the default title of the window. When a form is displayed, the form title (LAYOUT(TEXT="mytitle")) will be used as window title. ***We recommend that you define the window title in the form file!***
STYLE = *string*	Defines the default style of the window. If the form defines a window style, (LAYOUT(WINDOWSTYLE="mystyle")), it overwrites the default window style. See Window Styles for more details. ***We recommend that you define the window style in the form file!***
PROMPT LINE *integer* **TUI Only!**	In character mode, indicates the position of the prompt line for this window. The position can be specified with FIRST and LAST predefined line positions.
FORM LINE *integer* **TUI Only!**	In character mode, indicates the position of the form line for this window. The position can be specified with FIRST and LAST predefined line positions.
MENU LINE *integer* **TUI Only!**	In character mode, indicates the position of the ring menu line for this window. The position can be specified with FIRST and LAST predefined line positions.
MESSAGE LINE *integer* **TUI Only!**	In character mode, indicates the position of the message line for this window. The position can be specified with FIRST and LAST predefined line positions.

`ERROR LINE integer` **TUI Only!**	In character mode, indicates the position of the error line for this window. The position can be specified with `FIRST` and `LAST` predefined line positions.
`COMMENT LINE {OFF|integer}` **TUI Only!**	In character mode, indicates the position of the comment line or no comment line at all, for this window. The position can be specified with `FIRST` and `LAST` predefined line positions.
`BORDER` **TUI Only!**	Indicates if the window must be created with a border in character mode. A border frame is drawn outside the specified window. This means, that the window needs 2 additional lines and columns on the screen.
`BLACK, BLUE, CYAN, GREEN, MAGENTA, RED, WHITE, YELLOW` **TUI Only!**	Default color of the data displayed in the window.
`BOLD, DIM, INVISIBLE, NORMAL` **TUI Only!**	Default font attribute of the data displayed in the window.
`REVERSE, BLINK, UNDERLINE` **TUI Only!**	Default video attribute of the data displayed in the window.

The following list describes the default line positions in character mode:

- **First line:** Prompt line (output from PROMPT statement) and Menu line (command value from MENU statement).
- **Second line:** Message line (output from MESSAGE statement; also the descriptions of MENU options).
- **Third line:** Form line (output from DISPLAY FORM statement).
- **Last line:** Error line (output from ERROR statement); also comment line in any window except SCREEN.

Example:

```
01 MAIN
02    OPEN WINDOW w1 WITH FORM "customer"
03    MENU "Test"
04       COMMAND KEY(INTERRUPT) "exit" EXIT MENU
05    END MENU
06    CLOSE WINDOW w1
07 END MAIN
```

CLOSE WINDOW

Purpose:

Closes and destroys a window.

Syntax:

```
CLOSE WINDOW identifier
```

Notes:

1. *identifier* is the name of the window.
2. If the `OPEN WINDOW` statement includes the `WITH FORM` clause, it closes both the form and the window.
3. Closing a window has no effect on variables that were set while the window was open.
4. Closing the current window makes the next window on the stack the new current window. If you close any other window, the runtime system deletes it from the stack, leaving the current window unchanged.

Tips:

1. You can close the default screen window with the `CLOSE WINDOW SCREEN` instruction.

Warnings:

1. If the window is currently being used for input, `CLOSE WINDOW` generates a runtime error.

Example:

```
01 MAIN
02    OPEN WINDOW w1 WITH FORM "customer"
03    MENU "Test"
04       COMMAND KEY(INTERRUPT) "exit" EXIT MENU
05    END MENU
06    CLOSE WINDOW w1
07 END MAIN
```

CURRENT WINDOW

Purpose:

Makes a specified window the current window.

Syntax:

```
CURRENT WINDOW IS identifier
```

Notes:

1. *identifier* is the name of the window or the `SCREEN` keyword.

Usage:

Programs with multiple windows might need to switch to a different open window so that input and output occur in the appropriate window. To make a window the current window, use the `CURRENT WINDOW` statement.

When a program starts, the screen is the current window. Its name is SCREEN. To make this the current window, specify the keyword `SCREEN` instead of a window identifier.

If the window contains a form, that form becomes the current form when a CURRENT WINDOW statement specifies the name of that window. The CONSTRUCT, DISPLAY ARRAY, INPUT, INPUT ARRAY, and MENU statements use only the current window for input and output. If the user displays another form (for example, through an ON KEY clause) in one of these statements, the window containing the new form becomes the current window. When the CONSTRUCT, DISPLAY ARRAY, INPUT, INPUT ARRAY, or MENU statement resumes, its original window becomes the current window.

Example:

```
01 MAIN
02    OPEN WINDOW w1 WITH FORM "customer"
03    OPEN WINDOW w2 WITH FORM "custlist"
04    MENU "Test"
05       COMMAND "Win1"
06          CURRENT WINDOW IS w1
07       COMMAND "Win2"
08          CURRENT WINDOW IS w2
09       COMMAND KEY(INTERRUPT) "exit"
10          EXIT MENU
11    END MENU
12    CLOSE WINDOW w1
13    CLOSE WINDOW w2
14 END MAIN
```

CLEAR WINDOW TUI Only!

Purpose:

Clears the contents of a window in character mode.

Syntax:

```
CLEAR WINDOW identifier
```

Notes:

1. *identifier* is the name of the window, or the `SCREEN` keyword.

Warnings:

1. This instruction is provided for backward compatibility; it is only supported to clear windows created in character mode.

OPEN FORM

Purpose:

Declares a compiled form in the program.

Syntax:

```
OPEN FORM identifier FROM file-name
```

Notes:

1. *identifier* is the name of the window object.
2. The scope of reference of *identifier* is the entire program.
3. *file-name* is a string literal or variable defining the name of the compiled Form Specification File.
4. Form files are found by using the directory paths defined in the DBPATH environment variable.

Tips:

1. When the window is dedicated to the form, use the `OPEN WINDOW WITH FORM` instruction to create the window and the form object in one statement.
2. It is not recommended that you provide a path for *file-name*; You should use the DBPATH environment variable instead.

Usage:

In order to use a **42f** compiled version of a Form Specification File, the programs must declare the form with the `OPEN FORM` instruction and then display the form in the current window by using the DISPLAY FORM instruction. `OPEN FORM` / `DISPLAY FORM` are typically used at the beginning of programs to display the main form in the default `SCREEN` window:

```
01 OPEN FORM f FROM "customer"
02 DISPLAY FORM f
```

If you execute an `OPEN FORM` with the name of an open form, the runtime system first closes the existing form before opening the new form.

In character mode, the form is displayed in the current window at the position defined by the `FORM LINE` attribute that can be specified in the `ATTRIBUTE` clause of OPEN WINDOW or as default with the OPTIONS instruction.

After the form is loaded, you can activate the form by executing a CONSTRUCT, DISPLAY ARRAY, INPUT, INPUT ARRAY, or DIALOG statement. When the runtime system executes the `OPEN FORM` instruction, it allocates resources and loads the form into memory. To release the allocated resources when the form is no longer needed, the program must execute the CLOSE FORM instruction. This is a memory-management feature to recover memory from forms that the program no longer displays on the screen. If the form was loaded with a window by using the `WITH FORM` clause, it is automatically closed when the window is closed with a CLOSE WINDOW instruction.

The quoted string that follows the FROM keyword must specify the name of the file that contains the compiled screen form. This filename can include a pathname, but this is not recommended.

The form *identifier* does not need to match the name of the Form Specification File, but it must be unique among form names in the program. Its scope of reference is the entire program.

Example:

```
01 MAIN
02    OPEN FORM f1 FROM "customer"
03    DISPLAY FORM f1
04    CALL input_customer()
05    CLOSE FORM f1
06    OPEN FORM f2 FROM "custlist"
07    DISPLAY FORM f2
08    CALL input_custlist()
09    CLOSE FORM f2
10 END MAIN
```

DISPLAY FORM

Purpose:

Displays and associates a form with the current window.

Syntax:

```
DISPLAY FORM identifier
  [ ATTRIBUTE ( display-attributes ) ]
```

Notes:

1. *identifier* is the name of the form.
2. *window-attributes* defines the display attributes of the form. See below for more details.

Usage:

The following table shows the *display-attributes* supported by the `DISPLAY FORM` statement:

Attribute	Description
`BLACK, BLUE, CYAN, GREEN, MAGENTA, RED, WHITE, YELLOW`	Default color of the data displayed in the form.
`BOLD, DIM, INVISIBLE, NORMAL`	Default font attribute of the data displayed in the form. **Warning:** The `INVISIBLE` attribute is ignored.
`REVERSE, BLINK, UNDERLINE`	Default video attribute of the data displayed in the form.

The runtime system applies any other display attributes that you specify in the `ATTRIBUTE` clause to any fields that have not been assigned attributes by the ATTRIBUTES section of the Form Specification File, or by the database schema files, or by the OPTIONS statement. If the form is displayed in a window, color attributes from the `DISPLAY FORM` statement supersede any from the OPEN WINDOW statement. If subsequent CONSTRUCT, DISPLAY, or DISPLAY ARRAY statements that include an `ATTRIBUTE` clause reference the form, however, their attributes take precedence over those specified in the `DISPLAY FORM` instruction.

CLOSE FORM

Purpose:

Closes a form.

Syntax:

```
CLOSE FORM identifier
```

Notes:

1. *identifier* is the name of the form.
2. Releases the memory allocated to the form.

Tips:

1. A form associated with a window by the `OPEN WINDOW WITH FORM` instruction is automatically closed when the program closes the window with a CLOSE WINDOW instruction.

CLEAR SCREEN TUI Only!

Purpose:

Clears the complete application screen in character mode.

Syntax:

```
CLEAR SCREEN
```

Notes:

1. Clears the complete screen.

Warnings:

1. The `CLEAR SCREEN` instruction is provided for backward compatibility.

DISPLAY AT TUI Only!

Purpose:

Displays text at a given position in character mode in the current window.

Syntax:

```
DISPLAY text AT line, column [ ATTRIBUTE ( display-attributes ) ]
```

Notes:

1. *text* is any expression to be evaluated and displayed at the given position in the current window.
2. *line* is an integer literal or variable defining the line position in the current window.

3. *column* is an integer literal or variable defining the column position on the screen.
4. *display-attributes* defines the display attributes for the *text*. See below for more details.

Warnings:

1. The `DISPLAY AT` instruction is provided for backward compatibility and can only be used in character mode. To display data at a given place in a graphical form, use form fields and the DISPLAY TO instruction.

Usage:

The following table shows the *display-attributes* supported by the `DISPLAY AT` statement:

Attribute	Description
`BLACK, BLUE, CYAN, GREEN, MAGENTA, RED, WHITE, YELLOW` **TUI Only!**	The color of the displayed text.
`BOLD, DIM, INVISIBLE, NORMAL` **TUI Only!**	The font attribute of the displayed text.
`REVERSE, BLINK, UNDERLINE` **TUI Only!**	The video attribute of the displayed text.

Action Defaults

Summary:

- Basics
- Syntax
- Usage
  - Global Action Defaults
  - Form Action Defaults
  - Action Defaults and interactive instructions
  - Default accelerators for predefined actions
  - Controlling data validation when an action is fired
- Examples

See also: Dynamic User Interface, Predefined Actions, Action Defaults and MENU.

Basics

Action Defaults allow you to define default attributes for graphical objects (action views) associated with actions.

For example, you can specify the default text, comment, and image to use on graphical objects (buttons, toolbar items, and so on) bound to an action across all forms in the application.

You can centralize common action defaults in a global action defaults (.4ad) file. You can also define action defaults at the form level.

Syntax

Action defaults are defined in the .4ad file with the following syntax:

```
<ActionDefaultList>
  <ActionDefault name="action-name" [ attribute=value [...] ] />
  [...]
</ActionDefaultList>
```

Notes:

1. *action-name* identifies the action.
2. *attribute* is the name of an attribute.
3. *value* defines the value to be assigned to *attribute*.

Usage

Action defaults are provided to centralize the decoration attributes and accelerator keys for action views (graphical objects associated with actions). It is strongly recommended that you centralize these decoration attributes to avoid specifying them in all the source files that define the same action view.

Each attribute of an action view element bound to an action handler in the program will automatically be set to the value defined in the Action Defaults, if there is no value explicitly specified in the element definition for that attribute.

Note that in some situations, the action view can be bound to an action by specifying a sub-dialog prefix. For those views, the action defaults defined with the corresponding action name will be used to set the attributes with the default values. In other words, the prefix will be ignored. For example, if an action view is defined with the name "*custlist.append*", it will get the action defaults defined for the "*append*" action.

Warning: Action Defaults are applied only once, to newly created elements. Dynamic changes are not re-applied to action views. For example, if you first load a toolbar, then you load a global Action Defaults file, the attributes of the toolbar items will not be updated with the last loaded Action Defaults. If you dynamically create action views (like TopMenu or ToolBar), the action defaults are not applied, so you must set all decoration attributes by hand.

Action Defaults can be defined globally for the whole program or at the form level. Global Action Defaults are loaded from a default 4ad file or by using the ui.Interface.loadActionDefaults() method. Form Action Defaults can be specified in the form file or can be loaded with the ui.Form.loadActionDefaults() method.

For example, in most cases a *print* action needs a text decoration "Print", with a printer icon image and a CONTROL-P accelerator key. Those attributes can be centralized in the Action Defaults. Some action views of the *print* action may need specific attributes; for example, if the current form handles customer addresses, the comment attribute of a *print* button might be "Print current customer information". In this case you can define Action Defaults at the form level, which have a higher priority than the global Action Defaults. Additionally, if some action views must have a different image in the same form, you can specify the image attribute in the definition of each element to overwrite the defaults. For example, the toolbar button bound to the *print* action might have a small image, while the *print* button in the form might have a large one.

The final attribute values used for graphical elements are set based on the following priority:

1. Attribute defined in the action view element definition itself.
2. Attribute defined for the element action in the form Action Defaults.
3. Attribute defined for the element action in the global Action Defaults.

The following code defines a BUTTON in the form specification file::

```
01 ATTRIBUTES
02  BUTTON b1: print, TEXT="Do Print";
03 END
```

if the form Action Defaults define:

```
<ActionDefaultList>
  <ActionDefault name="print" image="smiley" comment="Print orders"
acceleratorName="control-p" />
</ActionDefaultList>
```

and the global Action Defaults define:

```
<ActionDefaultList>
  <ActionDefault name="print" text="Print" image="printer" />
</ActionDefaultList>
```

the button object will get the following final attribute values:

- `text = "Do Print"`
- `image = "smiley"`
- `comment = "Print orders"`

and the accelerator will be CONTROL-P.

Action Default attributes

The following attributes can be defined with Action Defaults:

Attribute	Description
`name = string`	This attribute identifies the action. See also predefined action names.
`text = string`	The default label to be displayed in action views (typically, the text of buttons).
`comment = string`	The default help text for this action (typically, displayed as bubble help).
`image = string`	The default image file to be displayed in the action view.
`acceleratorName = string`	The default accelerator key that can trigger the action, as defined in Accelerators.
`acceleratorName2 = string`	The second default accelerator key that can trigger the action, as defined in Accelerators.
`acceleratorName3 = string`	The third default accelerator key that can trigger the action, as defined in Accelerators.
`acceleratorName4 = string`	The fourth default accelerator key that can trigger the action, as defined in Accelerators.
`defaultView = string`	Indicates whether the front-end must show the

	default action view (buttons in control frame). Values can be: - `"yes"` the default action view is always visible. - `"no"` the default action view is never visible. - `"auto"` the default action view is visible if no other action view is explicitly defined. The default is `"auto"`.
`validate = string`	Defines the behavior of data validation when the action is fired. Values can be: - `"no"` no data validation is done (field text only available in input buffer). By default, data validation is driven by the dialog mode (`UNBUFFERED` or default mode).

Global Action Defaults

Global Action Defaults are defined in an XML file with the "`4ad`" extension. By default, the runtime system searches for a file named "`default.4ad`" in the current directory. If this file does not exist, it searches in the directories defined in the DBPATH environment variable. If no file was found using DBPATH, standard action default settings are loaded from the "`FGLDIR/lib/default.4ad`" file.

Warning: Global Action Defaults must be defined in a unique file; you cannot combine several "`4ad`" files.

You can override the default search by loading a specific Action Defaults file with the ui.Interface.loadActionDefaults() method. This method accepts a filename with or without the "`4ad`" extension. If you omit the file extension (recommended), the runtime system adds the extension automatically. If the file does not exist in the current directory, it is searched in the directories defined in the DBPATH environment variable.

Form Action Defaults

Action Defaults can be defined at the form level. When action defaults are defined in the form file, action views get the attributes defined locally.

You can define form action defaults with the ACTION DEFAULTS section in the form specification file. If you want to use common action defaults in several forms, you can use the preprocessor include directive to integrate an external file.

You can also load Form Action Defaults dynamically with the ui.Form.loadActionDefaults() method. This method accepts a filename with or without the "`4ad`" extension. If you omit the file extension (recommended), the runtime system adds the extension automatically. If the file does not exist in the current directory, it is searched in the directories defined in the DBPATH environment variable.

Action Defaults and interactive instructions

When using the `ON ACTION` clause in a dialog instruction, action defaults accelerators are applied in both GUI and TUI mode. For backward compatibility, this is not done in TUI mode when using the `ON KEY` clause.

The traditional `ON KEY` clause in a dialog like INPUT implicitly defines the *acceleratorName* attribute for the action, and the corresponding action default accelerator will be ignored. For example, when you define an `ON KEY(F10)` block, the first accelerator will be "F10", even if an action default defines an accelerator "F5" for the action "F10". However, you can set other accelerators with the *acceleratorName2, acceleratorName3* and *acceleratorName4* attributes in action defaults.

Warning: In TUI mode, actions created with ON KEY do not get accelerators of Action defaults; Only actions defined with ON ACTION will get accelerators of Action Defaults.

In menus, the behavior is a bit different, see the COMMAND and COMMAND KEY clause in MENU.

Default accelerators for predefined actions

If no accelerator is specified in action defaults for a Predefined Action, the runtime system sets one or more default accelerators according to the user interface mode. For example, the **accept** action will get the `Return` and `Enter` keys in GUI mode, but in TUI mode, the `Escape` key would be used.

If you want to force an action to have no accelerator, you can use "`none`" as accelerator name.

Controlling data validation when an action is fired

The `validate` attribute defines the behavior of the dialog for data validation when an action is fired. For more details, see Interaction Model.

Examples

Example 1: Loading a Action Defaults file:

Some Action Defaults in XML format:

```
01 <ActionDefaultList>
02    <ActionDefault name="print" text="Print" image="printer"
comment="Print report" />
03    <ActionDefault name="modify" text="Update" comment="Update the
record" />
```

```
04    <ActionDefault name="exit" text="Quit" image="byebye"
comment="Exit the program" validate="no" />
05 </ActionDefaultList>
```

The program loading the action defaults file:

```
01 MAIN
02   CALL ui.Interface.loadActionDefaults("mydefaults")
03 END MAIN
```

Example 2: Actions defaults in a form file:

```
01 ACTION DEFAULTS
02   ACTION accept ( COMMENT="Commit order record changes" )
03   ACTION cancel ( TEXT="Stop", IMAGE="stop", ACCELERATOR=SHIFT-F2,
VALIDATE=NO )
04   ACTION print ( COMMENT="Print order information",
ACCELERATOR=CONTROL-P, ACCELERATOR2=F5 )
05   ACTION zoom1 ( COMMENT="Open items list" )
06   ACTION zoom2 ( COMMENT="Open customers list" )
07 END
```

Presentation Styles

Summary:

- Introduction
- Syntax
- Usage
  - Defining a Style
  - Pseudo selectors
  - Using a specific style
  - Combining styles
  - Style attribute inheritance
  - Styles in the AUI tree
  - Loading Presentation Styles
  - Example of 4st file
- Element Types
- Colors
- Using Fonts
- Font Families
- Font Sizes
- Style Attributes Reference
  - Common Style Attributes
  - Window Style Attributes
  - MDI Container Style Attributes
  - Table Style Attributes
  - ComboBox Style Attributes
  - DateEdit Style Attributes
  - Label Style Attributes
  - ProgressBar Style Attributes
  - RadioGroup Style Attributes
  - TextEdit Style Attributes

See also: Dynamic User Interface.

Introduction

Presentation Styles allow you to define a set of decoration properties to be used in graphical objects. Presentation Styles are provided to centralize attributes related to the appearance of user interface elements.

Typical presentation attributes define font properties and foreground and background colors. Some presentation attributes will be specific to a given class of widgets (like the first day of week in a DATEEDIT).

Presentation Styles are defined in a resource file having an extension of **4st,** which must be distributed with other runtime files.

Syntax

```
<StyleList>
  <Style name="style-identifier" >
     <StyleAttribute name="attribute-name" value="attribute-value" />
     [...]
  </Style>
  [...]
</StyleList>
```

where *style-identifier* can be:

```
{ *
| element-type
| .style-name
| element-type.style-name }
```

Notes:

1. *element-type* defines the type of the graphical object (for example, `Window`).
2. *style-name* is the name of a specific style referenced by graphical objects using the `style` attribute.
3. *attribute-name* defines the name of the attribute.
4. *attribute-value* defines the value to be assigned to *attribute-name*.

Usage

Presentation Styles centralize the attributes related to the decoration of the elements of the user interface. Styles are applied implicitly by using global styles, or explicitly by naming a specific style in the `style` attribute of the element.

Defining a style

In the definition of a style, the 'name' attribute is used as a selector to apply style attributes to graphical elements.

You can define a style as global or specific:

- A style identified by a star (*) is a global style that is automatically applied to all elements:

  ```
  01 <Style name="*" >
  ```

- A style identified by an *element-type* is a global style that is automatically applied to all objects of this type:

```
01 <Style name="Window" >
02 <Style name="Edit" >
03 <Style name="ComboBox" >
```

- A style identified by a *style-name* is a specific style that can be applied to any element types using that style:

```
01 <Style name=".important" >
02 <Style name=".smallfont" >
```

- A style identified by an *element-type* followed by a dot and a *style-name* is a specific style that will only be applied to elements of the given type:

```
01 <Style name="Window.main" >
02 <Style name="Edit.mandatory" >
```

Priority: When different styles can be applied to an element, the following priority, from the most precise to the most generic, is used to determine the correct style :

1. `element-type.style-name`
2. `.style-name`
3. `element-type`
4. `*`

For instance, to find the style which will be applied to an `Edit` having the style attribute set to '`mandatory`', the following styles will be analyzed:

1. `Edit.mandatory`
2. `.mandatory`
3. `Edit`
4. `*`

Pseudo selectors

You can define a pseudo selector to make your style apply only when some conditions are fulfilled. You must precede it with a colon. You can also combine the pseudo selectors. If you do so, the style will be applied if all pseudo selector conditions are fulfilled.

```
01 <Style name="Table:even:input" >
02 <Style name="Edit:focus" >
03 <Style name="Edit.important:focus" >
```

Pseudo selectors have different priorities, and the style with the most important pseudo selector will be used when several styles match.

Priority	Pseudo selectors	Condition

1	`focus`	the widget has the focus
2	`query`	the widget is in construct mode
3	`display`	the widget is in a display array
4	`input`	the widget is in an input array, input or construct
5	`even`	this widget is on an even row if an array
6	`odd`	this widget is on an odd row if an array
7	`inactive`	the widget is inactive
8	`active`	the widget is active

Pseudo selectors also define the priority of your styles;: a more generic style will be used if the pseudo-selector has higher priority.

For instance: you want all important edits to have red text, but you want the current field to be displayed in blue:

```
01 <Style name="Edit.important" >
02 <Style name=":focus" >
```

Style ":focus" may be more generic than Edit.important; it will be used for the focused item, as the pseudo selector is more precise.

Using a specific style

To apply a specific style, set the *style-name* in the `style` attribute of the node representing the graphical element in the Abstract User Interface tree.

There are different ways to set the `style` attribute of a element:

- In the `ATTRIBUTES` clause of instructions such as OPEN WINDOW.
- As a form element attribute, in the Form Specification File.
- Dynamically by a program, using the DOM API or a built-in class method like setElementStyle().

For example, to define a style in a form file for a input field:

```
01 EDIT f001 = customer.fname, STYLE = "info";
```

Combining styles

You can combine several styles by using the space character as a separator in the `style` attribute:

```
01 EDIT f001 = customer.fname, STYLE = "info highlight mandatory";
```

When several styles are combined, the same presentation attribute might be defined by different styles. In this case, the first style listed that defines the attribute takes precedence over the other styles.

For example, if the `textColor` presentation attribute is defined as follows by the styles *info*, *highlight* and *mandatory*:

- *info* style does not define `textColor`.
- *highlight* style defines `textColor` as **blue**.
- *mandatory* style defines `textColor` as **red**.

Then the widgets having a style set to `"info highlight mandatory"` have `textColor` of **blue.**

Style Attribute Inheritance

A style attribute may be inherited by the descendants of a given node in the Abstract User Interface tree. For example, when using a style defining a `fontFamily` in a `GROUPBOX` container, you would expect that all the children in that groupbox would have the same font. However, some style attributes should not be inherited, such as `backgroundImage`.

Style inheritance is implicitly defined based on the attribute. The following sections contain tables with descriptions of style attributes, including the implicit inheritance for each attribute.

Presentation Styles in the Abstract User Interface tree

Presentation Styles are defined in the Abstract User Interface tree, under the `UserInterface` node, in a `StyleList` node following the syntax described above. The `StyleList` node holds a list of `Style` nodes that define a set of attribute values. Attribute values are defined in `StyleAttribute` nodes, with a `name` and a `value` attribute.

Loading Presentation Styles

Presentation Styles can be defined in an XML file that has the **4st** extension. By default, the runtime system searches for a file named "`default.4st`" in the current directory. If this file does not exist, it searches in the directories defined in the DBPATH environment variable. If the file was not found using DBPATH, standard Genero presentation styles are loaded from "`FGLDIR/lib/default.4st`" file.

You can overwrite the default search by loading a specific Presentation Style file with the ui.Interface.loadStyles() method. This method accepts an absolute path with the **4st** extension, or a simple file name without the **4st** extension. If you give a simple file name, for example "`mystyles`", the runtime system searches for the "`mystyles.4st`" file in the current directory. If the file does not exist, it searches in the directories defined in the DBPATH environment variable.

Warnings:

1. All styles must be defined in a unique file; you cannot combine several **4st** files.

Example of 4st file

```
<StyleList>
<Style name="*" >
 <StyleAttribute name="fontFamily" value="serif" />
</Style>
<Style name=".important" >
 <StyleAttribute name="textColor" value="#ff0000" />
</Style>
<Style name="Window" >
 <StyleAttribute name="toolBarPosition" value="top" />
 <StyleAttribute name="statusBarType" value="default" />
</Style>
<Style name="Window.dialog" >
 <StyleAttribute name="toolBarPosition" value="none" />
 <StyleAttribute name="statusBarType" value="node" />
</Style>
</StyleList>
```

Element Types

Styles may apply to any graphical elements of the user interface, such as the following:

- Button
- ButtonEdit
- CheckBox
- ComboBox
- DateEdit
- Edit
- Form
- Label
- RadioGroup
- Slider
- SpinEdit
- Table
- TimeEdit
- Window

The name of the element when used in a style file is case-sensitive (CheckBox, not checkbox).

Colors

This section describes how to specify a value for style attributes defining colors, such as `textColor`.

Syntax:

`{ generic-color | #rrggbb }`

Notes:

1. *generic-color* is any of the predefined colors supported by the language.

2. *#rrggbb* is a numerical color defined by a red/green/blue specification.

Usage:

The language defines a set of generic colors, interpreted by the front end according to the graphical capability of the workstation.

Generic color name	RGB Value	Color sample
white	#FFFFFF	
black	#000000	
darkGray	#A9A9A9	
gray	#808080	
lightGray	#D3D3D3	
darkBlue	#00008B	
blue	#0000FF	
lightBlue	#ADD8E6	
darkCyan	#008B8B	
cyan	#00FFFF	
lightCyan	#E0FFFF	
darkMagenta	#8B008B	
magenta	#FF00FF	
lightMagenta	#FFC0FF	
darkOlive	#505000	
olive	#808000	
lightOlive	#AAAA44	
darkGreen	#006400	
green	#00FF00	
lightGreen	#90EE90	
darkTeal	#005050	
teal	#008080	
lightTeal	#33CCCC	
darkRed	#8B0000	
red	#FF0000	
lightRed	#FF8080	
darkOrange	#FF8C00	
orange	#FFA500	
lightOrange	#FFCC00	
darkYellow	#AAAA00	
yellow	#FFFF00	
lightYellow	#FFFFE0	

You can also specify a generic system color:

Generic system color name	Meaning

`window`	Window background.
`windowText`	Text in Windows.
`buttonFace`	Face color for three-dimensional display elements.
`buttonText`	Text on PushButtons.
`highLight`	Item(s) selected in a control.
`highLightText`	Text of item(s) selected in a control
`infoBackground`	Background color for tooltip controls.
`infoText`	Text color for tooltip controls.
`grayText`	Grayed (disabled) text.
`appWorkSpace`	Background color of multiple document interface
`background`	Desktop background

You can also specify a color with the RGB notation, starting with a # dash character.

Each value of the RGB color specification must be provided in hexadecimal, in the range [00-FF].

Examples:

```
<StyleAttribute name="textColor" value="blue" />
<StyleAttribute name="textColor" value="#00FF45" />
```

Using Fonts

A desktop application should follow the current desktop settings. The front-end program (GDC, GJC, HTML browser) tries to determine the default font for the desktop, and also offers a global font chooser to let the end-user define which font best matches his expectations.

In most cases it is not possible to know what a potential end-user might expect regarding the font family. Therefore, the configuration should **avoid** using explicit font families and use only the `fontWeight`/`fontStyle`/`fontSize` properties. Only if the client can't determine a proper default font family for the desired platform should a known font family be added to the configuration.

Use abstract font sizes such as `medium`, `large`, `small`, or sizes relative to the user-chosen font (`em` units), rather than absolute point values. In an HTML browser you can choose two fonts (proportional/fixed), and a well-designed document should not use more than 2 fonts. This is also valid for applications.

Font Families

This section describes the possible values of the `fontFamily` style attribute.

Syntax:

`font-family` `[,...]`

Notes:

1. *font-family* defines a generic font family or a specific font family.
2. You can specify a comma-separated list of fonts.

Usage:

The language defines a set of generic font families, interpreted by the front end according to the graphical capability of the workstation:

Generic font family name	Real font family example	Text sample
`serif`	Times	**This is a nice font!**
`sans-serif`	Arial	**This is a nice font!**
`cursive`	Comic Sans Ms	**This is a nice font!**
`fantasy`	Algerian	**THIS IS A NICE FONT!**
`monospace`	Courier New	**This is a nice font!**

Any other name is interpreted as a specific font family, which identifies a local font supported by the front-end. Usually, it is one of the fonts installed on the workstation operating system. See front-end documentation for a list of supported local fonts.

Any font name containing white-spaces must be quoted, with single quotes.

You can specify a comma-separated list of font families.

Examples:

```
<StyleAttribute name="fontFamily" value="sans-serif" />
<StyleAttribute name="fontFamily" value="'Courier New'" />
<StyleAttribute name="fontFamily" value="'Times New Roman',Times,serif"
/>
```

Font Sizes

This section describes the possible values of the `fontSize` style attribute.

Syntax:

```
{ generic-size | nnpt | xxem }
```

Notes:

1. *generic-size* is one of the generic sizes such as '`small`' or '`xx-large`'.
2. *nn* defines an absolute size in number of points (1pt = 1/72 inches).
3. *xx* defines an relative size in the size units of the client (1em = as large as the font chosen in the client)

Usage:

The language defines a set of generic font sizes, interpreted by the front end according to the graphical capability of the workstation.

```
xx-small, x-small, small, medium, large, x-large, xx-large.
```

You can also specify an absolute font size, by giving a numeric value followed by the units (pt):

Examples:

```
<StyleAttribute name="fontSize" value="medium" />
<StyleAttribute name="fontSize" value="xx-large" />
<StyleAttribute name="fontSize" value="12pt" />
<StyleAttribute name="fontSize" value="1em" />
```

Style Attributes Reference

A Style attribute may be a common attribute that can be applied to any graphical element. Other Style attributes apply only to a specific graphical element (see below).

Common Style Attributes

The style attributes described in this section apply to any graphical elements, such as windows, layout containers, or form items.

Attribute	Inheritance	Description
`backgroundImage`	No	Defines an image file to be displayed in the background. Value can be a simple local image file name without the extension, or an URL. Default is no value (no background image).
`backgroundColor`	No	Defines the color to be used to fill the

		background of the object. For possible values, see Colors. Default is no value (default color of the object).
`fontFamily`	Yes	Defines the name of the font. For possible values, see Font Families. Default is no value (default object font or inherited font).
`fontSize`	Yes	Defines the size of the characters. For possible values, see Font Sizes. Default is no value (default object font or inherited font).
`fontStyle`	Yes	Defines the style of characters. Values can be `"normal", "italic"` or `"oblique"`. Default is no value (default object font or inherited font).
`fontWeight`	Yes	Defines the weight of the characters. Values can be `"light"`, `"normal"`, `"bold"`, `"demi-bold"` or `"black"`. Default is no value (default object font or inherited font).
`textColor`	Yes	Defines the color to be used to paint the text of the object. For possible values, see Colors. Default is no value (default object color or inherited color).
`textDecoration`	Yes	Defines the decoration for the text. Values can be `"overline"`, `"underline"` or `"line-through"`. Default is no value (default object font or inherited font).
`border`	No	Defines the border for the widget. If Value is `"none"`, it removes the border. Default is no value (the widget gets its default appearance). This attribute especially applies to Image, Edit, ButtonEdit, DateEdit, RadioGroup, Group, Button, Action, MenuAction, Menu, and Dialog.
`localAccelerators`	No	Defines how the widget must behaves regarding key strokes. If value is `"yes"` (default), the local accelerators have higher priority. Ex: "HOME" key moves the cursor to the first position. If value is `"no"`, the application

accelerators have higher priority. Ex: "HOME" key selects the first row of the current array.
The following keys are managed "locally" if attribute defined to `"yes"`.
TextEdits: `left, right, up, down, (control+)home, (control+)end, (control+)backspace, (control+)delete`
Edits Based widgets: `left, right, home, end, (control+)backspace, (control+)delete`

Warning: TTY attributes defined for the corresponding item have higher priority than the styles. For instance, if you define in your .per:

```
EDIT name: FORMONLY.name, COLOR=blue;
```

the EDIT will have blue text whatever you style defines.
Inherited TTY attributes (set on one of the parents) will nevertheless be overridden by styles.

Window Style Attributes

The following table shows the presentation attributes for Windows:

Attribute	Inheritance	Description
`windowType`	No	Defines the basic type of the window. Values can be `"normal"` or `"modal"`. Normal windows are displayed as typical application windows. Modal windows are displayed at the top of all other windows, typically used for temporary dialogs. Default is `"normal"`.
`windowState`	No	Defines the initial state of a window. Values can be `"normal"` or `"maximized"`. Default is `"normal"`.
`windowOptionClose`	No	Defines if the window can be closed with a system menu option or window header button. Values can be `"yes"`, `"no"` or `"auto"`. When value is `"auto"`, the option is enabled according to the window type. Default is `"auto"`.

		Warning: This attribute may have different behavior depending on the front end operating system. For example, when no system menu is used, it may not be possible to have this option enabled.
`windowOptionMinimize`	No	Defines if the window can be minimized with a system menu option or window header button. Values can be `"yes"`, `"no"` or `"auto"`. When value is `"auto"`, the option is enabled according to the window type. Default is `"auto"`. **Warning:** This attribute may have different behavior depending on the front end operating system. For example, when no system menu is used, it may not be possible to have this option enabled.
`windowOptionMaximize`	No	Defines if the window can be maximized with a system menu option or window header button. Values can be `"yes"`, `"no"` or `"auto"`. When value is `"auto"`, the option is enabled according to the window type. Default is `"auto"`. **Warning:** This attribute may have different behavior depending on the front end operating system. For example, when no system menu is used, it may not be possible to have this option enabled.
`windowSystemMenu`	No	Defines if the window shows a system menu. Values can be `"yes"`, `"no"` or `"auto"`. When value is `"auto"`, the system menu is enabled according to the window type. Default is `"auto"`.
`sizable`	No	Defines if the window can be resized by the user. Values can be `"yes"`, `"no"` or `"auto"`. When using `"auto"`, the window becomes resizable if the content of the first displayed form has resizable elements, for example when using a form with a TABLE container or an TEXTEDIT with

		STRETCH attribute. **Warning:** When using `"auto"`, the window becomes resizable based on the first form used in the window; the content of further forms is ignored. Default is `"yes"`.
`position`	No	Indicates the initial position of the window. Values can be `"default"`, `"field"`, `"previous"`, `"center"` or `"center2"`. When using `"default"`, the windows are displayed according to the window manager rules. When using `"field"` the window is displayed below the current field (works as `"default"` when current field does not exist). When using `"previous"` the window is displayed at the same position (top left corner) as the previous window. (works as `"default"` when there is no previous window). With `"center"`, the window is displayed in the center of the screen. With `"center2"`, the window is displayed in the center of the current window. Default is `"default"`. **Warning:** for front-ends using stored settings, `"field"`, `"previous"` and `"previous"` have higher priority than the settings.
`border`	No	Defines the border type of the window. Values can be `"normal"`, `"frame"`, `"tool"` or `"none"`. When using `"normal"`, the border is standard, with a normal window header with a caption. When using `"frame"`, only a frame appears, typically without a window header. When using `"tool"`, a small window header is used. When using `"none"`, the window has no border. Default is `"normal"`.
`forceDefaultSettings`	No	Indicates if the window content must be initialized with the saved positions

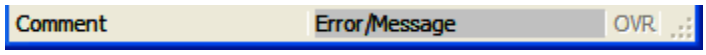
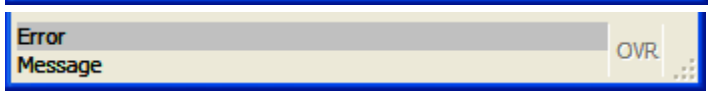
		and sizes. By default, windows are re-opened at the position and with the size they had when they were closed. You can force the use of the initial settings with this attribute. This applies also to column position and width in tables. Default is `"0"`.
`actionPanelPosition`	No	Defines the position of the action button frame (OK/Cancel). Values can be `"none"`, `"top"`, `"left"`, `"bottom"` or `"right"`. Default is `"right"`.
`actionPanelButtonSize`	No	Defines the width of buttons. Values can be `"normal"`, `"shrink"`, `"tiny"`, `"small"`, `"medium"`, `"large"` or `"huge"`. When using `"normal"` and `"shrink"`, buttons are sized according to the text or image, where `"shrink"` uses the minimum size needed to display the content of the button. Default is `"normal"`.
`actionPanelButtonSpace`	No	Defines the space between buttons. Values can be `"none"`, `"tiny"`, `"small"`, `"medium"`, `"large"` or `"huge"`. Default is `"medium"`.
`actionPanelScroll`	No	Defines if the action panel is "ring" - that is, when the last button is shown, pressing on the "down" button will show the first one again. Values can be `"0"` or `"1"`. Default is `"1"`.
`actionPanelScrollStep`	No	Defines how the action panel should scroll when clicking the "down" button, to shown the next visible buttons. Values can be `"line"` or `"page"`, default is `"line"`. When `"line"`, the panel will scroll by one line, and then show only the next button. When `"page"`, the scrolling will be done page by page.
`actionPanelHAlign`	No	Defines the alignment of the action panel when `actionPanelPosition` is `"top"` or `"bottom"`. Values can be `"left"`, `"right"` or `"center"`.

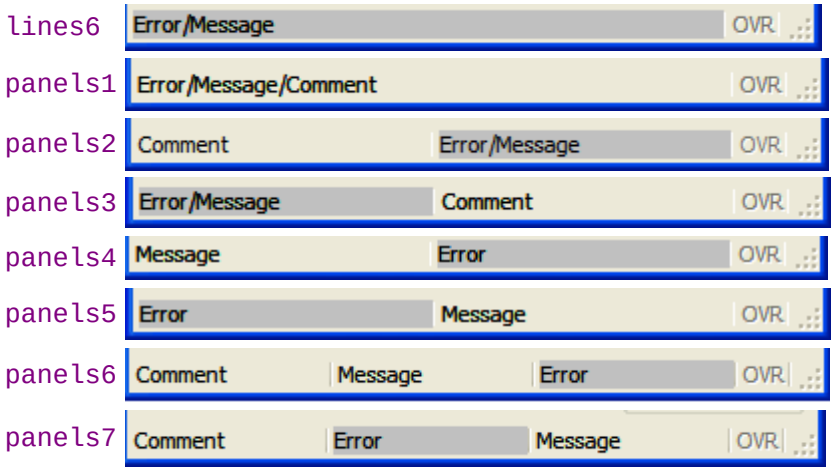
		Default is `"left"`.
`ringMenuPosition`	No	Defines the position of the ring menu frame (MENU). Values can be `"none"`, `"top"`, `"left"`, `"bottom"` or `"right"`. Default is `"right"`.
`ringMenuButtonSize`	No	Defines the width of buttons. Values can be `"normal"`, `"shrink"`, `"tiny"`, `"small"`, `"medium"`, `"large"` or `"huge"`. When using `"normal"` and `"shrink"`, buttons are sized according to the text or image, where `"shrink"` uses the minimum size needed to display the content of the button. Default is `"normal"`.
`ringMenuButtonSpace`	No	Defines the space between buttons. Values can be `"none"`, `"tiny"`, `"small"`, `"medium"`, `"large"` or `"huge"`. Default is `"medium"`.
`ringMenuScroll`	No	Defines if the ring menu is "ring" - that is. when the last button is shown, pressing on the "down" button or using the "down" key will show the first one again. Values can be `"0"` or `"1"`. Default is `"1"`.
`ringMenuScrollStep`	No	Defines how the ring menu should scroll when clicking "down" when the visible button is selected, to show the next buttons. Values can be `"line"` or `"page"`, default is `"line"`. When `"line"`, the menu will scroll by one line, and show only the next button. When `"page"`, the scrolling will be done page by page.
`ringMenuHAlign`	No	Defines the alignment of the ring menu when `ringMenuPosition` is `"top"` or `"bottom"`. Values can be `"left"`, `"right"` or `"center"`. Default is `"left"`.
`toolBarPosition`	No	Indicates the position of the toolbar, when a toolbar is defined. Values can be `"none"`, `"top"`, `"left"`, `"bottom"` or `"right"`. Default is `"top"`.

`commentPosition`	No	Defines the output type of the status bar comment field. Values can be `"statusbar"`, `"popup"`, `"statustip"`, `"both"`. `"popup"` will bring a window popup to the front; it should be used with care, since it can annoy the user. `"statustip"` will add a small "down" arrow button that will show the popup once the user clicks on it; this can be useful to display very long text. Default is `"statusbar"`.
`messagePosition`	No	Defines the output type of the status bar message field. Values can be `"statusbar"`, `"popup"`, `"statustip"`, `"both"`. Default is `"statusbar"`.
`errorMessagePosition`	No	Defines the output type of the status bar error field. Values can be `"statusbar"`, `"popup"`, `"statustip"`, `"both"`. Default is `"statusbar"`.
`statusBarType`	No	Defines the type of status bar the window will display. See below for all possible values. Default is `"default"`.

Status bar types:

The next table shows all possible status bar types you can set with the **statusBarType** attribute for Windows:

Value	Screenshot
`default`	
`lines1`	
`lines2`	
`lines3`	
`lines4`	
`lines5`	



StartMenu:

The following Window style attributes modify how the window will manage StartMenus.

Attribute	Inheritance	Description
`startMenuPosition`	No	Indicates the position of the startmenu, when one is defined. Values can be `"none"`, `"tree"`, `"menu"` or `"poptree"`. `"tree"` - the startmenu is displayed as a treeview, always visible on the right side of the window. `"menu"` - the startmenu is displayed as a pull-down menu, always visible at the top of the window. `"poptree"` - the startmenu is displayed as a treeview in a popup window that can be opened with a short-cut (see `startMenuShortcut`). Default is `"none"`.
`startMenuSize`	No	Defines the size of the startmenu, when one is defined and the position is defined as `"tree"` or `"poptree"`. The values can be `"tiny"`, `"small"`, `"medium"`, `"large"` or `"huge"`. Default is `"medium"`. Note: the size will also depend on the font used for the startmenu.
`startMenuShortcut`	No	Defines the shortcut key to open a startmenu, when the position is defined as `"poptree"`.

		Default is `"control-shift-F12"`.
`startMenuAccelerator` `startMenuExecShortcut2`	No	Defines the shortcut keys to execute the select startmenu item, when the position is defined as `"tree"` or `"poptree"`. By default, `"space"`, `"enter"` and `"return"` start the application linked to the current item.

MDI Container Style Attributes

The following table shows the presentation attributes for the MDI container:

Attribute	Inheritance	Description
`windowMenu`	No	Defines if the MDI Container should display an automatic "Window" menu, which holds the *Cascade* and *Tile* features, and list of open Windows.

Table Style Attributes

The following table shows the presentation attributes for Tables:

Attribute	Inheritance	Description
`forceDefaultSettings`	No	Indicates if the table must be initialized with the saved columns positions and sizes. By default, tables are re-opened with column positions and sizes they had when the window was closed. You can force the use of the initial settings with this attribute. Default is `"0"`.
`highlightColor`	No	Defines the highlight color of rows for the table. For possible values, see Colors.
`highlightTextColor`	No	Defines the highlighted text color of rows for the table. For possible values, see Colors.
`highlightCurrentRow`	No	Indicates if the current row must be highlighted in a table. Values can be 1 or 0.

		By default, when a Table is in read-only mode (DISPLAY ARRAY), the front-end automatically highlights the current row. But in editable mode (INPUT ARRAY), no row highlighting is done by default. You can change this behavior by setting this attribute to 1.
`highlightCurrentCell`	No	Indicates if the current cell must be highlighted in a table. Values can be 1 or 0. By default the current edit cell in table has a white background. You can change this behavior by setting this attribute to 1, to use the same color as when highlightCurrentRow is used. Only some type of cells, checkboxes for example, can be highlighted. Normal editor cells stay in white, because this is the editor background color.
`showGrid`	No	Indicates if the grid lines must be visible in a table. Values can be 1 or 0. By default, when a Table is in editable mode (INPUT ARRAY), the front-end displays grid lines in the table. You can change this behavior by setting this attribute to 0.
`headerHidden`	No	Defines if the horizontal header must be visible in a table. Values can be 1 or 0 (default).

ComboBox Style Attributes

The following table shows the presentation attributes for ComboBox:

Attribute	Inheritance	Description
`autoSelectionStart`	No	Defines the item from which the auto-selection will start, when pressing keys. Possible values are "first", "current". If 'first', the auto-selection will look for the first corresponding item after the first item of the object. If 'current', it will

	look for the first corresponding item after the current item of the object. Default is "current".

DateEdit Style Attributes

The following table shows the presentation attributes for DateEdit:

Attribute	Inheritance	Description
`firstDayOfWeek`	No	Defines the first day of the week to be displayed in the calendar. Possible values are "monday", "tuesday", "wednesday", "thursday", "friday", "saturday", "sunday". Default is "saturday".
`daysOff`	No	Defines the days of the week that are grayed out. Possible values are "monday", "tuesday", "wednesday", "thursday", "friday", "saturday", "sunday". Default is "saturday sunday".. The days of week can be combined, as shown.
`buttonIcon`	No	Defines the icon name to use for the button.

Label Style Attributes

The following table shows the presentation attributes for Label:

Attribute	Inheritance	Description
`textFormat`	No	Defines the rendering of the content of the widget. Possible values are "plain", "html". If 'plain', the value assigned to this widget is interpreted as plain text. If 'html', it is interpreted as HTML. The support for HTML is basic; for example, you can't add links, frames...

ProgressBar Style Attributes

The following table shows the presentation attributes for ProgressBar:

Attribute	Inheritance	Description
`percentageVisible`	No	Defines whether the current progress value is displayed. Possible values are "center", "system" and "no". If "center", the progress will be displayed in the middle of the progressbar. If "system", it will follow the system theme. If "no", no progress is displayed. Default is "no".

RadioGroup Style Attributes

The following table shows the presentation attributes for Radiogroup:

Attribute	Inheritance	Description
`autoSelectionStart`	No	Defines the item from which the auto-selection will start, when pressing keys. Possible values are "first", "current". If 'first', the auto-selection will look for the first corresponding item after the first item of the object. If 'current', it will look for the first corresponding item after the current item of the object. Default is "current".

TextEdit Style Attributes

The following table shows the presentation attributes for TextEdit:

Attribute	Inheritance	Description
`textFormat`	No	Defines the rendering of the content of the widget. Possible values are "plain", "html". If 'plain', the value assigned to this widget is interpreted as plain text. If 'html', it is interpreted as HTML. The support for HTML is basic. For example you can't add links or

		frames...
`textSyntaxHighlight`	No	Defines syntax highlighting for the widget. The value is currently limited to "per" for .per files syntax highlighting.
`wrapPolicy`	No	Defines where the text can be wrapped in word wrap mode. Possible values are "atWordBoundary" - the text will wrap at word boundaries, and "anywhere". - the text breaks anywhere, including within words. Default is "atWordBoundary"

Form Specification Files

Summary:

- Definition
- Concepts
  - Form Items
  - Form Fields
  - Item Tags
  - HBox Tags
  - Layout Tags
- Form file structure
  - SCHEMA section
  - ACTION DEFAULTS section
  - TOPMENU section
  - TOOLBAR section
  - LAYOUT section
    - HBOX container
    - VBOX container
    - GROUP container
    - FOLDER container
    - PAGE container
    - GRID container
    - SCROLLGRID container
    - TABLE container
  - TABLES section
  - ATTRIBUTES section
    - FIELD item type
    - EDIT item type
    - BUTTON item type
    - BUTTONEDIT item type
    - CANVAS item type
    - COMBOBOX item type
    - CHECKBOX item type
    - DATEEDIT item type
    - GROUP item type
    - IMAGE item type
    - LABEL item type
    - PROGRESSBAR item type
    - RADIOGROUP item type
    - SCROLLGRID item type
    - SLIDER item type
    - SPINEDIT item type
    - TABLE item type
    - TEXTEDIT item type
    - TIMEEDIT item type
  - INSTRUCTIONS section
  - KEYS section
- Miscellaneous
  - Boolean Expressions

- Compiling Form Files
- Using Forms in Programs

See also: Form Attributes, Database Schema, Localized Strings, Windows and Forms, Dynamic User Interface.

Definition

Purpose:

A Form Specification File is a source file that defines an application screen. This file defines the disposition, presentation, and behavior of screen elements called Form Items.

Syntax:

```
filename.per
```

Notes:

1. A form specification file is a text-based source file using a specific syntax.
2. Form specification files have a **.per** suffix.
3. To be used by programs, form specification files must be compiled into **.42f** files with the fglform tool.
4. See the structure of a form specification file for more details about writing **.per** files.

Warnings:

1. Compiled form files must be distributed to production sites.

Concepts

To write a form specification file, you need to understand the following concepts:

- Form Items
- Form Fields
- Item Tags
- HBox Tags
- Layout Tags
- Form Structure

A form file is described with a specific structure, based on a layout definition using containers which hold Form Items.

Form file structure

A form specification file is defined by a set of sections, which must appear in the order listed below.

- The SCHEMA section
- The ACTION DEFAULTS section
- The TOPMENU section
- The TOOLBAR section
- The LAYOUT section
- The TABLES section
- The ATTRIBUTES section
- The INSTRUCTIONS section

Notes:

1. Each section must begin with the keyword for which it is named.
2. The `LAYOUT` and `ATTRIBUTES` sections are mandatory.
3. The `SCHEMA`, `TOPMENU`, `TOOLBAR`, `TABLES` and `INSTRUCTIONS` sections are optional.

SCHEMA Section

Each form specification file can begin with a `SCHEMA` section identifying the database schema (if any) on which the form is based. This can be any database schema that is defined with a database schema file. Form field data types can be automatically extracted from the schema file if you specify the table and column name in the form field definition (see ATTRIBUTES section).

Syntax 1:

```
SCHEMA { database[@dbserver] | string | FORMONLY }
```

1. This section is optional; if you do not specify it, database schema specification defaults to `SCHEMA FORMONLY`.
2. *database* is the name of the database schema to be used for the form compilation.
3. *dbserver* identifies the Informix database server (INFORMIXSERVER) (see warnings).
4. *string* can be a string literal containing the database name.
5. You can create a form that is not related to any database schema by using the `FORMONLY` keyword. When using this option, you must omit the TABLES section and define field data types explicitly in the ATTRIBUTES section.

Warnings:

1. The DATABASE instruction is supported for backward compatibility, we recommend to use SCHEMA instead.
2. The *database* and *dbserver* specifications are supported (but ignored) for backward compatibility with Informix form specification.
3. When using a specific database schema, the field data types are taken from the schema file **during compilation**. Make sure that the schema file of the development database corresponds to the production database; otherwise the form fields defined in the compiled version of your forms will not match the table structures of the production database.

Syntax 2:(supported for backward compatibility)

```
DATABASE { database[@dbserver] | string | FORMONLY } [ WITHOUT NULL
INPUT ]
```

The DATABASE syntax is supported for compatibility with Informix 4gl; using SCHEMA is recommended.

Notes:

1. This section is optional; if you do not specify it, database schema specification defaults to FORMONLY.
2. *database* is the name of the database schema to be used for the form compilation.
3. *dbserver* identifies the Informix database server (INFORMIXSERVER) (see warnings).
4. *string* can be a string literal containing the database name.
5. You can create a form that is not related to any database schema by using the FORMONLY keyword. When using this option, you must omit the TABLES section and define field data types explicitly in the ATTRIBUTES section.
6. The use of the WITHOUT NULL INPUT option is supported for backward compatibility, but is ignored.

ACTION DEFAULTS Section

The ACTION DEFAULTS section defines local action view default attributes for the form elements.

Syntax:

```
ACTION DEFAULTS
  ACTION action-identifier ( action-attribute [,...] )
  [...]
END
```

Notes:

1. The ACTION DEFAULTS section must appear after SCHEMA.
2. This section is optional.
3. *action-identifier* defines the name of the action.
4. *action-attribute* defines an attribute value. Valid attribute values include: TEXT, IMAGE, COMMENT, ACCELERATOR, ACCELERATOR2, ACCELERATOR3, ACCELERATOR4, DEFAULTVIEW, VALIDATE.

Usage:

The ACTION DEFAULTS section centralizes action view attributes (text, comment, image, accelerators) at the form level.

You give a list of ACTION elements and specify attributes for each action. The action is identified by the name following the ACTION keyword, and attributes are specified in a list between parenthesis.

The attributes defined in this section are applied to form action views like Buttons, Toolbar buttons, or Topmenu options, if the individual action views do not explicitly define their own attributes.

If an attribute is not found in the form action defaults, and has not been defined specifically for the individual action view, the runtime system searches for the attribute value in the global action defaults.

See Action Defaults and Interaction Model for more details about each attribute.

Example:

```
01 ACTION DEFAULTS
02   ACTION accept ( COMMENT="Commit order record changes" )
03   ACTION cancel ( TEXT="Stop", IMAGE="stop", ACCELERATOR=SHIFT-F2,
VALIDATE=NO )
04   ACTION print ( COMMENT="Print order information",
ACCELERATOR=CONTROL-P, ACCELERATOR2=F5 )
05   ACTION zoom1 ( COMMENT="Open items list", VALIDATE=NO )
06   ACTION zoom2 ( COMMENT="Open customers list", VALIDATE=NO )
07 END
```

TOPMENU Section

The TOPMENU section defines a pull-down menu with options that are bound to actions.

Syntax:

```
TOPMENU [menu-identifier] ( menu-attribute [,...] )
  group
```

```
  [...]
END
```

where *group* is:

```
GROUP group-identifier ( group-attribute [,...] )
  { command
  | group
  | separator
  } [...]
END
```

where *command* is:

```
COMMAND command-identifier ( command-attribute [,...] )
```

and *separator* is:

```
SEPARATOR [separator-identifier] ( separator-attribute [,...] )
```

Notes:

1. The `TOPMENU` section must appear in the sequence described in Form File Structure.
2. This section is optional.
3. *menu-identifier* defines the name of the top menu (optional).
4. *group-identifier* defines the name of the group.
5. *command-identifier* defines the name of the action to bind to. The action name can be prefixed with the sub-dialog identifier.
6. *separator-identifier* defines the name of the top menu separator (optional).
7. *menu-attribute* can be: TAG.
8. *group-attribute* is one of: TEXT, IMAGE, COMMENT, TAG, HIDDEN.
9. *command-attribute* is one of: TEXT, IMAGE, COMMENT, TAG, HIDDEN, ACCELERATOR.
10. *separator-attribute* is one of: TAG, HIDDEN.

Usage:

The `TOPMENU` section is provided to define a pull-down menu in a form. You build a tree of `GROUP` elements to design the pull-down menu. A `GROUP` can contain `COMMAND`, `SEPARATOR` or `GROUP` children. A `COMMAND` defines a pull-down menu option that triggers an action when it is selected. In the Topmenu specification, *command-identifier* defines which action a menu option is bound to. For example, if you define a Topmenu option as "`COMMAND zoom`", it can be controlled by an "`ON ACTION zoom`" clause in an interactive instruction.

The Topmenu commands are enabled according to the actions defined by the current interactive instruction, which can be MENU, INPUT, INPUT ARRAY, DISPLAY ARRAY or CONSTRUCT. See also Interaction Model for more details about action management.

You can use the Predefined Actions to bind Topmenu commands to common actions such as dialog validation and cancellation.

An accelerator name can be defined for a TopMenu Command; this accelerator name will be used for display in the command item. You must define he same accelerator in the Action Defaults section for the action name of the TopMenu command.

Example:

```
01 TOPMENU myTopMenu
02   GROUP form (TEXT="Form")
03      COMMAND help (TEXT="Help", IMAGE="quest")
04      COMMAND quit (TEXT="Quit")
05   END
06   GROUP edit (TEXT="Edit")
07      COMMAND accept (TEXT="Validate", IMAGE="ok", TAG="acceptMenu")
08      COMMAND cancel (TEXT="Cancel", IMAGE="cancel")
09      SEPARATOR
10      COMMAND editcut   -- Gets its decoration from action defaults
11      COMMAND editcopy  -- Gets its decoration from action defaults
12      COMMAND editpaste -- Gets its decoration from action defaults
13   END
14   GROUP records (TEXT="Records")
15      COMMAND append (TEXT="Add", IMAGE="plus")
16      COMMAND delete (TEXT="Remove", IMAGE="minus")
17      COMMAND update (TEXT="Modify", IMAGE="accept")
18      SEPARATOR (TAG="lastSeparator")
19      COMMAND search (TEXT="Search", IMAGE="find")
20   END
21 END
```

TOOLBAR Section

The TOOLBAR section defines a toolbar with buttons that are bound to actions.

Syntax:

```
TOOLBAR [toolbar-identifier] [ ( toolbar-attribute [,...] ) ]
  { ITEM item-identifier  [ ( item-attribute [,...] ) ]
  | SEPARATOR [separator-identifier] [ ( separator-attribute [,...] ) ]
  }
  [...]
END
```

Notes:

1. The TOOLBAR section must appear in the sequence described in Form File Structure.
2. This section is optional.
3. *toolbar-identifier* defines the name of the toolbar (optional).

4. *item-identifier* defines the name of the action to bind to. Can be prefixed with the sub-dialog identifier.
5. *separator-identifier* defines the name of the top menu (optional).
6. *toolbar-attribute* is one of: TAG, BUTTONTEXTHIDDEN.
7. *item-attribute* is one of: TAG, TEXT, IMAGE, COMMENT, HIDDEN.
8. *separator-attribute* is one of: TAG, HIDDEN.

Usage:

The `TOOLBAR` section defines a toolbar in a form. A `TOOLBAR` section defines a set of `ITEM` elements that can be grouped by using a `SEPARATOR` element. Each `ITEM` defines a toolbar button associated to an action by name. The `SEPARATOR` keyword specifies a vertical line.

The Toolbar buttons are enabled according to the actions defined by the current interactive instruction, which can be MENU, INPUT, INPUT ARRAY, DISPLAY ARRAY or CONSTRUCT. See also Interaction Model for more details about action management. You can use the Predefined Actions to bind toolbar buttons to common actions such as dialog validation and cancellation.

The `TOOLBAR` supports the BUTTONTEXTHIDDEN attribute to hide the labels of buttons. Button labels are visible by default.

Example:

```
01 TOOLBAR
02   ITEM accept ( TEXT="Ok", IMAGE="ok" )
03   ITEM cancel ( TEXT="Cancel", IMAGE="cancel" )
04   SEPARATOR
05   ITEM editcut   -- Gets its decoration from action defaults
06   ITEM editcopy  -- Gets its decoration from action defaults
07   ITEM editpaste -- Gets its decoration from action defaults
08   SEPARATOR ( TAG="lastSeparator")
09   ITEM append ( TEXT="Append", IMAGE="add" )
10   ITEM update ( TEXT="Update", IMAGE="modify" )
11   ITEM delete ( TEXT="Delete", IMAGE="del" )
12   ITEM search ( TEXT="Search", IMAGE="find" )
13 END
```

LAYOUT Section

The `LAYOUT` section defines the graphical alignment of the form by using a tree of *layout containers*.

Syntax:

```
LAYOUT [(attribute[=value][,...])]
  root-layout-container
[END]
```

Notes:

1. The LAYOUT section must appear in the sequence described in Form File Structure.
2. This section is mandatory, unless you use a SCREEN section for backward compatibility.
3. *attribute* can, for example, be 'TEXT' to define the title of the topwindow.
4. *root-layout-container* is the first container that holds other containers.
5. Indentation is supported in the LAYOUT section.
6. The END keyword is optional.

Attributes:

MINHEIGHT, MINWIDTH, TEXT, TAG, STYLE, VERSION, SPACING, WINDOWSTYLE.

Usage:

You define the layout tree of the form by associating *layout containers*. Different kind of layout containers are provided, each of them having a specific role. Some containers can hold children containers, while others can define a *screen area*. Containers using a *screen area* define a formatted region containing *static text labels*, *item tags* and *layout tags*. The END keyword is mandatory; it defines the end of a container block.

```
LAYOUT
  VBOX
    GRID
      grid-area
    END
    GROUP
      HBOX
        GRID
          grid-area
        END
        TABLE
          table-area
        END
      END
    END
  END
END
```

The above definition would result in a layout tree that looks like this:

```
-- VBOX
   |
   +-- GRID 1
   |
   +-- GROUP
       |
       +-- HBOX
           |
           +-- GRID 2
```

```
                |
                +-- TABLE 1
```

The layout section can also contain a simple `GRID` container (equivalent to a V3 SCREEN definition):

```
LAYOUT
  GRID
    grid-area
  END
END
```

Description of attributes:

The MINHEIGHT, MINWIDTH attributes can be used to specify a minimum width and height for the form. You typically use these attributes to force the form to get a bigger size as the default when it is first rendered. Note that if the front-end stores window sizes, these attributes will only be significant the first time the form is opened, or each time the VERSION attribute is changed.

The VERSION attribute can be used to specify a version for the form. This allows you to indicate that the form content has changed. Typically used to avoid having the front-end reload the saved window settings.

The TEXT attribute can be used to define the title of the window that will display the form. This attribute will automatically be applied to the parent window when a form is loaded. See Windows and Forms for more details.

The STYLE attribute defines the decoration style for form elements, you can for example define a font property for all form elements.

With the WINDOWSTYLE attribute, you can define the window type and decoration. This attribute will automatically be applied to the parent window when a form is loaded. See Windows and Forms for more details. For backward compatibility, the `STYLE` attribute is used as the default `WINDOWSTYLE` if this attribute is not used.

Example:

```
01 LAYOUT ( TEXT="Customers", WINDOWSTYLE="dialog", VERSION="1.20" )
```

SCREEN Section (supported for backward compatibility)

To support existing V3 form files, you can define a `SCREEN` section in place of `LAYOUT`.

Syntax:

```
SCREEN [ SIZE lines [ BY chars ] ] [ TITLE "title" ]
{
  { text | [ item-tag [ | item-tag ] [...] ] }
  [...]
```

```
}
[END]
```

Notes:

1. The SCREEN section must be used to design TUI mode screens.
2. This section is mandatory, unless you use a LAYOUT section instead.
3. *lines* is the number of characters the form can display vertically. The default is 24.
4. *chars* is the number of characters the form can display horizontally. The default is the maximum number of characters in any line of the screen definition.
5. *title* is the title for the topwindow.
6. The {} curly braces are used to delimit the body of the screen.
7. See LAYOUT section for the definition of *item-tag* and *text*.
8. The END keyword is optional.

The screen body

Inside the SCREEN section, you can define the position of text labels and form fields.

Example:

```
01 SCREEN
02 {
03   CustId  : [f001   ] Name: [f002               ]
04   Address : [f003                               ]
05             [f003                               ]
06 }
07 END
```

Layout Containers

Layout Containers are blocks holding other layout containers or defining a formatted screen region.

Syntax:

```
container-type [identifier] [(attribute[=value][,...])]
  child-container
  [...]
END
```

where *child-container* can be:

```
{
  VBOX [identifier] [(attribute[=value][,...])]
    child-container
    [...]
  END
```

```
|
  HBOX [identifier] [(attribute[=value][,...])]
    child-container
    [...]
  END
|
  GROUP [identifier] [(attribute[=value][,...])]
    child-container
    [...]
  END
|
  FOLDER [identifier] [(attribute[=value][,...])]
    PAGE [identifier] [(attribute[=value][,...])]
      child-container
      [...]
    END
    [...]
  END
|
  GRID [identifier] [(attribute[=value][,...])]
  {
   grid-area
  }
  END
|
  SCROLLGRID [identifier] [(attribute[=value][,...])]
  {
   scroll-area
  }
  END
|
  TABLE [identifier] [(attribute[=value][,...])]
  {
   table-area
  }
  END
}
```

Notes:

1. *container-type* defines the type of container. A container type can be one of the keywords listed below.
2. *identifier* is an optional name that can be used in the program to identify the container.
3. *attribute* is a predefined attribute name that can be used to customize the layout container.
4. *value* can be a quoted string, an integer, or a boolean value (TRUE/FALSE).
5. *grid-area* is a text block delimited by curly braces, containing *static text labels*, *item tags* and *layout tags*.
   See GRID for more details.
6. *scroll-area* is a text block similar to *grid-area*, except that you can define multiple rows for a list-grid view.
   See SCROLLGRID for more details.

7. *table-area* is a special kind of *grid-area*, used to define the columns of a screen array.
   See TABLE for more details.
8. The position of the opening curly brace defines the left-most character in a *screen- area*, *scroll-area* and *table-area*.
9. The *grid-area*, *scroll-area* and *table-area* must end with a line having a closing curly brace.
10. The `END` keyword is mandatory.

Type of Containers:

Different types of *layout containers* are provided, each of them having a specific usage:

Name	Can Hold	Description
`VBOX`	`VBOX, HBOX, GROUP, FOLDER, GRID, SCROLLGRID, TABLE`	Packs contained elements vertically, without any decoration.
`HBOX`	`VBOX, HBOX, GROUP, FOLDER, GRID, SCROLLGRID, TABLE`	Packs contained elements horizontally, without any decoration.
`GROUP`	`VBOX, HBOX, GROUP, FOLDER, GRID, SCROLLGRID, TABLE`	Decorates the contained element with a rounded box that has a title.
`FOLDER`	`PAGE`	Presents contained pages in a folder tab. Can only contain PAGE children!
`PAGE`	`VBOX, HBOX, GROUP, FOLDER, GRID, SCROLLGRID, TABLE`	Defines a page of a `FOLDER` container. Can only be used in FOLDER!
`GRID`	*`grid-area`*	Unique-record presentation with positioned fields and labels.
`SCROLLGRID`	*`scroll-area`*	Multiple-record presentation with positioned fields and labels.
`TABLE`	*`table-area`*	Record-list presentation with columns and rows.

Identifying Containers:

In most cases you do not need to give a name to a container because it is only used in the form file to define the layout. However, if you want to change some attributes at runtime, you must identify the container. You can give a name to the container by writing an *identifier* after the container type, for example:

```
01 GROUP group1 (TEXT="Customer")
```

In this example, the group name is '`group1`', and it can be used in a program to identify the element:

```
01 DEFINE w ui.Window
02 DEFINE g om.DomNode
03 LET w = ui.Window.getCurrent()
04 LET g = w.findNode("Group","group1")
05 CALL g.setAttribute("text","This is the first group")
```

HBOX Container

The `HBOX` container automatically packs the contained elements horizontally from left to right. Contained elements are packed in the order in which they appear in the `LAYOUT` section of the form file. No decoration is added when you use a `HBOX` container. By combining `VBOX` and `HBOX` containers, you can define any alignment you choose.

Syntax:

```
HBOX [identifier] [(attribute[=value][,...])]
  layout-container
  [...]
END
```

Attributes:

COMMENT, FONTPITCH, HIDDEN, STYLE, SPLITTER, TAG.

Example:

```
01 HBOX
02   GROUP ( TEXT = "Customer" )
03   {
04    ...
05   }
06   END
07   TABLE
08   {
09    ...
10   }
11   END
```

```
12 END
```

VBOX Container

The VBOX container automatically packs the contained elements vertically from top to bottom. Contained elements are packed in the order in which they appear in the LAYOUT section of the form file. No decoration is added when you use a VBOX container. By combining VBOX and HBOX containers, you can define any alignment you choose.

Syntax:

```
VBOX [identifier] [ (attribute[=value][,...])]
  layout-container
  [...]
END
```

Attributes:

COMMENT, FONTPITCH, HIDDEN, STYLE, SPLITTER, TAG.

Example:

```
01 VBOX
02   GROUP ( TEXT = "Customer" )
03   {
04    ...
05   }
06   END
07   TABLE
08   {
09    ...
10   }
11   END
12 END
```

GROUP Container

A GROUP container can be used to display a titled box (usually called a *groupbox*) around contained elements. To display a groupbox widget around a set of fields, you simply put a GROUP declaration around a GRID definition. If you want to include several children in a GROUP, you can add a VBOX or HBOX into the GROUP, to define how these elements are aligned.

Syntax:

```
GROUP [identifier] [(attribute[=value][,...])]
  layout-container
  [...]
END
```

Attributes:

COMMENT, FONTPITCH, STYLE, TAG, HIDDEN. TEXT.

Usage:

Note that when using the GROUP container syntax, you cannot set the GRIDCHILDRENINPARENT attribute. This attribute makes only sense if the parent of the GROUP is a GRID.

Example:

```
01 GROUP ( TEXT = "Customer" )
02   VBOX
03     GRID
04     {
05      ...
06     }
07     END
08     TABLE
09     {
10      ...
11     }
12     END
13   END
14 END
```

FOLDER Container

A FOLDER container can be used to display children (pages) inside a "folder tab" widget. You must define each folder page with a PAGE container inside the FOLDER container. Each PAGE container will be displayed on a separate folder page, accessed by TAB CONTROL click. If you want to include several containers in one page of a FOLDER, you can add a VBOX or an HBOX container to define how these elements are aligned.

Syntax:

```
FOLDER [identifier] [(attribute[=value][,...])]
  page-definition
  [...]
END
```

Attributes:

COMMENT, FONTPITCH, STYLE, TAG, HIDDEN.

In the above syntax, the *page-definition* defines one page of the folder. See PAGE container for more details.

PAGE Container

A PAGE container can only be a child of a FOLDER container. A PAGE container is defined as follows:

```
PAGE [identifier] [(attribute[=value][,...])]
  layout-container
  [...]
END
```

Attributes:

ACTION, COMMENT, FONTPITCH, STYLE, TAG, HIDDEN, IMAGE, TEXT.

Usage:

By default PAGE containers are used to group elements for decoration only. With the TABINDEX form field attribute, you can define which field gets the focus when a folder page is selected.

The TEXT attributes defines the label of the folder page. The IMAGE attribute can be used to specify which image to use as an icon.

If needed, you can use the ACTION attribute to bind an action to a folder page. When the page is selected, the program gets the corresponding action event.

Example:

```
01 FOLDER
02   PAGE p1 ( TEXT="Global info" )
03     GRID
04     {
05      ...
06     }
07     END
08   END
09   PAGE p2 ( IMAGE="list" )
10     TABLE
11     {
12      ...
13     }
14     END
```

```
15   END
16 END
```

GRID Container

The GRID container declares a formatted text block defining the dimensions and the positions of the logical elements of a screen for a unique-record presentation. With GRID, you can specify the position of labels, form fields for data entry or additional interactive objects such as buttons. You design the layout of a GRID by using static text, item tags, HBox tags, and layout tags.

Syntax:

```
GRID [identifier] [(attribute[=value][,...] )]
{
  { text
  | item-tag
  | hbox-tag
  | layout-tag
  | h-line }
  [...]
}
END
```

Notes:

1. *text* is literal text that will appear in the form as a static label.
2. *item-tag* defines the position and length of a Form Item.
3. *hbox-tag* defines the position and length of several Form Items inside an horizontal box.
4. *layout-tag* defines the position and length of a layout tag.
5. *h-line* is a set of dash characters defining a horizontal line.

Attributes:

COMMENT, FONTPITCH, STYLE, TAG, HIDDEN.

Usage:

A GRID container defines a layout area based on character cells. It is used to place Form Items such as labels, fields, or buttons at a specific position. Form items are located with item tags in the grid layout area. You can use layout tags to place some type of containers inside a grid.

Example:

Simple GRID example defining 3 labels and 3 fields:

```
01 GRID
02 {
03  Id:   [f1] Name: [f2    ]
04  Addr: [f3           ]
05 }
06 END
```

For more details about layout rules in grids, see Form Rendering.

SCROLLGRID Container

The SCROLLGRID container declares a formatted text block defining the dimensions and the position of the logical elements of a screen for a multi-record presentation. This container is similar to the GRID container, except that you can repeat the screen elements on several "row-templates", in order to design a multiple-record view that appears with a vertical scrollbar.

Syntax:

```
SCROLLGRID [identifier] [(attribute[=value][,...])]
{
  row-template
  [...]
}
END
```

where *row-template* is a text block containing:

```
{ text
| item-tag
| h-line }
[...]
```

Notes:

1. *text* is literal text that will appear in the form.
2. *item-tag* defines the position and length of a Form Item.
3. *h-line* is a set of dash characters defining a horizontal line.

Attributes:

COMMENT, FONTPITCH, GRIDCHILDRENINPARENT, STYLE, TAG, HIDDEN.

Usage:

Same layout rules apply as in a GRID container.

Example:

```
01 SCROLLGRID
02 {
03  Id:    [f001   ]   Name: [f002                        ]
04  Addr:  [f003                                          ]
05  ------------------------------------------------------
06  Id:    [f001   ]   Name: [f002                        ]
07  Addr:  [f003                                          ]
08  ------------------------------------------------------
09  Id:    [f001   ]   Name: [f002                        ]
10  Addr:  [f003                                          ]
11  ------------------------------------------------------
12  Id:    [f001   ]   Name: [f002                        ]
13  Addr:  [f003                                          ]
14  ------------------------------------------------------
15 }
16 END
```

TABLE Container

The `TABLE` container defines the presentation of a list of records, bound to a screen record list (also called "*screen array*"). When using this layout container with curly braces, the position of the static labels and item tags is automatically detected by the form compiler to build a graphical object displaying a list of records. Column titles for the table list can be defined in the table layout, or as attributes in the definition of the form fields that make up the table columns.

Syntax:

```
TABLE [identifier] [(attribute[=value][,...])]
{
 title [...]
[identifier [|...] ]
[...]
}
END
```

Notes:

1. *title* is the text to be displayed as column title.
2. *identifier* references a Form Item.

Warnings:

1. The screen record definition must have exactly the same number of columns as the `TABLE` container.
2. If column titles are used in the table layout, the first line of a table-area must be a set of text entries defining the column titles. The column title can contain blank characters, but several blanks will be interpreted as a column title separator.

3. When column titles are used in the table layout, the second line defines the columns, referencing form fields receiving data. Otherwise, the first line defines the columns. This line can be repeated several times on the other lines.

Attributes:

COMMENT, DOUBLECLICK, HIDDEN, FONTPITCH, STYLE, TAG, UNHIDABLECOLUMNS, UNMOVABLECOLUMNS, UNSIZABLECOLUMNS, UNSORTABLECOLUMNS, WANTFIXEDPAGESIZE, WIDTH, HEIGHT.

Usage:

To create a table view, you must define the following elements in the form file:

1. The layout of the list, with a `TABLE` container in the LAYOUT section.
2. The column data types and field properties, in the ATTRIBUTES section.
3. The field list definition to group form fields together with a screen record, in the INSTRUCTIONS section.

The default width and height of a table are defined respectively by the columns and the number of lines used in the table layout. You can overwrite the defaults by specifying the WIDTH and HEIGHT attributes, as in the following example:

```
01 TABLE t1 ( WIDTH = 5 COLUMNS, HEIGHT = 10 LINES )
```

You design the `TABLE` layout in curly braces. The layout can contain column titles as well as the tag identifiers for each column's form fields. The form compiler can associate column titles in the table layout with the form field columns if they are aligned properly - the first character of each column title must appear at the same text column position as the first character of the tag identifier for the form field. In the following example, `Title1` and `Title2` will be associated with `column1` and `column2`, but `Title3` cannot be identified as a column title:

```
01 TABLE
02 {
03  Title1      Title2            Title3
04 [column1    |column2              |column3              ]
05 [column1    |column2              |column3              ]
06 [column1    |column2              |column3              ]
07 }
08 END
```

The column data type and additional properties are defined in the ATTRIBUTES section, as form fields:

```
01 ATTRIBUTES
02 EDIT column1 = customer.cust_num;
03 EDIT column2= customer.cust_name,
04 EDIT column3= customer.cust_cdate;
```

As an alternative, you can set the column titles of a table container by using the TITLE attribute in the definition of the form fields, instead of using column header text in the table layout. This allows you to use Localized Strings for the column titles:

```
01 TABLE
02 {
03  [c1  |c2          |c3         ]
04  [c1  |c2          |c3         ]
05  [c1  |c2          |c3         ]
06 }
07 END
08 ...
09 ATTRIBUTES
10 EDIT c1 = FORMONLY.col1, TITLE=%"Num";
11 LABEL c2 = FORMONLY.col2, TITLE=%"Name";
12 CHECKBOX c3 = FORMONLY.col3, TITLE=%"Status", VALUECHECKED="Y",
VALUEUNCHECKED="N";;
13 ...
```

Each form field must be grouped in the INSTRUCTIONS section in a screen record definition:

```
01  SCREEN RECORD listarr( col1, col2, col3 )
```

The screen record identifies the record list in BDL programs when you use an INPUT ARRAY or DISPLAY ARRAY instruction:

```
01 INPUT ARRAY custarr FROM listarr.*
```

Warning: The screen record definition must have exactly the same columns as the TABLE container.

By default, the current row in a `TABLE` is highlighted in display mode (DISPLAY ARRAY), but it is not highlighted in input mode (INPUT ARRAY, CONSTRUCT). You can set decoration attributes of a table with a style; see style attributes of the Table class.

With the DOUBLECLICK attribute, you can define a particular action to be send when the user double-clicks on a row.

After a dialog execution, the current row may be unselected, depending on the KEEP CURRENT ROW dialog attribute.

Example:

```
01 SCHEMA videolab
02 LAYOUT ( TEXT="Customer list" )
03 TABLE ( TAG="normal" )
04 {
05   Num     Customer name              Date       S
06  [c1     |c2                        |c3        |c4 ]
07  [c1     |c2                        |c3        |c4 ]
08  [c1     |c2                        |c3        |c4 ]
```

```
09  [c1     |c2                          |c3        |c4 ]
10  [c1     |c2                          |c3        |c4 ]
11  [c1     |c2                          |c3        |c4 ]
12 }
13 END
14 END
15 TABLES
16 customer
17 END
18 ATTRIBUTES
19 EDIT c1 = customer.cust_num;
20 EDIT c2 = customer.cust_name;
21 EDIT c3 = customer.cust_cdate;
22 CHECKBOX c4 = customer.cust_status;
23 END
24 INSTRUCTIONS
25  SCREEN RECORD custlist( cust_num, cust_name, cust_cdate,
cust_status )
26 END
```

Form Items

A **Form Item** defines the properties of a form element. For example, a Form Item can define an input area (such as an `EDIT` field), a push `BUTTON`, or a layout element (such as a `GROUPBOX`).

The position and length of a Form Item is defined by a place holder called 'tag' (Item Tag, HBox Tag or Layout Tag). Such place holders are used in the body of `GRID`, `SCROLLGRID` and `TABLE` containers.

The appearance and the behavior of a Form Item is defined in the ATTRIBUTES section.

Form Items defined for data management are called **Form Fields**.

Example:

```
01 LAYOUT( TEXT = "Vehicles" )
02   GRID
03   {
04    <G g1                                  >
05     Number:   [f1           ]
06     Name:     [f2                       ]
07               [b1             ]
08
09   }
10   END
11 END
12 ATTRIBUTES
13   GROUP  g1 : group1, TEXT="Identification" ;
14   EDIT   f1 = vehicle.num;
```

```
15   EDIT   f2 = vehicle.name;
16   BUTTON b1 : validate, TEXT="Ok";
17 END
```

Form Fields

A Form Field is a Form Item dedicated to data management. It associates a Form Item with a screen record field.

A Form Field defines an area where the user can view and edit data, depending on its description in the form specification file and the interactive statements in the program. The interactive instruction in your program must mediate between screen record fields and database columns by using program variables.

Form Fields linked to database columns

Unless a form field is FORMONLY, its field description must specify the SQL identifier of a database column as the name of the display field. Fields are associated with database columns only during the compilation of the form specification file. During the compilation process, the form compiler examines the database schema file to identify the data type of the column, and two optional files, containing the definitions of the **syscolval** and **syscolatt** tables, for default values of the attributes that you have associated with any columns of the database.

Syntax:

```
item-type item-tag = [table.]column
                     [ , attribute-list ] ;
```

After the form compiler extracts any default attributes and identifies data types from the schema file, the association between fields and database columns is broken, and the form cannot distinguish the name or synonym of a table or view from the name of a screen record.

Example:

```
01 EDIT f001 = customer.fname, NOT NULL, REQUIRED, COMMENTS="Customer
name" ;
```

The programs only have access to screen record fields, in order to display or input data using program variables. Regardless of how you define them, there is no implicit relationship between the values of program variables, form fields, and database columns. Even, for example, if you declare a variable `lname LIKE customer.lname`, the changes that you make to the variable do not imply any change in the column value. Functional relationships among these entities must be specified in the logic of your program, through screen interaction statements, and through SQL statements. It is up to the programmer to determine what data a form displays and what to do with data values

that the user enters into the fields of a form. You must indicate the binding explicitly in any statement that connects variables to forms or to database columns.

FORMONLY Form Fields

FORMONLY form fields are not associated with columns of any database table or view. They can be used to enter or display the values of program variables. If the SCHEMA section specifies FORMONLY, this is the only kind of Form Item description that you can specify in the ATTRIBUTES section.

Syntax:

```
item-type item-tag = FORMONLY.field [ TYPE { LIKE [table.]column |
datatype [NOT NULL] } ]
                     [ , attribute-list ] ;
```

The optional data type specification uses a restricted subset of the data type declaration syntax that the DEFINE statement supports. When using CHAR or VARCHAR data types, you do not have to specify the length, because it is defined by the size of the field tag in the LAYOUT section. Additionally, the STRING data type is not supported.

The NOT NULL keywords specify that if you reference the form field in an INPUT statement, the user must enter a non-null value in the field. This option is more restrictive than the REQUIRED attribute, which permits the user to enter a NULL value.

Example:

```
01 EDIT f001 = FORMONLY.total TYPE DECIMAL(10,2), NOENTRY ;
```

Field Input Length

The input length of a form field is the number of characters the user can type into the text editor. The input length is defined by the data type of the program variable used by the dialog and the width of the item tag. The width of the item tag is defined by the number of ASCII characters used between the square braces.

```
01      [f01 ]    -- width = 4
```

When the program variable is defined with a DATE data type, the input length is the maximum of:

- the number of characters of the FORMAT attribute, if this attribute is used
- the width of the form field, defined by the item-tag.

When the program variable is defined with a character or numeric data type, the input length is defined by the width of the form field. This means, the maximum number of characters a user can input is defined by the size of the item-tag in the form.

For character data types, you can specify the SCROLL attribute to force the input length to be as large as the program variable. For example, when using a CHAR(20) variable with a form field defined with an item-tag large as 3 characters, the input length will be 20 characters instead of 3.

Warning:

1. In a multi-byte character set, the input length represents the number of bytes in the locale of the application. In other words, it is the number of bytes used by the character string in the character set used by the runtime system. For example, when using a Chinese BIG5 encoding, a field having a width of 6 ASCII characters in the form file, represents a maximum input length of 6 bytes. In BIG5, Latin characters (a,b,c) use one byte each, while Chinese characters use 2 bytes. So, if the input length is 6, the user can enter 6 Latin characters like "abcdef", or 4 Latin characters and one Chinese, or 3 Chinese characters.

Remark: When you display a program variable to a form field with the DISPLAY TO or DISPLAY BY NAME instruction, the input length is used to truncate the text resulting from the data conversion. If the resulting text does not fit into the input length, the runtime system displays star characters (asterisks) in the form field, to indicate a size overflow.

Item Tags

An Item Tag defines the position and size of a Form Item in a *grid-area* of a GRID or SCROLLGRID.

Syntax:

```
[identifier [-] [|...] ]
```

Notes:

1. *identifier* references a Form Item.
2. The optional `-` dash defines the real width of the element.
3. The `|` pipe can be used as item tag separator (equivalent to `][`).

Usage:

An item tag is delimited by square braces (`[]`) and contains an identifier used to reference the description of the Form Item in the ATTRIBUTES section.

Each item tag must be indicated by left and right delimiters to show the length of the item and its position within the container layout. Both delimiters must appear on the same line. You must use left and right braces (`[]`) to delimit item tags. The number of characters and the delimiters define the width of the region to be used by the item:

```
01 GRID
02 {
03   Name:  [f001                                ]
04 }
05 END
```

The Form Item position starts after the open square brace and the length is defined by the number of characters between the square braces. The following example defines a Form Item starting at position 3, with a length of 2:

```
01 GRID
02 {
03 1234567890
04  [f1]
05 }
06 END
```

By default, the real width of the Form Item is defined by the number of characters used between the tag delimiters. For some special items like BUTTONEDIT, COMBOBOX and DATEEDIT, the width of the field is adjusted to include the button. The form compiler computes the width as: *width=nbchars*-2 if *nbchars*>2:

```
01 GRID
02 {
03  1234567
04 [f1     ]  -- this EDIT gets a width of 7
05 [f2     ]  -- this BUTTONEDIT gets a width of 5 (7-2)
06 }
07 END
```

If the default width generated by the form compiler does not fit, the - dash symbol can be used to define the real width of the item. In the following example, the Form Item occupies 7 grid cells, but gets a real width of 5 (i.e. for an EDIT field, you would be able to enter 5 characters):

```
01 GRID
02 {
03  1234567
04 [f1   - ]
05 }
06 END
```

To make two items appear directly next to each other, you can use the pipe symbol (|) to indicate the end of the first item and the beginning of the second item:

```
01 GRID
02 {
03   Info:  [f001    |f002            |f003    ]
04 }
05 END
```

If you need the form to support items with a specific height (more that one line), you can specify *multiple-segment* item tags that occupy several lines of a *grid-area*. To create a

multiple-segment item, repeat the item tag delimiters without the item *identifier* on successive lines:

```
01 GRID
02 {
03   Multi-segment: [f001                              ]
04                  [                                  ]
05                  [                                  ]
06                  [                                  ]
07                  [                                  ]
08 }
09 END
```

Warnings:

1. This notation applies to the new LAYOUT section only. For backward compatibility (when using a SCREEN section), multiple-segment items can be specified by repeating the *identifier* in sub-lines.

If the same item tag (i.e. using the same *identifier*) appears more than once in the layout, it defines a column of a screen array:

```
01 GRID
02 {
03   Single-line array:
04     [f001          ] [f002          ]  [f003          ]
05     [f001          ] [f002          ]  [f003          ]
06     [f001          ] [f002          ]  [f003          ]
07     [f001          ] [f002          ]  [f003          ]
08 }
09 END
```

You can even define a multi-line list of fields:

```
01 GRID
02 {
03   Multi-line array:
04     [f001          ]  [f002          ]
05         [f003                                        ]
06     [f001          ]  [f002          ]
07         [f003                                        ]
08     [f001          ]  [f002          ]
09         [f003                                        ]
10     [f001          ]  [f002          ]
11         [f003                                        ]
12 }
13 END
```

HBox Tags

An HBox Tag defines the position and size in a GRID of an horizontal box containing several Form Items.

Syntax:

```
[ element : [...] ]
```

where *element* can be:

```
{ identifier [-] | string-literal | spacer }
```

Notes:

1. *identifier* references a Form Item.
2. The optional - dash defines the real width of the element.
3. *string-literal* is quoted text that defines a static label.
4. *spacer* is zero or more blanks that define an invisible element that expends automatically.
5. The colon is a delimiter for HBox Tag elements.

Warnings:

1. HBox Tags are not allowed for fields of Screen Arrays; you will get a form compiler error as the AUI structure does not allow this. The client needs a Matrix Element directly in a Grid or a ScrollGrid to perform the necessary positioning calculations for the individual fields.

Usage:

HBox Tags are provided to control the alignment of Form Items in a grid. HBox tags allow you to stack Form Items horizontally without the elements being influenced by elements above or below. In an HBox, you can mix Form Items, static labels and spacers. A typical use of the HBox is to have zip-code/city form fields side by side with predictable spacing in-between.

An HBox tag is delimited by square braces ([]) and must contain at least one *string-list* or an *identifier* preceded or followed by a colon (:). A *string-list* is combination of *string-literals* (quoted text) and *spacers* (blank characters). The colon is a delimiter for HBox tag elements, which are included in the horizontal box.

The following example shows simple HBox tags:

```
01 GRID
02 {
03  ["Label:":        ]
04  [f001       :     ]
05  [  :f002          ]
06 }
```

```
07 END
```

In this example:

1. Line 03 contains two elements: a static label and a spacer.
2. Line 04 contains two elements: a form item and a spacer.
3. Line 05 contains two elements: a spacer followed by a form item.

An HBox tag defines the position and width (in grid cells) of several Form Items grouped inside an horizontal box. The position and width (in grid cells) of the horizontal box is defined by the square braces ([]) delimiting the HBox tag.

When using an *identifier*, you define the position of a Form Item which is described in the ATTRIBUTES section. When using a *string-list*, you can define static labels and/or spacers. The following example defines an HBox tag generating 7 items (a static label, a spacer, a Form Item, a spacer, a static label, a spacer and a Form Item):

```
01 GRID
02 {
03  ["Num:"  :num  :  :"Name:"  :name          ]
04 }
05 END
```

A *spacer* is an invisible element that automatically expands. It can be used to align elements left, right or center in the HBox. The following example defines 3 HBoxes with the same width. Each HBox contains one field. The first field is aligned to the left, the second is aligned to the right and third is centered:

```
01 GRID
02 {
03  [left  :              ]
04  [          :right     ]
05  [      :centered:     ]
06 }
07 END
08
09 ATTRIBUTES
10  LABEL left : label1, TEXT="LEFT";
11  LABEL right : label2, TEXT="RIGHT";
12  LABEL centered : label3, TEXT="CENTER";
13 END
```

When you use string literals, the quotes define where the label starts and stops. If there is free space after the quote that ends the label, then it is filled by a spacer. Consider the following example:

```
01 GRID
02 {
03  [     :"Label1"     ]
04  [          :"Label2"]
05 }
06 END
```

In this example:

1. Line 03 contains a spacer, followed by the static label, followed by another spacer. The quotation marks end the string literal; a colon is not required to delimit the label from the final spacer.
2. Line 04 contains a spacer, followed by the static label. Because there is no empty space between the end of the static label and the closing bracket of the HBox Tag ( ]).

A typical use of HBox tags is to vertically align some Form Items - that must appear on the same line - with one or more Form Items that appear on the other lines:

```
01 GRID
02 {
03  Id:      [num   :"Name: ":name        ]
04  Address: [street                  : ]
05           [zipcode:city               ]
06  Phones:  [phone        :fax          ]
07 }
08 END
```

In the above example, the form compiler will generate a grid containing 7 elements (3 Labels and 4 HBoxes):

1. The label "Id:",
2. The HBox A which defines 3 cells, where:
   - The field 'num' will occupy the cell (1,1),
   - The label "Name:" will occupy the cell (2,1),
   - The field 'name' will occupy the cell (3,1).
3. The label "Address:" will occupy cell (1,2),
4. The HBox B which defines 1 cell, where:
   - The field 'street' will occupy the cell (1,1).
5. The HBox C which defines 2 cells, where:
   - The field 'zipcode' will occupy the cell (1,1),
   - The field 'city' will occupy the cell (2,1).
6. The label "Phones:" will occupy cell (1,4),
7. The HBox B which defines 2 cells, where:
   - The field 'phone' will occupy the cell (1,1),
   - The field 'fax' will occupy the cell (2,1).

Inside an HBox tag, the positions and widths of elements are independent of other HBoxes. It is not possible to align elements over HBoxes. The position of items inside an HBox depends on the spacer and the real size of the elements. The following example does not align the items as you would expect, following the character positions in the layout definition:

```
01 GRID
02 {
03   ["Num:     "  :fnum  :        ]
04   ["Name:    "  :fname          ]
05 }
```

```
06 END
```

A big advantage in using elements in an HBox is that the fields gets their real sizes according to the .per definition. The following example illustrates the case:

```
01 GRID
02 {
03  MMMMM
04 [f1   ]
05 [f2 : ]
06 }
07 END
```

Here all items will occupy the same number of grid columns (5). The MMMMM static label will have the largest width and define the width of the 5 grid cells. The first field is defined with a normal item tag, and expands to the width of the 5 grid cells. The line 5 defines an HBox that will expand to the size of the 5 grid cells, according to the static label, but its child element - the field f2 - gets a size corresponding to the number of characters used before the ':' colon (i.e. 3 characters).

If the default width generated by the form compiler does not fit, the - dash symbol can be used to define the real width of the item. In the following example, the HBox tag occupies 20 grid cells, the first Form Item gets a width of 5, and the second Form Item gets a width of 3:

```
01 GRID
02 {
03  12345678901234567890
04 [f1   - :f2 -    :   ]
05 }
06 END
```

The - dash size indicator is especially useful in BUTTONEDIT, DATEEDIT and COMBOBOX form fields, for which the default width computed by the form compiler may not fit. See BUTTONEDIT for example.

In the following example, a static label is positioned above a TEXTEDIT field. The label will be centered over the TEXTEDIT field, and will remain centered as the field expands or contracts with the resizing of the window.

```
01 GRID
02 {
03  [ :"label": ]
04  [textedit   ]
05 }
06 END
07
08 ATTRIBUTES
09  TEXTEDIT textedit = formonly.textedit, STRETCH=BOTH;
10 END
```

Layout Tags

Layout Tags can be used to define layout containers inside the frame of a grid-based container.

Syntax:

```
<type [identifier]  >
 content
<                  >
```

or

```
<type [identifier]  >
 content
```

Notes:

1. A *layout tag* is delimited by angle braces (<>).
2. *type* defines the kind of layout tag to be inserted at this position.
3. *identifier* defines the name of the layout tag that can optionally be used in the ATTRIBUTE section to define attributes.
4. *identifier* must be unique in the form specification file.
5. *identifier* is optional.
6. *content* defines Form Items inside the layout tag.
7. Note that the (< >) ending tag is optional.

Usage:

While complex layout with nested frames can be defined with HBOX and VBOX containers, it is also possible to define a form with a complex layout by using layout tags within a grid.

A layout tag defines a *layout region* in a frame of a grid-based container (such as the GRID or SCROLLGRID containers).

A layout tag has a type that defines what kind of container will be generated in the compiled form. The following table shows the different type of layout tags:

Tag Type	Container Type	Description
G	GROUP	Defines a group box layout tag, resulting in the same presentation as the GROUP container.
T	TABLE	Defines a list view layout tag, resulting in the same presentation as the TABLE container.
S	SCROLLGRID	Defines a scrollable grid layout tag, resulting in the same presentation as the SCROLLGRID container.

In the ATTRIBUTE section, you can specify attributes for the element corresponding to the layout tag. In the following example, the layout tag **g1** is defined in the `ATTRIBUTE` section with the `GROUP` Form Item type to set the name and text:

```
01 LAYOUT
02 GRID
03 {
04 <G g1            >
05 [text1           ]
06 [                ]
07 [                ]
08 <                >
09 }
10 END
11 END
12 ATTRIBUTES
13 GROUP g1:group1, TEXT="Description";
14 TEXTEDIT text1=FORMONLY.text1;
15 END
```

The *layout region* is a rectangle, in which the width is defined by the length of the layout tag, and the height by a closing tag (`< >`) .

In the following example, the layout region defined by the layout tag named "group1" is shown in yellow:

```
01 <G group1                >
02
03
04 <                        >
```

Form Items must be placed inside the layout region, shown in light blue here:

```
01 <G group1                >
02  Item:     [f001        ]
03  Quantity: [f002    ]
04  Date:     [f003        ]
05 <                        >
```

Note that the `[ ]` square brace delimiters are not counted to define the width of an item tag. The width of the item is defined by the number of character between the square braces. Thus, the following layout is valid and can be compiled:

```
01 <G group1                >
02 [f001                    ]
03 [f002                    ]
04  Static labels must fit!!
05 <                        >
06 <T table1                >
07 [colA  |colB              ]
08 [colA  |colB              ]
09 [colA  |colB              ]
10 [colA  |colB              ]
```

You can place several layout tags on the same layout line in order to split the frame horizontally. The following example defines six layout regions (four group boxes and two tables):

```
01 <G group1          ><G group2                ><G group4        >
02  FName: [f001    ]  Phone: [f004           ] [f012           ]
03  LName: [f002    ]  EMail: [f005           ] [               ]
04 <                  ><                        >[               ]
05 <G group3                                    >[               ]
06 [f010                                        ][               ]
07 <                                            >< >
08 <T table1                   ><T table2                        >
09 [c11   |c12   |c13          ][c21   |c22                      ]
10 [c11   |c12   |c13          ][c21   |c22                      ]
11 [c11   |c12   |c13          ][c21   |c22                      ]
12 [c11   |c12   |c13          ][c21   |c22                      ]
13 <                           ><                                >
```

The `< >` closing layout tag is optional. When not specified, the end of the layout region is defined by the underlying layout tag or by the end of the current grid. However, the ending tag must be specified if the form compiler cannot detect the end of the layout region. This is usually the case with group layout tags. In the next example, the table does not need an ending layout tag because it is defined by the starting tag of the group, but the group needs and ending tag otherwise it would include the last field (*field3*). Additionally, if *field3* would have a different size, the form compiler would raise an error because the group and the last field geometry would conflict.

```
01 <T table1          >
02 [colA  |colB       ]
03 [colA  |colB       ]
04 [colA  |colB       ]
05 [colA  |colB       ]
06 [colA  |colB       ]
07 [colA  |colB       ]
08 <G group2          >
09 [field1            ]
10 [field2            ]
11 <                  >
10 [field3            ]
```

It is possible to mix container layout tags with singular form items. You typically put form items using a large area of the form, such as IMAGEs, TEXTEDITs. Note that the `[ ]` square brace delimiters are not used to compute the size of the singular form items:

```
01  <G group1                 >[image1          ]
02   FName: [f001             ][                ]
03   LName: [f002             ][                ]
04  <                         >[                ]
05 [textedit1                 |                 ]
06 [                          |                 ]
07 [                          |                 ]
```

Table layout tags can be embedded inside group layout tags:

```
01  <G group1                  >
02   <T table1                >
03   [colA  |colB             ]
04   [colA  |colB             ]
05   [colA  |colB             ]
06   [colA  |colB             ]
07  <                         >
```

HBox or VBox containers with splitter are automatically created by the form compiler in the following conditions:

- HBox is created when two or more stretchable elements are stacked side by side and touch each other (nothing space between).
- VBox is created when two or more stretchable elements are stacked vertically and touch each other (nothing space between).

Stretchable elements are containers such as TABLEs, or form items like IMAGEs, TEXTEDITs with STRETCH attribute.

The example below defines two tables stacked vertically, generating a VBox with splitter (note that ending tags are omitted):

```
01 <T table1          >
02 [colA  |colB       ]
03 [colA  |colB       ]
04 [colA  |colB       ]
05 [colA  |colB       ]
06 <T table2          >
07 [colC  |colD       ]
08 [colC  |colD       ]
```

Below the layout defines two stretchable TEXTEDITs placed side by side which would generate an automatic HBox with splitter. Note that to make both textedits touch you need to use a pipe delimiter in between:

```
01 [textedit1           |textedit2                ]
02 [                    |                        ]
03 [                    |                        ]
04 [                    |                        ]
```

The next layout example would make the form compiler create an automatic VBox with splitter to hold *table2* and *textedit1*, plus an HBox with splitter to hold *table1* and the first VBox (note that we must use a pipe character to delimit the end of *colB* and *textedit1* so that both tables can be placed side by side):

```
01 <T table1          ><T table2                 >
02 [colA  |colB       ][colC|colD                ]
03 [colA  |colB       ][colC|colD                ]
04 [colA  |colB       ][colC|colD                ]
05 [colA  |colB       ]textedit1                 ]
06 [colA  |colB       ]                          ]
07 [colA  |colB       ]                          ]
```

If you want to avoid automatic HBox or VBox with splitter creation, you must add blanks between elements:

```
01 <T table1          >  <T table2               >
02 [colA  |colB       ]  [colC|colD              ]
03 [colA  |colB       ]  [colC|colD              ]
04 [colA  |colB       ]  [colC|colD              ]
05 [colA  |colB       ]
06 [colA  |colB       ] [textedit1               ]
07 [colA  |colB       ] [                        ]
08 [colA  |colB       ] [                        ]
```

Examples:

The typical Ok/Cancel window:

```
01 LAYOUT
02 GRID
03 {
04 <G g1                        >
05 [com                         ]
06 <                            >
07 [            :bok   |bno     ]
08 }
09 END
10 END
11 ATTRIBUTES
12 LABEL com: comment;
13 BUTTON bok: accept;
14 BUTTON bno: cancel;
15 ...
```

The following example shows multiple uses of layout tags:

```
01 LAYOUT
02 GRID
03 {
04 <S scrollgrid1                        ><G g1            >
05  Ident: [f001   ] [f002              ] [text1           ]
06         [f003                        ] [                ]
07  Ident: [f001   ] [f002              ] [                ]
08         [f003                        ] [                ]
09  Ident: [f001   ] [f002              ] [                ]
10         [f003                        ] [                ]
11 <                                      ><                >
12 <G g2                                                    >
13 [text2                                                   ]
14 [                                                        ]
15 [                                                        ]
16 <                                                        >
17 <T t1                                                    >
18   Num      Name               State  Value
19 [col1    |col2              |col3  |col4              ]
20 [col1    |col2              |col3  |col4              ]
21 [col1    |col2              |col3  |col4              ]
```

```
[col1    |col2              |col3  |col4             ]
<                                                     >
}
END
END
ATTRIBUTES
GROUP g1:group1, TEXT="Comment";
GROUP g2: TEXT="Description";
TABLE t1:table1, UNSORTABLECOLUMNS;
...
```

Form layout example

```
LAYOUT ( TEXT = "Customer orders" )
  VBOX
    GROUP group1 ( TEXT = "Customer" )
      GRID
      {
        <G Name                                          >
         [f001                                          ]
        <                                                >
        <G Identifiers           ><G Contact             >
         FCode: [f002           ]  Phone: [f004         ]
         LNumb: [f003           ]  EMail: [f005         ]
        <                         ><                     >
      }
      END
    END
    TABLE
    {
      OrdNo  Date       Ship date   Weight
     [c01   |c02       |c03        |c04        ]
     [c01   |c02       |c03        |c04        ]
     [c01   |c02       |c03        |c04        ]
     [c01   |c02       |c03        |c04        ]
    }
    END
    FOLDER
      PAGE pg1 ( TEXT = "Address" )
        GRID
        {
           Address:  [f011                          ]
           State:    [f012           ]
           Zip Code: [f013      ]
        }
        END
      END
      PAGE pg2 ( TEXT = "Comments" )
        GRID
        {
          [f021                                     ]
          [                                         ]
          [                                         ]
```

```
41            [                                          ]
42          }
43          END
44        END
45      END
46    END
47 END
```

TABLES Section

The TABLES section lists every table or view referenced elsewhere in the form specification file (specifically the ATTRIBUTES section).

Syntax:

```
TABLES
[ alias = [database[@dbserver]:][owner.] ] table [,...]
[END]
```

Notes:

1. *alias* represents an alias name for the given table.
2. *table* is the name of the database table.
3. *database* is the name of the database of the table (see warnings).
4. *dbserver* identifies the Informix database server (INFORMIXSERVER) (see warnings).
5. *owner* is the name of the table owner (see warnings).

Usage:

This section is mandatory when form fields reference database columns defined in the schema file. The TABLES section must follow the LAYOUT section, and the SCHEMA section must also exist to define the database schema. The END keyword is optional.

Field identifiers in programs or in other sections of the form specification file can reference screen fields as *column*, *alias.colum*n, or *table.column*.

The same *alias* must also appear in screen interaction statements of programs that reference screen fields linked to the columns of a table that has an *alias*.

If a table requires the name of an *owner* or of a *databas*e as a qualifier, the TABLES section must also declare an alias for the table. The *alias* can be the same identifier as *table*.

Warnings:

1. For backward compatibility with the Informix form specification, the comma separator is optional and the *database*, *dbserver* and *owner* specifications are ignored.

Example:

```
01 SCHEMA stores
02 LAYOUT
03 {
04   ...
05 }
06 END
07 TABLES
08  customer,
09  orders
10 END
11 ...
```

ATTRIBUTES Section

The ATTRIBUTES section describes properties of the elements used in the form.

Syntax:

```
ATTRIBUTES
{ form-field-definition | form-item-definition }
[...]
[END]
```

where *form-field-definition* is:

```
item-type item-tag = field-name [ , attribute-list ] ;
```

where *form-item-definition* is:

```
item-type item-tag : item-name [ , attribute-list ] ;
```

Notes:

1. The ATTRIBUTES section is mandatory.
2. This section must follow the LAYOUT section or if present, the TABLES section.
3. The END keyword is optional.
4. Each *item-tag* used in the LAYOUT section must be described in this section.
5. *item-type* defines the type of the Form Item.
6. *item-tag* is the name of the screen element used in the LAYOUT section.
7. *field-name* defines the screen record field to be associated to the Form Item. See Field Definition for more details.

8. *item-name* identifies the Form Item, used to identify items not defined as Form Fields.
9. *attribute-list* defines the aspect and behavior of the Form Item. See Attribute List for more details.

Usage:

A Form Item definition is associated *by name* to an Item Tag or Layout Tag defined in the LAYOUT section.

In order to define a Form Field, the Form Item definition must use the equal sign notation to associate a screen record field with the Form Item. If the Form Item is not associated with a screen record field (for example, a push button), you must use the colon notation.

Form item definitions can optionally include an *attribute-list* to specify the appearance and behavior of the item. For example, you can define acceptable input values, on-screen comments, and default values for fields.

When no screen record is defined in the INSTRUCTION section, a default screen record is built for each set of Form Items declared with the same table name.

The order in which you list the Form Items determines the order of fields in the default screen records that the form compiler creates for each table.

Tips:

1. To define Form Items as form fields, you are not required to specify *table* unless the name *column* is not unique within the form specification. However, it is recommended that you always specify *table***.***column* rather than the unqualified *column* name. As you can refer to field names collectively through a screen record built upon all the fields linked to the same table, your forms might be easier to work with if you specify *table* for each field. For more information on declaring screen records, see the INSTRUCTIONS section.

Form Item Types:

The *item-type* defines the kind of Form Item, to indicate which graphical object must be used to display the form element. The following table describes the supported Form Item types:

Form Item Type	Description
BUTTON	Standard push button with a label or a picture.
BUTTONEDIT	Line edit box with a button on the right side.
CANVAS	Area reserved for drawing.
CHECKBOX	Boolean entry with a box and a text label.
COMBOBOX	Field with a button that opens a list of values.
DATEEDIT	Line edit box with a button that opens a calendar window.

EDIT	Simple line edit box for data input or display.
FIELD	Abstract form field that can be defined in schema files.
GROUP	Group container specified with a layout tag.
IMAGE	Area where a picture file can be displayed.
LABEL	Simple read-only text widget.
PROGRESSBAR	Progress bar widget to display an integer value.
RADIOGROUP	Field presented with a set of radio buttons.
SCROLLGRID	Scrollable grid container specified with a layout tag.
SLIDER	Slider widget to enter an integer value within a defined range.
SPINEDIT	Text editor to enter an integer value.
TABLE	Table container specified with a layout tag.
TEXTEDIT	Multi-line edit box for data input or display.
TIMEEDIT	Text editor to enter time values..

Warnings:

1. When used in a table some graphical objects are rendered only when the user enters in the field. For example `RadioGroup, CheckBox, ComboBox, ProgressBar ...`

Example:

```
01 ATTRIBUTES
02  EDIT f001 = player.name, REQUIRED, COMMENT="Enter player's name";
03  EDIT f002 = player.ident, NOENTRY;
04  COMBOBOX f003 = player.level, NOT NULL,
ITEMS=((1,"Beginner"),(2,"Normal"),(3,"Expert"));
05  CHECKBOX f004 = FORMONLY.winner, VALUECHECKED=1, VALUEUNCHECKED=0,
TEXT="Winner";
06  BUTTON b1 : print, TEXT="Print Report";
07  GROUP g1 : print, TEXT="Description";
08 END
```

Field Definition

The *field-name* as used in the ATTRIBUTES syntax, associates the Form Item to a screen record field to define a Form Field.

Syntax:

A field definition can reference a database column defined in the database schema files:

```
[table.]column
```

or, it can be defined as a FORMONLY field. The data type of the field is defined with an indirect reference to a database column or with an explicit data type:

```
FORMONLY.field [ TYPE { LIKE [table.]column | datatype [NOT NULL] } ]
```

Notes:

1. *table* is the name or alias of a table, synonym, or view, as declared in the TABLES section.
2. *column* is the unqualified SQL identifier of a database column.
3. *field* is an identifier associated with a FORMONLY form field (not associated with any database column).
4. *datatype* is any data type. When no data type is specified, the default is CHAR.

Example:

```
01 ATTRIBUTES
02  EDIT f001 = player.name, REQUIRED, COMMENT="Enter player's name";
03 END
```

Attribute List

The *attribute-list* as used in the ATTRIBUTES syntax describes how the runtime system should display and handle a Form Item.

Syntax:

```
attribute [ = { value | value-list } ] [,...]
```

where *value-list* is:

```
( { value | value-list } [,...] )
```

Notes:

1. *attribute* identifies the attribute.
2. *value* is a string, date or numeric literal, or predefined constant like TODAY.
3. *value-list* is a set of values separated by comma, supporting sub-set definitions as in "`(1,(21,22),(31,32,33))`".

Usage:

The attribute list can by used, for example, to supply a default value, limit the values that can be entered, or set the text and color of the Form Item.

FIELD Item Type

Purpose:

The `FIELD` item type defines a generic form field that can be defined in database schema files.

Syntax:

```
FIELD item-tag = field-name [ , attribute-list ] ;
```

Notes:

1. *item-tag* is an identifier that defines the name of the item tag.
2. *field-name* identifies the screen record field. See Field Definition for more details.
3. *attribute-list* defines the aspect and behavior of the Form Item.

Attributes:

COMMENT, DEFAULT, HIDDEN, NOT NULL, NOENTRY, REQUIRED, STYLE, SIZEPOLICY, TAG, TABINDEX.

Table Column only: UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, TITLE.

Example:

```
01 FIELD f001 = order.state, REQUIRED, STYLE="important";
```

Usage:

This item type defines a generic form field for data input or display. The real item type (i.e. the widget) and the attributes must be defined in the database schema files.

The definition of the form field is determined by the **.val** database schema file, based on the *field-name* (table.column). The item type (EDIT, COMBOBOX, etc) is defined by the ITEMTYPE attribute in the **.val** schema file.

By using this form field specification, you can centralize the definition of form fields in the database schema file, to enforce reusability. You can, for example, specify that the "order.state" database column is a COMBOBOX, with a list of ITEMS, as if the field was defined directly in the **.per** form specification file.

It is also possible to use the attributes defined in the database schema files with other Form Item types.

The attributes defined directly in the form specification file take precedence over the attributes defined in the database schema files.

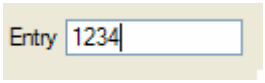
The database schema files can be edited manually or by using a tool.

See also Form Field, Database Schema.

EDIT Item Type

Purpose:

The `EDIT` item type defines a simple line-edit field.



Syntax:

```
EDIT item-tag = field-name [ , attribute-list ] ;
```

Notes:

1. *item-tag* is an identifier that defines the name of the item tag.
2. *field-name* identifies the screen record field. See Field Definition for more details.
3. *attribute-list* defines the aspect and behavior of the Form Item.

Attributes:

AUTONEXT, CENTURY, COLOR, COLOR WHERE, COMMENT, DEFAULT, DISPLAY LIKE, DOWNSHIFT, HIDDEN, FONTPITCH, FORMAT, INCLUDE, INVISIBLE, JUSTIFY, KEY, NOT NULL, NOENTRY, PICTURE, PROGRAM, REQUIRED, REVERSE, SAMPLE, STYLE, SCROLL, SIZEPOLICY, TAG, TABINDEX, UPSHIFT, VALIDATE LIKE, VERIFY.

Table Column only: UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, TITLE.

Example:

```
01 EDIT f001 = customer.state, REQUIRED, INCLUDE=(0,1,2);
```

Usage:

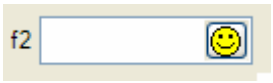
Defines a simple line edit box for data input or display.

See also Form Field.

BUTTONEDIT Item Type

Purpose:

The BUTTONEDIT item type defines a line-edit with a push-button that can trigger an action.



Syntax:

```
BUTTONEDIT item-tag = field-name [ , attribute-list ] ;
```

Notes:

1. *item-tag* is an identifier that defines the name of the item tag.
2. *field-name* identifies the screen record field. See Field Definition for more details.
3. *attribute-list* defines the aspect and behavior of the Form Item.

Attributes:

ACTION, AUTONEXT, CENTURY, COLOR, COLOR WHERE, COMMENT, DEFAULT, DISPLAY LIKE, DOWNSHIFT, FONTPITCH, HIDDEN, FORMAT, IMAGE, INCLUDE, INVISIBLE, JUSTIFY, KEY, NOT NULL, NOENTRY, PICTURE, PROGRAM, REVERSE, SAMPLE, SCROLL, SIZEPOLICY, STYLE, REQUIRED, TAG, TABINDEX, UPSHIFT, VALIDATE LIKE, VERIFY.

Table Column only: UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, TITLE.

Example:

```
01 BUTTONEDIT f001 = customer.state, REQUIRED, IMAGE="smiley",
ACTION=zoom;
```

Usage:

The BUTTONEDIT Form Item defines a line edit box with a button on the right side.

This kind of Form Item is typically used to open a new window for data selection.

The ACTION attribute defines the name of the action to be sent to the program when the user clicks on the button. The IMAGE attribute defines the picture to be displayed in the button.

When you use an HBox tag combined to the SAMPLE attribute, it is possible to specify the exact with of a BUTTONEDIT.

By default, the real width of BUTTONEDIT, DATEEDIT and COMBOBOX is computed as follows (*nbchars* represents the number of characters used in the form layout by the item tag to define the width of the item):

If *nbchars* is greater as 2, *width* = *nbchars* - 2; otherwise, *width* = *nbchars*.

For example:

```
01 LAYOUT
02 GRID
03 {
04  ButtonEdit A  [ba     ]
05  ButtonEdit B  [bb:    ]
06  ButtonEdit C  [bc   : ]
07  ButtonEdit D  [bd  -: ]
08 }
09 END
10 END
11 ATTRIBUTES
12 BUTTONEDIT ba = FORMONLY.ba, SAMPLE="0",  ACTION=zoom1;
13 BUTTONEDIT bb = FORMONLY.bb, SAMPLE="M",  ACTION=zoom2;
14 BUTTONEDIT bc = FORMONLY.bc, SAMPLE="Pi", ACTION=zoom3;
15 BUTTONEDIT bd = FORMONLY.bd, SAMPLE="0",  ACTION=zoom4;
16 END
```

Here the BUTTONEDIT **ba** occupies 7 grid columns and gets a real width of 5 (7-2). The SAMPLE attribute makes the edit field part as large as 5 characters '0' in the current font, so with this field you can input or display only 5 digits.

The BUTTONEDIT **bb**, which is in an HBox tag that occupies 7 grid columns, gets a width of 2. Since the SAMPLE attribute is "M", one can input 2 characters as wide as an "M".

The BUTTONEDIT **bc**, which is in an HBox tag that occupies 7 grid columns, gets a width of 3 (5-2). Since the SAMPLE attribute is "Pi", the edit field part will be as large as the word "Pi". (If SAMPLE contains more than 1 character it must have the same number of characters as in the field definition).

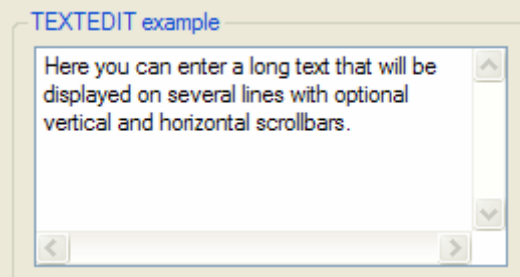
When using an HBox tag, one can explicitly specify the width of the field with the dash size indicator: The BUTTONEDIT **bd**, which is in an HBox tag that occupies 7 grid columns, gets a width of 4 (because of the dash size indicator). Since the SAMPLE attribute is "0", the edit field part will be as large as 4 digits.

See also Form Field.

TEXTEDIT Item Type

Purpose:

The `TEXTEDIT` item type defines a multi line-edit field.



Syntax:

```
TEXTEDIT item-tag = field-name [ , attribute-list ] ;
```

Notes:

1. *item-tag* is an identifier that defines the name of the item tag.
2. *field-name* identifies the screen record field. See Field Definition for more details.
3. *attribute-list* defines the aspect and behavior of the Form Item.

Attributes:

COLOR, COLOR WHERE, COMMENT, DEFAULT, DOWNSHIFT, FONTPITCH, HIDDEN, INCLUDE, KEY, NOT NULL, NOENTRY, PROGRAM, REQUIRED, SCROLLBARS, SIZEPOLICY, STYLE, STRETCH, TAG, TABINDEX, UPSHIFT, WANTTABS, WANTNORETURNS.

Table Column only: UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, TITLE.

Example:

```
01 TEXTEDIT f001 = customer.address, WANTTABS, SCROLLBARS=BOTH;
```

Usage:

This kind of form field allows the user to enter a long text on multiple lines.

By default, when the focus is in a `TEXTEDIT` field, the TAB key moves to the next field, while the RETURN key adds a NewLine (ASCII 10) character in the text. To control the user input when the TAB and RETURN keys are pressed, you can specify the WANTTABS and WANTNORETURNS attributes. When you specify WANTTABS, the TAB key is consumed by the `TEXTEDIT` field, and a TAB character is added to the text. When you specify WANTNORETURNS, the RETURN key is not consumed by the `TEXTEDIT` field, and the dialog is validated.

You can use the SCROLLBARS attribute to define vertical and/or horizontal scrollbars for the `TEXTEDIT` form field. By default, this attribute is set to `VERTICAL` for `TEXTEDIT` fields.

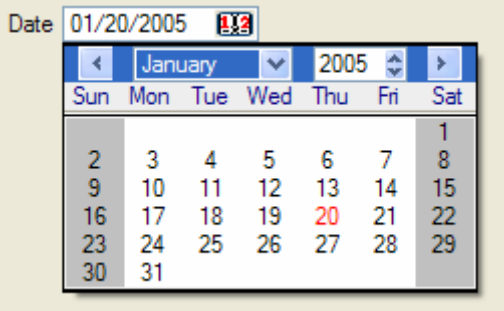
The STRETCH attribute can be used to force the `TEXTEDIT` field to stretch when the parent container is resized. Values can be `NONE`, `X`, `Y` or `BOTH`. By default, this attribute is set to `NONE` for `TEXTEDIT` fields.

See also Form Field.

DATEEDIT Item Type

Purpose:

The `DATEEDIT` item type defines a line-edit with a button that opens a calendar.



Syntax:

```
DATEEDIT item-tag = field-name [ , attribute-list ] ;
```

Notes:

1. *item-tag* is an identifier that defines the name of the item tag.
2. *field-name* identifies the screen record field. See Field Definition for more details.
3. *attribute-list* defines the aspect and behavior of the Form Item.

Attributes:

CENTURY, COLOR, COLOR WHERE, COMMENT, DEFAULT, FONTPITCH, FORMAT, HIDDEN, INCLUDE, JUSTIFY, KEY, NOT NULL, NOENTRY, REQUIRED, SAMPLE, SIZEPOLICY, STYLE, TAG, TABINDEX.

Table Column only: UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, TITLE.

Example:

```
01 DATEEDIT f001 = order.shipdate;
```

Usage:

This item type defines a line-edit with a button on the right that opens a calendar, dedicated to DATE input.

When you use an HBox tag combined with the SAMPLE attribute, it is possible to specify the exact wdith of a `DATEEDIT`. By default, the real width is computed as *width=nbchars-2* when *nbchars*>2. See BUTTONEDIT for more details.

Warnings:

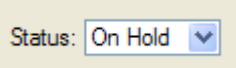
1. When the SAMPLE attribute is not specified, the default width for a `DATEEDIT` is dependent upon DBDATE, and FORMAT when this attribute is used in the field.

See also Form Field.

COMBOBOX Item Type

Purpose:

The `COMBOBOX` item type defines a line-edit with a drop-down list of values.



Syntax:

```
COMBOBOX item-tag = field-name [ , attribute-list ] ;
```

Notes:

1. *item-tag* is an identifier that defines the name of the item tag.
2. *field-name* identifies the screen record field. See Field Definition for more details.
3. *attribute-list* defines the aspect and behavior of the Form Item.

Attributes:

COLOR, COLOR WHERE, COMMENT, DEFAULT, DOWNSHIFT, FONTPITCH, HIDDEN, KEY, INCLUDE, INITIALIZER, ITEMS, NOT NULL, NOENTRY, QUERYEDITABLE, REQUIRED, SAMPLE, SCROLL, SIZEPOLICY, STYLE, UPSHIFT, TAG, TABINDEX.

Table Column only: UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, TITLE.

Example:

```
01 COMBOBOX f001 = customer.city,
ITEMS=((1,"Paris"),(2,"Madrid"),(3,"London"));
02 COMBOBOX f002 = customer.sector, REQUIRED, ITEMS=("A","B","C");
03 COMBOBOX f003 = customer.state, NOT NULL, INITIALIZER=myinit;
```

Usage:

This item type defines a line-edit with a button on the right side that opens a drop-down list.

The values of the list are defined by the ITEMS attribute. Real values can be associated to display values, for example:

```
((1,"Paris"),(2,"Madrid"),(3,"London"))
```

The INITIALIZER attribute allows you to define an initialization function for the `COMBOBOX`. This function will be invoked at runtime when the form is loaded, to fill the item list dynamically with database records, for example. It is recommended that you use the TAG attribute, so you can identify in the program the kind of `COMBOBOX` Form Item to be initialized.

If neither ITEMS nor INITIALIZER attributes are specified, the form compiler automatically fills the list of items with the values of the INCLUDE attribute, when specified. However, the item list will not automatically be populated with include range values (i.e. values defined using the TO keyword). The INCLUDE attribute can be specified directly in the form or indirectly in the schema files.

During an INPUT, a `COMBOBOX` field value can only be one of the values specified in the ITEMS attribute. During an CONSTRUCT, a `COMBOBOX` field gets an additional 'empty' item (even if the field is NOT NULL), to let the user clear the search condition.

If one of the items is explicitly defined with `NULL` and the NOT NULL attribute is omitted; In `INPUT`, selecting the corresponding combobox list item sets the field value to null. In `CONSTRUCT`, selecting the list item corresponding to null will be equivalent to the = query operator, which will generate a "*colname is null*" SQL condition.

By default, during a CONSTRUCT, a `COMBOBOX` is not editable (you are forced to set one of the values of the list as defined by the ITEMS attribute, or set the 'empty' item). The QUERYEDITABLE attribute can be used to force the `COMBOBOX` to be editable during a CONSTRUCT instruction. This makes sense if the display values match the real values, in order to allow the user to enter a search criterion such as "`A*`".

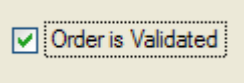
When using an HBox tag combined with the SAMPLE attribute, it is possible to specify the exact width of a `COMBOBOX`. By default, the real width is computed as *width=nbchars-2* when *nbchars*>2. See BUTTONEDIT for more details.

See also Form Field.

CHECKBOX Item Type

Purpose:

The `CHECKBOX` item type defines a boolean checkbox field.



Syntax:

```
CHECKBOX item-tag = field-name [ , attribute-list ] ;
```

Notes:

1. *item-tag* is an identifier that defines the name of the item tag.
2. *field-name* identifies the screen record field. See Field Definition for more details.
3. *attribute-list* defines the aspect and behavior of the Form Item.

Attributes:

COLOR, COLOR WHERE, COMMENT, DEFAULT, FONTPITCH, HIDDEN, INCLUDE, KEY, NOT NULL, NOENTRY, REQUIRED, SIZEPOLICY, STYLE, TAG, TABINDEX, TEXT, VALUECHECKED, VALUEUNCHECKED.

Table Column only: UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, TITLE.

Example:

```
01 CHECKBOX f001 = customer.active, REQUIRED, TEXT="Active",
VALUECHECKED="Y", VALUEUNCHECKED="N";
```

Usage:

The `CHECKBOX` item type defines a boolean entry with a box and a text label.

The TEXT attribute defines the label to be displayed near the check box.

The box shows a checkmark when the form field contains the value defined in the VALUECHECKED attribute (for example: `"Y"`), and shows no checkmark if the field value is equal to the value defined by the VALUEUNCHECKED attribute (for example: `"N"`). If you do not specify the VALUECHECKED or VALUEUNCHECKED attributes, they respectively default to `TRUE` (integer 1) and `FALSE` (integer 0).

By default, during an INPUT, a CHECKBOX field can have three states:

- Grayed (NULL value)
- Checked (VALUECHECKED value)
- Unchecked (VALUEUNCHECKED value)

If the field is declared as NOT NULL, the initial state can be grayed if the default value is NULL; once the user has changed the state of the CHECKBOX field, it switches only between Checked and Unchecked states.

During an CONSTRUCT, a CHECKBOX field always has three possible states (even if the field is NOT NULL), to let the user clear the search condition:

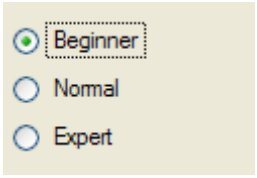
- Grayed (No search condition)
- Checked (Condition column = VALUECHECKED value)
- Unchecked (Condition column = VALUEUNCHECKED value)

See also Form Field.

RADIOGROUP Item Type

Purpose:

The RADIOGROUP item type defines a set of radio buttons.



Syntax:

```
RADIOGROUP item-tag = field-name [ , attribute-list ] ;
```

Notes:

1. *item-tag* is an identifier that defines the name of the item tag.
2. *field-name* identifies the screen record field. See Field Definition for more details.
3. *attribute-list* defines the aspect and behavior of the Form Item.

Attributes:

COLOR, COLOR WHERE, COMMENT, DEFAULT, FONTPITCH, HIDDEN, INCLUDE, ITEMS, KEY, NOT NULL, NOENTRY, ORIENTATION, REQUIRED, SIZEPOLICY, STYLE, TAG, TABINDEX.

Table Column only: UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, TITLE.

Example:

```
01 RADIOGROUP f001 = player.level,
ITEMS=((1,"Beginner"),(2,"Normal"),(3,"Expert"));
```

Usage:

This item type defines a set of radio buttons where each button is associated with a value defined in the ITEMS attribute.

The text associated with each value will be used as the label of the corresponding radio button, for example:

```
((1,"Beginner"),(2,"Normal"),(3,"Expert"))
```

During an INPUT, a `RADIOGROUP` field value can only be one of the values specified in the ITEMS attribute. During an CONSTRUCT, a `RADIOGROUP` field allows to uncheck all items (even if the field is NOT NULL), to let the user clear the search condition.

If one of the items is explicitly defined with `NULL` and the NOT NULL attribute is omitted; In `INPUT`, selecting the corresponding radio button sets the field value to null. In `CONSTRUCT`, selecting the radio button corresponding to null will be equivalent to the = query operator, which will generate a "*colname is null*" SQL condition.

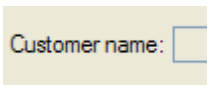
Use the ORIENTATION attribute to define if the radio group must be displayed vertically or horizontally.

See also Form Field.

LABEL Item Type

Purpose:

The `LABEL` item type defines a simple text area to display a read-only value.



Syntax 1: Defining a Form Field Label

```
LABEL item-tag = field-name [ , attribute-list ] ;
```

Syntax 2: Defining a Static Label

```
LABEL item-tag : item-name [ , attribute-list ] ;
```

Notes:

1. *item-tag* is an identifier that defines the name of the item tag.
2. *field-name* identifies the screen record field. See Field Definition for more details.
3. *item-name* identifies the form element of a static label.
4. *attribute-list* defines the aspect and behavior of the Form Item.

Attributes:

COLOR, COLOR WHERE, COMMENT, FONTPITCH, HIDDEN, JUSTIFY, REVERSE, SIZEPOLICY, STYLE, TAG.

Form Field Label only: FORMAT, SAMPLE.

Static Label only: TEXT.

Table Column only: UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, TITLE.

Example:

```
01 LABEL f001 = vehicle.description; -- This is a form field label
02 LABEL lab1 : label1, TEXT="Hello"; -- This is a static label
```

Usage:

This item type can be used to define a read-only text area as a form field or as a static label.

Form Field Label

This type of label item must be used to display values that change often during program execution, like database information. The text of the label is defined by the value of the corresponding form field. The text can be changed from the BDL program by using the DISPLAY TO instruction to set the value of the field, or within a list by using a DISPLAY ARRAY. This kind of Form Item does not allow data entry; it is only used to display values.

See also Form Field for more details.

Static Label

This type of label item must be used to display text that does not change often, like field descriptions. The text of the label is defined by the TEXT attribute; the item is not a form field. The text can be changed from the BDL program by using the API provided to manipulate the user interface (see Dynamic User Interface for more details). It is not

possible to change the text with a DISPLAY TO instruction. This kind of item is not affected by instructions such as CLEAR FORM. Static labels display only character text values, and therefore do not follow any justification rule as form field labels.

IMAGE Item Type

Purpose:

The `IMAGE` item type defines an area that can display an image from a pixel-map file.

Syntax 1: Defining a Form Field Image

```
IMAGE item-tag = field-name [ , attribute-list ] ;
```

Syntax 2: Defining a Static Image

```
IMAGE item-tag : item-name [ , attribute-list ] ;
```

Notes:

1. *item-tag* is an identifier that defines the name of the item tag.
2. *field-name* identifies the screen record field. See Field Definition for more details.
3. *item-name* identifies the form element of a static image.
4. *attribute-list* defines the aspect and behavior of the Form Item.

Attributes:

COLOR, COLOR WHERE, COMMENT, AUTOSCALE, HIDDEN, SIZEPOLICY, WIDTH, HEIGHT, STYLE, STRETCH, TAG.

Static Image only: IMAGE

Table Column only: UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, TITLE.

Example:

```
01 IMAGE f001 = cars.picture, HEIGHT=300 PIXELS, WIDTH=400 PIXELS,
STRETCH=BOTH;
02 IMAGE img1 : logo, IMAGE="fourjs.gif", STRETCH=BOTH;
```

Usage:

This item type defines an area where a picture file can be displayed, as a form field or as a static image.

The default picture size is defined by the placement of the *item-tag* inside the layout, but you can overwrite the default size by using the WIDTH and HEIGHT attributes. If these

are not used, the size of the image defaults to the relative width and height defined by the *item-tag* in the form layout section.

The STRETCH attribute can be used to define how the image size must change when the parent container is resized. Values can be `NONE`, `X`, `Y` or `BOTH`. Default value is `NONE` for `IMAGE` fields.

The AUTOSCALE boolean attribute can be set to indicate if the image must be scaled to the available space defined by the width and height of the Form Item.

If SIZEPOLICY attribute is set to `FIXED`, the size of the widget will be defined by the form specification file. Then, if AUTOSCALE is not set, scrollbars may appear if the picture is greater than the widget. When `INITIAL`, the size of the widget will be define by the first picture. When `DYNAMIC`, changing the picture may change the size of the widget.

Widget Size	**Picture Size**	SIZEPOLICY	AUTOSCALE
Size of Form Specification File	Size of Widget (image may shrink or grow)	`INITIAL`, `FIXED`, `DYNAMIC`	`TRUE` (Attribute is set)
Size of Form Specification File	Original Size (Scrollbars may appear)	`FIXED`	`FALSE` (Attribute is not set)
Size of Picture (widget can grow)	Original Size	`INITIAL`, `DYNAMIC`	`FALSE` (Attribute is not set)

Form Field Image

This type of image item must be used to display values that change often during program execution, like database information. The picture is defined in a file specified with an URL (Uniform Resource Locator) as the value of the corresponding field. The picture can be changed from the BDL program by using the DISPLAY TO instruction to set the value of the field.

See also Form Field.

Static Image

This type of image item must be used to display text that does not change often, such as background pictures or logos. The source file of the image is defined by the IMAGE attribute; the item is not a form field. The image file can be changed from the BDL program by using the API provided to manipulate the user interface (see Dynamic User Interface for more details). It is not possible to change the image with a DISPLAY TO instruction. This kind of item is not affected by instructions such as CLEAR FORM.

PROGRESSBAR Item Type

Purpose:

The PROGRESSBAR item type defines a horizontal bar with a progress indicator.

Syntax:

```
PROGRESSBAR item-tag = field-name [ , attribute-list ] ;
```

Notes:

1. *item-tag* is an identifier that defines the name of the item tag.
2. *field-name* identifies the screen record field. See Field Definition for more details.
3. *attribute-list* defines the aspect and behavior of the Form Item.

Attributes:

COLOR, COLOR WHERE, COMMENT, HIDDEN, VALUEMIN, VALUEMAX, SIZEPOLICY, STYLE, TAG.

Table Column only: UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, TITLE.

Example:

```
01 PROGRESSBAR f001 = workstate.position, VALUEMIN=-100, VALUEMAX=+100;
```

Usage:

This item type can be used to show progress information.

The position of the progress bar is defined by the value of the corresponding form field. The value can be changed from the BDL program by using the DISPLAY TO instruction to set the value of the field. This kind of Form Item does not allow data entry; it is only used to display integer values.

The VALUEMIN and VALUEMAX attributes define respectively the lower and upper integer limit of the progress information. Any value outside this range will not be displayed. Default values are `VALUEMIN=0` and `VALUEMAX=100`.

See also Form Field.

SLIDER Item Type

Purpose:

The SLIDER item type defines a horizontal or vertical slider.

Syntax:

```
SLIDER item-tag = field-name [ , attribute-list ] ;
```

Notes:

1. *item-tag* is an identifier that defines the name of the item tag.
2. *field-name* identifies the screen record field. See Field Definition for more details.
3. *attribute-list* defines the aspect and behavior of the Form Item.

Attributes:

COLOR, COLOR WHERE, COMMENT, DEFAULT, HIDDEN, ORIENTATION, SIZEPOLICY, STEP, STYLE, TABINDEX, TAG, VALUEMIN, VALUEMAX.

Table Column only: UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, TITLE.

Example:

```
01 SLIDER f001 = workstate.duration, VALUEMIN=0, VALUEMAX=5, STEP=1;
```

Usage:

This item type defines a classic widget for controlling a bounded value. It lets the user move a slider along a horizontal or vertical groove and translates the slider's position into a value within the legal range.

The VALUEMIN and VALUEMAX attributes define respectively the lower and upper integer limit of the slider information. Any value outside this range will not be displayed; the step between two marks is defined by the STEP attribute. The ORIENTATION attribute defines whether the SLIDER is displayed vertically or horizontally.

See also Form Field.

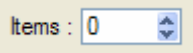
Warnings:

1. This widget is not designed for CONSTRUCT, as you can only select one value.

SPINEDIT Item Type

Purpose:

The `SPINEDIT` item type defines a spin box widget (spin button).



Syntax:

```
SPINEDIT item-tag = field-name [ , attribute-list ] ;
```

Notes:

1. *item-tag* is an identifier that defines the name of the item tag.
2. *field-name* identifies the screen record field. See Field Definition for more details.
3. *attribute-list* defines the aspect and behavior of the Form Item.

Attributes:

COLOR, COLOR WHERE, COMMENT, DEFAULT, FONTPITCH, HIDDEN, NOT NULL, NOENTRY, REQUIRED, SIZEPOLICY, STEP, STYLE, TABINDEX, TAG.

Table Column only: UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, TITLE.

Example:

```
01 SPINEDIT f001 = command.nbItems, STEP=5;
```

Usage:

This item type allows the user to choose a value either by clicking the up/down buttons to increase/decrease the value currently displayed, or by typing the value directly into the spin box.

The step between two values is defined by the STEP attribute.

See also Form Field.

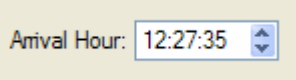
Warnings:

1. With this widget, the user can only enter an integer value.
2. This widget is not designed for CONSTRUCT, as you can only enter integer.

TIMEEDIT Item Type

Purpose:

The `TIMEEDIT` item type defines a time editor widget.



Syntax:

```
TIMEEDIT item-tag = field-name [ , attribute-list ] ;
```

Notes:

1. *item-tag* is an identifier that defines the name of the item tag.
2. *field-name* identifies the screen record field. See Field Definition for more details.
3. *attribute-list* defines the aspect and behavior of the Form Item.

Attributes:

COLOR, COLOR WHERE, COMMENT, DEFAULT, FONTPITCH, HIDDEN, NOT NULL, NOENTRY, REQUIRED, SIZEPOLICY, STYLE, TABINDEX, TAG.

Table Column only: UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, TITLE.

Example:

```
01 TIMEEDIT f001 = pakcage.arrTime;
```

Usage:

This item type allows the user to edit times by using the keyboard or the arrow keys to increase/decrease time values.

See also Form Field.

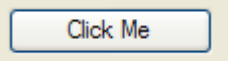
Warnings:

1. With this widget, the user can only enter a time value.
2. This widget is not designed for CONSTRUCT, as you can only enter time.

BUTTON Item Type

Purpose:

The BUTTON item type defines a push-button that can trigger an action.



Syntax:

```
BUTTON item-tag : item-name [ , attribute-list ] ;
```

Notes:

1. *item-tag* is an identifier that defines the name of the item tag.
2. *item-name* identifies the button and the corresponding action the button must be bound to.
   This name can be prefixed with the sub-dialog identifier.
3. *attribute-list* defines the aspect and behavior of the Form Item.

Attributes:

COMMENT, FONTPITCH, HIDDEN, IMAGE, SIZEPOLICY, STYLE, TAG, TEXT.

Example:

```
01 BUTTON btn1 : print, TEXT="Print Report", IMAGE="printer";
```

Usage:

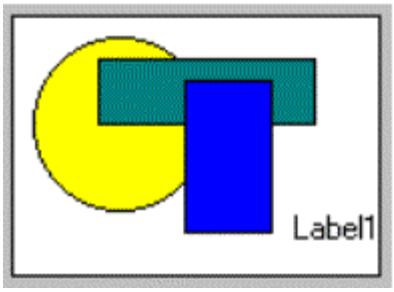
This item type defines a standard push button with a label or a picture.

In the BUTTON Form Item, the label is defined with the TEXT attribute, the picture is defined by the IMAGE attribute, and the COMMENT attribute can be used to define a help bubble for the button.

CANVAS Item Type

Purpose:

The CANVAS item type defines an area in which you can draw shapes.



Syntax:

```
CANVAS item-tag : item-name [ , attribute-list ] ;
```

Notes:

1. *item-tag* is an identifier that defines the name of the item tag.
2. *item-name* identifies the canvas in the program.
3. *attribute-list* defines the aspect and behavior of the Form Item.

Attributes:

COMMENT, HIDDEN, TAG.

Example:

```
01 CANVAS cvs1 : canvas1;
```

Usage:

See Canvas for more details.

GROUP Item Type

Purpose:

The `GROUP` item type defines attributes for a groupbox layout tag.

Syntax:

```
GROUP layout-tag : item-name [ , attribute-list ] ;
```

Notes:

1. *layout-tag* is an identifier that defines the name of the layout tag.
2. *item-name* identifies the groupbox in the program.
3. *attribute-list* defines the aspect and behavior of the Form Item.

Attributes:

COMMENT, FONTPITCH, GRIDCHILDRENINPARENT, HIDDEN, STYLE, TAG, TEXT.

Example:

```
01 GROUP g1 : group1, TEXT="Description", GRIDCHILDRENINPARENT;
```

Usage:

Use this item type to specify the attributes of a GROUP container defined with a layout tag.

SCROLLGRID Item Type

Purpose:

The `SCROLLGRID` item type defines attributes for a scrollgrid layout tag.

Syntax:

```
SCROLLGRID layout-tag : item-name [ , attribute-list ] ;
```

Notes:

1. *layout-tag* is an identifier that defines the name of the layout tag.
2. *item-name* identifies the scrollgrid in the program.
3. *attribute-list* defines the aspect and behavior of the Form Item.

Attributes:

COMMENT, FONTPITCH, GRIDCHILDRENINPARENT, HIDDEN, STYLE, TAG.

Example:

```
01 SCROLLGRID sg1 : scrollgrid1, GRIDCHILDRENINPARENT;
```

Usage:

Use this item type to specify the attributes of a SCROLLGRID container defined with a layout tag.

TABLE Item Type

Purpose:

The TABLE item type defines attributes for a table layout tag.

Syntax:

```
TABLE layout-tag : item-name [ , attribute-list ] ;
```

Notes:

1. *layout-tag* is an identifier that defines the name of the layout tag.
2. *item-name* identifies the table in the program.
3. *attribute-list* defines the aspect and behavior of the Form Item.

Attributes:

COMMENT, DOUBLECLICK, FONTPITCH, HIDDEN, STYLE, TAG, UNHIDABLECOLUMNS, UNMOVABLECOLUMNS, UNSIZABLECOLUMNS, UNSORTABLECOLUMNS, WANTFIXEDPAGESIZE, WIDTH, HEIGHT.

Example:

```
01 TABLE t1 : table1, UNSORTABLECOLUMNS;
```

Usage:

Use this item type to specify the attributes of a TABLE defined with a layout tag.

With the DOUBLECLICK attribute, you can define a particular action to be send when the user double-clicks on a row.

By default, a table allow to hide, move, resize columns and sort the list when the user clicks on a column header. The UNHIDABLECOLUMNS, UNMOVABLECOLUMNS, UNSIZABLECOLUMNS, UNSORTABLECOLUMNS attributes can be used to deny these features.

The WIDTH and HEIGHT attributes can be used to specify a initial geometry for the table.

By default, tables can be resized in height. Use the WANTFIXEDPAGESIZE attribute to deny table resizing.

INSTRUCTIONS Section

The INSTRUCTIONS section can specify screen arrays, non-default screen records and field delimiters.

Syntax:

```
INSTRUCTIONS
{ screen-record | screen-array } [;]
[...]
[ delimiters [;] ]
[ DEFAULT SAMPLE = "string" ]
[END]
```

where *screen-record* is:

```
SCREEN RECORD record-name ( field-list  )
```

where *screen-array* is:

```
SCREEN RECORD array-name [ size ] ( field-list )
```

where *field-list* is:

```
{ table.* | field [ {THROUGH|THRU} lastfield ] [,...] }
```

and where *delimiters* is:

```
DELIMITERS "AB"
```
TUI Only!

Notes:

1. The INSTRUCTIONS section is optional. If present, it must appear after the ATTRIBUTES section.
2. The END keyword is optional.
3. *record-name* is the name of an explicit screen record.
4. *array-name* is the name of an explicit screen array.
5. *size* is an integer representing the number of screen records in the screen array.
6. *field* is a field identifier as defined in the right operand of a field definition in the ATTRIBUTES section.
7. *table*.* notation instructs the form compiler to build the screen record with all fields declared in the ATTRIBUTES section for the given table.
8. *A* and *B* define the opening and closing field delimiters for character based terminals.

Warnings:

1. You must specify the *table* qualifier if the field name is not unique among the fields in the ATTRIBUTES section, or if *table* is a required alias.

2. `DELIMITERS` is provided for backward compatibility; it is only useful when using character terminals.

Example:

```
01 ...
02 INSTRUCTIONS
03 SCREEN RECORD s_items[10]
04   ( stock.*,
05     items.quantity,
06     FORMONLY.total_price )
07 DELIMITERS "[]"
08 END
```

Default sample:

The `DEFAULT SAMPLE` directive defines the default sample text for all fields. See the SAMPLE attribute for more details:

```
01 DEFAULT SAMPLE = "MMM"
```

Screen records:

A *screen record* is a group of fields that screen interaction statements of the program can reference as a single object. By establishing a correspondence between a set of screen fields (the screen record) and a set of program variables (typically a program record), you can pass values between the program and the fields of the screen record. In many applications, it is convenient to define a screen record that corresponds to a *row* of a database table.

The form compiler builds *default screen record*s that consist of all the screen fields linked to the same database table within a given form. A *default record* is automatically created for each table that is used to reference a field in the ATTRIBUTES section. The components of the default record correspond to the set of display fields that are linked to columns in that table. The name of the default screen record is the table name (or the alias, if you have declared an alias for that table in the TABLES section). For example, all the fields linked to columns of the **customer** table constitute a default screen record whose name is **customer**. If a form includes one or more FORMONLY fields, those fields constitute a default screen record called **formonly**.

Like the name of a screen field, the identifier of a screen record must be unique within the form, and it has a scope that is restricted to when its form is open. Statements can reference *record* only when the screen form that includes it is being displayed. The form compiler returns an error if *record* is the same as the name or alias of a table in the TABLES section.

In the `INSTRUCTIONS` section of a form specification file, you can declare *explicit screen records* by using the `SCREEN RECORD` keywords to declare a name for the screen record and to specify a list of fields that are members of the screen record.

Screen arrays:

A *screen array* is usually a repetitive array of fields in the screen layout, each containing identical groups of screen fields. Each *row* of a screen array is a screen record. Each *column* of a screen array consists of fields with the same field tag in the LAYOUT section of the form specification file.

You must declare screen arrays in the `INSTRUCTIONS` section with the `SCREEN RECORD` keyword, as in the declaration of a screen record, but with an additional parameter to specify the *number of screen records* in the array.

The *size* value should be the number of lines in the logical form where the set of fields that comprise each screen record is repeated within the screen array. For example, a GRID container of the LAYOUT section might represent a screen array like this:

```
01 ...
02 LAYOUT
03 GRID
04 {
05   OrdId      Date        Total Price
06  [f001      |f002       |f003       ]
07  [f001      |f002       |f003       ]
08  [f001      |f002       |f003       ]
09  [f001      |f002       |f003       ]
10 }
11 END
12 END
13 ...
```

This example requires a *size* of `[4]`. Except for the *size* parameter, the syntax for specifying the identifier and the field names of a screen array is the same as for a simple screen record.

You can reference the names of the screen array in the DISPLAY TO, DISPLAY ARRAY, INPUT and INPUT ARRAY statements of programs, but only when the screen form that includes the screen array is the current form.

Warnings:

1. You cannot define multiple screen arrays for the same TABLE definition. Only one `SCREEN RECORD` specification is allowed.

Using screen records and screen arrays in programs:

Screen records and screen arrays can display program records. If the fields in the screen record have the same sequence of data types as the columns in a database table, you can use the screen record to simplify operations that pass values between program variables and rows of the database.

Field delimiters:

You can change the delimiters used for fields from brackets ([ ]) to any other printable character, including blank spaces. The `DELIMITERS` instruction tells the form compiler what symbols to use as field delimiters when the runtime system displays the form on the screen. The opening and closing delimiter marks must be enclosed within quotation marks (").

To use only one space between fields, in the LAYOUT section, substitute a pipe symbol (|) for paired back-to- back ( ][ ) brackets that separate adjacent fields. In the `INSTRUCTIONS` section, define some symbol as both the beginning and the ending delimiter. For example, you could specify "| |" or "/ /" or ": :" or " " (blanks).

Field delimiters are not displayed when using a graphical front end, but they must be used to delimit the fields in the form specification file.

KEYS Section

The `KEYS` section can be used to define default key labels for the current form.

Syntax:

```
KEYS
key-name = [%]"label"
[...]
[END]
```

Notes:

1. The `KEYS` section is optional. If present, it must appear after the INSTRUCTIONS section.
2. The `END` keyword is optional.
3. *key-name* is the name of a key (like `F10`, `Control-z`).
4. *label* is the text to be displayed in the button corresponding to the key.

Warnings:

1. The `KEYS` section is supported for backward compatibility. See also Setting Key Labels.

Example:

```
01 ...
02 KEYS
03 F10 = "City list"
04 F11 = "State list"
05 F15 = "Validate"
06 END
```

For more details, see "Setting Key Labels".

Boolean expressions in form files

Purpose:

This section describes the syntax supported for Boolean expressions in form specification files, as in the COLOR WHERE attribute specification, for example.

Syntax:

```
[(] boolexpr {AND|OR} boolexpr [)] [...]
```

where *boolexpr* is:

```
[NOT]
{ expression
    {   =
    |   <>
    |   <=
    |   >=
    |   <
    |   >
    |   !=
    |   IS [NOT] NULL
    |   [NOT] BETWEEN expression AND expression
    }
| charexpr
    {   [NOT] MATCHES "string"
    |   [NOT] LIKE "string"
    }
}
```

Notes:

1. *expression* can be the current field tag, a character string, numeric or datetime literal.
2. *charexpr* can be the current field tag or a character string literal.
3. When a *field-tag* is used in the Boolean expression, the runtime system replaces *field-tag* at runtime with the current value in the screen field and evaluates the expression.

Compiling form files

In order for your program to work with a screen form, you must create a form specification file that conforms to the syntax described earlier in this section, and then compile the form. You must use the form compiler to do the compilation. Form

compilation requires that the database schema files already exist if the form specification file uses a SCHEMA specification.

Warnings:

1. Make sure that the database schema files of the development database correspond to the production database, otherwise the form fields defined in the compiled version of your forms will not match the table structures of the production database.

Tips:

1. As compiled form files depend on both the source files and the database schema files, it is strongly recommended that you use the MAKE utility to manage form file compilation. In the makefiles, you should define the dependence rules according to these constraints.

Using forms in programs

To work with a screen form, the application requires a *form driver,* which is a logical set of interactive statements that control the display of the form, bind form fields to program variables, and respond to actions by the user in fields.

Compiled forms must be loaded by the program with the OPEN FORM or the OPEN WINDOW WITH FORM instruction.

Warnings:

1. The DBPATH environment variable must contain the directory where the compiled form file is located, if the form file is not in the current directory.

Once a form is loaded, the program can manipulate forms to display or let the user edit data, with instructions like DISPLAY TO, INPUT, INPUT ARRAY, DISPLAY ARRAY and CONSTRUCT. Program variables are used as display and/or input buffers.

Form Specification File Attributes

Summary:

- Attributes Summary

See also: Form Specification Files.

Attributes Summary

Attribute	Description
ACCELERATOR	First accelerator key for an action default
ACCELERATOR2	Second accelerator key for an action default
ACCELERATOR3	Third accelerator key for an action default
ACCELERATOR4	Fourth accelerator key for an action default
ACTION	Action name to be sent to the program when the item is activated
AUTOSCALE	Forces the item's contents to be scaled according to the available space
AUTONEXT	Automatically gives the focus to the next field when all data is entered
BUTTONTEXTHIDDEN	Indicates that the button labels of the element must be hidden
CENTURY	Specifies expansion of 2-digit years in DATE and DATETIME fields
COLOR	Specifies the foreground color of the text displayed by a form item
COLOR WHERE	Defines a Boolean condition based on field values to set the color attribute dynamically
COMMENT	Specifies a message to display on the Comment line
DEFAULT	Assigns a default value to a field during data entry
DEFAULTVIEW	Defines if the default view must be displayed for an action
DISPLAY LIKE	Assigns attributes from the database schema files
DOUBLECLICK	Defines the action to be sent when user double-clicks on the item
DOWNSHIFT	Converts to lowercase any uppercase character data
HEIGHT	Explicitly defines the height of a form element
HIDDEN	Makes an element invisible

FONTPITCH	Defines the character font type as fixed or variable
FORMAT	Formats DECIMAL, SMALLFLOAT, FLOAT, or DATE output
GRIDCHILDRENINPARENT	Aligns children to the parent container
INCLUDE	Lists a set of acceptable values during data entry
INITIALIZER	Specifies an initialization function for the ComboBox item
INVISIBLE	Does not echo characters on the screen during data entry
IMAGE	Defines the URL of the image resource associated to the form item
ITEMS	Defines a list of values to be used by the form item
JUSTIFY	Specifies the text justification
MINHEIGHT	Defines a minimum height for the form item
MINWIDTH	Defines a minimum width for the form item
NOT NULL	Indicates that the field does not accept NULL values
NOENTRY	Prevents the user from entering data in the field
ORIENTATION	Defines the orientation of an element as vertical or horizontal
PICTURE	Imposes a data-entry format on CHAR or VARCHAR fields
PROGRAM	Invokes an external program to display TEXT or BYTE values
QUERYEDITABLE	Allows a combobox field to be editable during a CONSTRUCT
REQUIRED	Requires the user to supply a value during input instructions
SAMPLE	Provides the text to be used as a sample to compute the width
SCROLL	Allows scrolling within the field
SCROLLBARS	Defines vertical and/or horizontal scrollbars for the form item
SIZEPOLICY	Indicates sizing hint to display form elements.
SPACING	Indicates spacing hint to display form elements
SPLITTER	Indicates that the container must use a splitter
STEP	Defines how much the value increases or decreases with a single click
STRETCH	Defines how the widget must resize according to the parent container
STYLE	Defines a presentation style for the form element
TAG	Defines a string identifier for the form item

TABINDEX	Defines the tab order for the form item
TEXT	Defines the label to be associated with the form item
TITLE	Defines the title to be associated with the form item
UPSHIFT	Converts to uppercase any lowercase character data
UNHIDABLE	Indicates that the column cannot be hidden
UNHIDABLECOLUMNS	The table does not allow columns to be hidden
UNMOVABLE	Indicates that the column cannot be moved
UNMOVABLECOLUMNS	Prevents the user from changing the order of the columns
UNSIZABLE	Indicates that the column cannot be resized
UNSIZABLECOLUMNS	The table does not allow columns to be resized
UNSORTABLE	Indicates that the column cannot be used for sorting
UNSORTABLECOLUMNS	The table does not allow rows to be sorted
VALIDATE	Defines the data validation mode for a given action
VALIDATE LIKE	Validates data entry with definitions from the database schema files
VALUEMIN	Defines the lower limit for widgets (such as progressbars)
VALUEMAX	Defines the upper limit for widgets (such as progressbars)
VALUECHECKED	Defines the value to be associated with a checked checkbox
VALUEUNCHECKED	Defines the value to be associated with an unchecked checkbox
VERIFY	Requires that data be entered twice when the database is modified
VERSION	Defines a user version string for an element
WANTTABS	Forces the field to consume TAB keys
WANTNORETURNS	Forces the field to reject RETURN keys
WANTFIXEDPAGESIZE	Forces the table to have a fixed height when the parent window is resized
WIDTH	Explicitly defines the width of a form element
WINDOWSTYLE	Specifies the style to be used by the parent window
WORDWRAP	Invokes a multiple-line editor in multiple-segment fields
CLASS	Specifies the behavior of a field defined with the WIDGET attribute
CONFIG	Specifies the parameters for the definition of a widget (only used with WIDGET attribute)

KEY	Defines the label of a key when the field gets the focus
OPTIONS	Specifies widget definition options
REVERSE	Causes values in the field to be displayed in reverse video
WIDGET	Defines the type of widget to be used for presentation

---

ACCELERATOR Attribute

Purpose:

The ACCELERATOR attribute defines the primary accelerator key of an action default item.

Syntax:

```
ACCELERATOR = [CONTROL-][SHIFT-][ALT-]key
```

Notes:

1. *key* defines the accelerator key as described in Accelerators

---

ACCELERATOR2 Attribute

Purpose:

The ACCELERATOR2 attribute defines the secondary accelerator key of an action default item.

Syntax:

```
ACCELERATOR2 = [CONTROL-][SHIFT-][ALT-]key
```

Notes:

1. *key* defines the accelerator key as described in Accelerators

---

ACCELERATOR3 Attribute

Purpose:

The ACCELERATOR3 attribute defines the third accelerator key of an action default item.

Syntax:

```
ACCELERATOR3 = [CONTROL-][SHIFT-][ALT-]key
```

Notes:

1. *key* defines the accelerator key as described in Accelerators

---

ACCELERATOR4 Attribute

Purpose:

The ACCELERATOR4 attribute defines the fourth accelerator key of an action default item.

Syntax:

```
ACCELERATOR4 = [CONTROL-][SHIFT-][ALT-]key
```

Notes:

1. *key* defines the accelerator key as described in Accelerators

---

ACTION Attribute

Purpose:

The ACTION attribute defines the name of the action to be sent to the program when the user activates the form item.

Syntax:

```
ACTION = action-name
```

Notes:

1. *action-name* is an identifier that defines the name of the action to be sent.

Tips:

1. Try to use abstract action names instead of key identifiers.

Example:

```
01 BUTTONEDIT f001 = customer.state, ACTION = print;
```

AUTOSCALE Attribute

Purpose:

The `AUTOSCALE` attribute causes the form element contents to automatically scale to the size given to the item.

Syntax:

```
AUTOSCALE
```

Notes:

1. For images, this attribute forces the image to be stretched to fit in the area reserved for the image.

AUTONEXT Attribute

Purpose:

The `AUTONEXT` attribute causes the cursor to automatically advance during input to the next field when the current field is full.

Syntax:

```
AUTONEXT
```

Notes:

1. If data values entered in the field do not meet the requirements of other field attributes like `INCLUDE` or `PICTURE`, the cursor does *not* automatically move to the next field but remains in the current field, and an error message displays.

Tips:

1. `AUTONEXT` is particularly useful with character fields in which the input data is of a standard length, such as numeric postal codes or the abbreviations in the **state** table. It is also useful if a character field has a length of 1 because only one keystroke is required to enter data and move to the next field.

CENTURY Attribute

Purpose:

The `CENTURY` attribute specifies how to expand abbreviated one- and two-digit *year* specifications in a DATE and DATETIME field. Expansion is based on this setting (and on the year value from the system clock at runtime).

Syntax:

```
CENTURY = { "R" | "C" | "F" | "P" }
```

Notes:

1. The `CENTURY` attribute can specify any of four algorithms to expand abbreviated years into four-digit year values that end with the same digits (or digit) that the user has entered.
2. `CENTURY` supports the same settings as the DBCENTURY environment variable, but with a scope that is restricted to a single field.
3. If the `CENTURY` and DBCENTURY settings are different, `CENTURY` takes precedence.

Warnings:

1. Unlike DBCENTURY, the `CENTURY` attribute is not case sensitive. However, we recommend that you use uppercase letters in the attribute.

CLASS Attribute

Purpose:

The `CLASS` attribute is used to define the behavior of a field.

Syntax:

```
CLASS = "identifier"
```

Notes:

1. *identifier* is a predefined keyword defining the class of the field.

Supported field classes:

Class	Description
KEY	Field is used to trigger a keystroke instead of being a normal input field. Only supported with special widgets such as buttons, checkboxes and radiobuttons.
PASSWORD	Field input is masked by replacing normal character echo by stars "*".

Warnings:

1. The attribute is supported for backward compatibility.

COLOR Attribute

Purpose:

The COLOR attribute defines the foreground color of the text displayed by a form element.

Syntax:

```
COLOR = color-name
```

Notes:

1. *color-name* can be: BLACK, BLUE, CYAN, GREEN, MAGENTA, RED, WHITE, and YELLOW.
2. For backward compatibility, *color-name* can be combined with an intensity keyword: REVERSE, LEFT, BLINK, and UNDERLINE.

Example:

```
01 EDIT f001 = customer.name, COLOR = RED;
```

COLOR WHERE Attribute

Purpose:

The `COLOR WHERE` attribute defines a condition to set the foreground color dynamically.

Syntax:

```
COLOR = color-name [...] WHERE boolexpr
```

Notes:

1. *color-name* can be: `BLACK`, `BLUE`, `CYAN`, `GREEN`, `MAGENTA`, `RED`, `WHITE`, and `YELLOW`.
2. *color-name* can also be an intensity keyword: `REVERSE`, `LEFT`, `BLINK`, and `UNDERLINE`.
3. *boolexpr* defines a Boolean expression with a restricted syntax.
4. The Boolean expression is automatically evaluated at runtime to check when the color attribute must be set.

Warnings:

1. The condition in `COLOR WHERE` can only reference the field for which the attribute is set.
2. The `COLOR WHERE` attribute may not be supported in all situations; it is not supported in TABLE columns.

Example:

```
01 EDIT f001 = item.price, COLOR = RED WHERE f001>100;
```

CONFIG Attribute

Purpose:

The `CONFIG` attribute is used with the WIDGET attribute to define the behavior and decoration of the field.

Syntax:

```
CONFIG = "parameter [...]"
```

Notes:

1. The `CONFIG` attribute can only be used with the WIDGET attribute. It is ignored if `WIDGET` is not used.

2. *parameter* is the value of a configuration parameter.
3. Configuration parameters are separated by blanks.
4. If a configuration parameter holds blank characters, you must use `{}` curly braces to delimit the parameter value.
5. See the WIDGET attribute for more details about configuration.

COMMENT Attribute

Purpose:

The `COMMENT` attribute defines text that can be shown when the element becomes current.

Syntax:

```
COMMENT = [%]"string"
```

Notes:

1. *string* is the text to be displayed.
2. *string* can be a localized string.
3. The screen location where the message is displayed depends on external configuration. It can be displayed in the COMMENT LINE, or in the STATUSBAR when using a graphical user interface.

Tips:

1. The most common use of the `COMMENT` attribute is to give information or instructions to the user. This is particularly appropriate when the field accepts only a limited set of values.

Warnings:

1. If the OPEN WINDOW statement specifies `COMMENT LINE OFF`, any output to the comment area is hidden even if the window displays a form that includes fields that include the `COMMENT` attribute.

Example:

```
01 EDIT f001 = customer.name, COMMENT = "The customer name";
```

DEFAULT Attribute

Purpose:

The `DEFAULT` attribute assigns a default value to a field during data entry.

Syntax:

```
DEFAULT = value
```

Notes:

1. *value* can be any literal expression supported by the form compiler.
2. *value* can be `TODAY` to specify the current system date as default.
3. *value* can be `CURRENT` to specify the current system datetime as default.
4. Default values have no effect when you execute the INPUT statement using the `WITHOUT DEFAULTS` option. In this case, the runtime system displays the values in the program variables list on the screen. The situation is the same for the INPUT ARRAY statement, except that the default values are displayed when the user inserts a new row.
5. If the field is FORMONLY, you must also specify a data type when you assign the `DEFAULT` attribute to a field.
6. If both the `DEFAULT` attribute and the REQUIRED attribute are assigned to the same field, the `REQUIRED` attribute is ignored.
7. If you do not use the `WITHOUT NULL INPUT` option in the DATABASE section, all fields default to null values unless you have specified a `DEFAULT` attribute.

Warnings:

1. DATETIME and INTERVAL literals are not supported.

Example:

```
01 EDIT f001 = order.orderdate, DEFAULT = TODAY;
```

DEFAULTVIEW Attribute

Purpose:

`DEFAULTVIEW` is an Action Default attribute defining whether the default view (button) must be displayed for an action.

Syntax:

```
DEFAULTVIEW = [ AUTO | YES | NO ]
```

Notes:

1. `AUTO` means that the view must be displayed if no explicit action view is used for that action. This is the default.
2. `YES` indicates that a default action view must always be displayed for this action.
3. `NO` indicates that no default action view must be displayed for this action.

DISPLAY LIKE Attribute

Purpose:

The `DISPLAY LIKE` attribute takes column attributes defined in the database schema files and applies them to a field.

Syntax:

```
DISPLAY LIKE [table.]column
```

Notes:

1. *table* is the optional table name to qualify the column.
2. *column* is the name of the column to be used to retrieve display attributes.
3. Specifying this attribute is equivalent to listing all the attributes that are assigned to *table.column* in the database schema file generated from the **syscolatt** table.
4. Display attributes are automatically taken from the schema file if the field is linked to *table.column* in the field name specification.

Warnings:

1. The `DISPLAY LIKE` clause is evaluated at compile time, not at runtime. If the database schema file changes, you might need to recompile a program that uses the `LIKE` clause. Even if all of the fields in the form are `FORMONLY`, this attribute requires the form compiler to access the database schema file that contains the description of *table*.

Example:

```
01 EDIT f001 = FORMONLY.fullname, DISPLAY LIKE customer.custname;
```

HIDDEN Attribute

Purpose:

The `HIDDEN` attribute indicates that the element should not be displayed.

Syntax:

```
HIDDEN [ = USER ]
```

Notes:

1. `HIDDEN` sets the underlying item attribute to 1.
2. `HIDDEN=USER` sets the underlying item attribute to 2.

Warnings:

1. When you set a hidden attribute for a form field, the model node gets the hidden attribute, not the view node.

Usage:

By default, all elements are visible. You can use the `HIDDEN` attribute to hide an element, such as a form field or a groupbox. The runtime system handles hidden form fields. If you write an INPUT statement using a hidden field, the field is ignored (as if it was declared as NOENTRY). Programs may change the visibility of form fields dynamically with the ui.Form built-in class.

When you use the `HIDDEN` keyword only, the underlying item attribute is set to 1. The value 1 indicates that the element is hidden to the user without the possibility of showing the element, for example with the context menu of table headers. In this hidden mode, the UNHIDABLE attribute is ignored by the front end.

When you use `HIDDEN=USER`, the underlying item attribute is set to 2. The value 2 indicates that the element is hidden by default, but the user can show/hide the element as needed. For example, the user can change a hidden column back to visible.

Example:

```
01 EDIT f001 = FORMONLY.field1, HIDDEN;
02 EDIT col1 = FORMONLY.column1, HIDDEN=USER;
```

HEIGHT Attribute

Purpose:

The `HEIGHT` attribute defines an explicit height for a form element.

Syntax:

```
HEIGHT = integer [CHARACTERS|LINES|POINTS|PIXELS]
```

Notes:

1. *integer* defines the height of the element.

Usage:

By default, the size of an element is defined in characters and automatically computed by the form compiler according to the size of the form element in the layout.

For items like images, the default height is defined by the number of lines of the item tag (as a vertical character height). You can overwrite this default by specifying the `HEIGHT` attribute. You typically give a number of pixels.

For tables, the default height is defined by the number of lines used in the table layout. You can overwrite this default by specifying the `HEIGHT` attribute.

If you don't specify any unit, the size unit defaults to `CHARACTERS`, which defines the number of grid cells.

See also: WIDTH.

Example:

```
01 IMAGE img1 : image1, WIDTH = 200 PIXELS, HEIGHT = 120 PIXELS;
```

BUTTONTEXTHIDDEN Attribute

Purpose:

The `BUTTONTEXTHIDDEN` attribute indicates that the labels of the buttons of this element should not be displayed.

Syntax:

```
BUTTONTEXTHIDDEN
```

Usage:

Use in a TOOLBAR definition to hide the labels of buttons.

DOUBLECLICK Attribute

Purpose:

The `DOUBLECLICK` attribute defines the action to be sent when the user double-clicks on a TABLE row.

Syntax:

```
DOUBLECLICK = action-name
```

Usage:

This attribute is typically used in a TABLE container, to define the action to be sent when the user double-clicks on a row. By default, if the TABLE is driven by a DISPLAY ARRAY, a double-click fires the *accept* action. When using an INPUT ARRAY, double-click selects the whole text if the current widget is editable. If `DOUBLECLICK` is defined when using an `INPUT ARRAY`, the action can only be sent when the user double-clicks on a non-editable widget like a LABEL.

DOWNSHIFT Attribute

Purpose:

Assign the `DOWNSHIFT` attribute to a character field when you want the runtime system to convert uppercase letters entered by the user to lowercase letters, both on the screen and in the corresponding program variable.

Syntax:

```
DOWNSHIFT
```

Notes:

1. Because uppercase and lowercase letters have different values, storing character strings in one or the other format can simplify sorting and querying a database.
2. Characters entered by the user are converted in INPUT, INPUT ARRAY and CONSTRUCT instructions.
3. When using single byte runners, the conversion of ASCII characters >127 is controlled by the LC_CTYPE environment variable.
4. The results of conversions between uppercase and lowercase letters are based on the locale settings (LANG).

FORMAT Attribute

Purpose:

You can use the `FORMAT` attribute with numeric and date time fields to control the format of output displays.

Syntax:

```
FORMAT = "format-string"
```

Notes:

1. *format-string* is a string of characters that specifies a data display format.
2. You must enclose *format-string* within quotation marks (").
3. Use the PICTURE attribute to format data entered in the field by the user.
4. When using `FORMAT` the data is right-aligned in the field.
5. If *format-string* is smaller than the field width, you get a compile-time warning, but the form is usable.
6. When this attribute is not used, environment variable settings define the default format.

Warnings:

1. To follow abstract user interface programming and support internationalization, it is not recommended that you use this attribute.

Numeric formats:

For DECIMAL, SMALLFLOAT, and FLOAT data types, *format-string* consists of pound signs (#) that represent digits and a decimal point. For example, "###.##" produces three places to the left of the decimal point and exactly two to the right.

If the numeric value is too large to fit in the number of characters defined by the format, an overflow text is displayed (****).

If the actual number displayed requires fewer characters than *format-string* specifies, numbers are right-aligned and padded on the left with blanks.

If necessary to satisfy the *format-string* specification, the number values are rounded before display.

Character	Description
*	Fills with asterisks any position that would otherwise be blank.
&	Fills with zeros any position that would otherwise be blank.
#	This does not change any blank positions in the display.
<	Causes left alignment.

, (comma)	Defines the position of the comma (not displayed if there are no digits on the left).
. (period)	Defines the position of the period (only one can be used).
-	Displays a minus sign for negative numbers.
+	Displays a plus sign for positive numbers.
$	This is the placeholder for the *front* specification of DBMONEY or DBFORMAT.
(	Displayed as left parentheses for negative numbers (accounting parentheses).
)	Displayed as right parentheses for negative numbers (accounting parentheses).

Date formats:

For DATE data types, the runtime system recognizes these symbols as special in *format-string*:

Character	Description
dd	Day of the month as a 2-digit integer.
ddd	Three-letter English-language abbreviation of the day of the week, for example, Mon, Tue.
mm	Month as a 2-digit integer.
mmm	Three-letter English-language abbreviation of the month, for example, Jan, Feb.
yy	Year, as a 2-digits integer representing the 2 trailing digits.
yyyy	Year as a 4-digit number.

The form compiler interprets any other characters as literals and displays them wherever you place them within *format-string*.

These *format-string* examples and their corresponding display formats for DATE fields display the twenty-third day of September 1999:

FORMAT attribute	Result
none	09/23/1999
`FORMAT = "mm/dd/yy"`	09/23/99
`FORMAT = "mmm dd, yyyy"`	Sep 23, 1999
`FORMAT = "yymmdd"`	990923
`FORMAT = "dd-mm-yy"`	23-09-99
`FORMAT = "(ddd.) mmm. dd, yyyy"`	(Thu.) Sep. 23, 1999

Example:

```
01 EDIT f001 = order.thedate, FORMAT = "mm/dd/yyyy";
```

FONTPITCH Attribute

Purpose:

This attribute defines the character font type as fixed or variable when the default font is used.

Syntax:

```
FONTPITCH = {FIXED|VARIABLE}
```

Notes:

1. When using `FIXED`, you force the characters to have a fixed size.
2. When using `VARIABLE`, you allow the characters to have a variable size.

Usage:

By default, most front ends use variable width character fonts, but in some cases you might need to use a fixed font.

It is recommended that you use a STYLE defining a fixed font instead of this attribute.

GRIDCHILDRENINPARENT Attribute

Purpose:

This attribute is used for a container to align its children to the parent container.

Syntax:

```
GRIDCHILDRENINPARENT
```

Usage:

By default, child elements of a container are aligned locally inside the container layout cells. With this attribute, you can force children to be aligned according to the layout cells of the parent container of the container to which you assign this attribute.

This is useful, for example, when you want to align fields across groups defined with Layout Tags inside a GRID:

```
01 LAYOUT
02   GRID
03   {
04    <G g1                              >
05     Field1 [f1                       ]
06     Field2 [f2                       ]
07     Field3 [f3                       ]
08
09    <G g2                              >
10     F4     [f4                   ]
11     F5     [f5                   ]
12
13   }
14   END
15 END
16 ATTRIBUTES
17 GROUP g1 : GRIDCHILDRENINPARENT;
18 GROUP g2 : GRIDCHILDRENINPARENT;
19 EDIT f1 = FORMONLY.field1;
20 EDIT f2 = FORMONLY.field2;
21 EDIT f3 = FORMONLY.field3;
22 EDIT f4 = FORMONLY.field4;
23 EDIT f5 = FORMONLY.field5;
24 END
```

INCLUDE Attribute

Purpose:

The `INCLUDE` attribute specifies acceptable values for a field and causes the runtime system to check the data before accepting an input value.

Syntax:

```
INCLUDE = ( { NULL | literal [ TO literal ] } [,...] )
```

Notes:

1. *literal* can be any literal expression supported by the form compiler.
2. If the field is FORMONLY, you must also specify a data type when you assign the `INCLUDE` attribute to a field.

Warnings:

1. DATETIME and INTERVAL literals are not supported.

Example:

```
01 EDIT f001 = compute.rate, INCLUDE = ( 1 TO 100, 200, NULL);
02 EDIT f002 = customer.state, INCLUDE = ( "AL" TO "GA", "IA" TO "WY"
);
```

INVISIBLE Attribute

Purpose:

The `INVISIBLE` attribute prevents user-entered data from being echoed on the screen during an interactive statement.

Syntax:

```
INVISIBLE
```

Notes:

1. Characters that the user enters in a field with this attribute are not displayed during data entry. Depending on the front end type, the typed characters are displayed using the blank, star, underscore or dot characters.

Warnings:

1. This attribute does *not* prevent display instructions like DISPLAY, DISPLAY ARRAY from explicitly displaying data in the field.

IMAGE Attribute

Purpose:

The `IMAGE` attribute defines the image file to be associated with the form item.

Syntax:

```
IMAGE = "name"
```

Notes:

1. *name* can be a simple file name, a path, or URL to an image server.

Usage:

This attribute is used to define the image file containing the icon to be displayed in a button or the picture of an image form item.

The *name* can be a simple file name, a complete or relative path, or an URL (Uniform Resource Locator) path to an image server.

While you can give a complete path with the file extension, it is recommended that you use a simple name or a relative path without the file extension, so that the system can automatically search for the image file using the specific extension (bmp, gif, ...).

If the image specification is a simple file name, the file is first sought in the pictures directory on the client workstation. If the file is not found, the front-end automatically sends an image request to the runtime system, in order to search for an image on the application server. The runtime system searches for server-side images by using the FGLIMAGEPATH environment variable.

To simplify deployment, you should use server-side images to centralize all images on the application server.

Example:

```
01 BUTTONEDIT f001 = FORMONLY.field01, IMAGE = "zoom";
```

KEY Attribute

Purpose:

The `KEY` attribute is used to define the labels of keys when the field is made current.

Syntax:

```
KEY keyname = [%]"label"
```

Notes:

1. *keyname* is the name of a key (like `F10`, `"Control-z"`).
2. Note that the *keyname* has to be specified in quotes if you want to use Control / Shift / Alt key modifiers.
3. *label* is the text to be displayed in the button corresponding to the key.
4. See also the KEYS section to define key labels for the whole form.

Warning: This attributes is supported for backward compatibility.

Example:

```
01 EDIT f001 = customer.city, KEY F10 = "City list";
02 EDIT f002 = customer.state, KEY "Control-z" = "Open Zoom";
```

MINHEIGHT Attribute

Purpose:

The MINHEIGHT attribute defines the minimum height of a form.

Syntax:

```
MINHEIGHT = integer
```

Notes:

1. *integer* defines the minimum height of the element, as a number of grid cells.

Usage:

The MINHEIGHT attribute is used to define a minimum height of the form/window. It must be specified in the attributes of the LAYOUT section.

The unit defaults to a number of grid cells. This is the equivalent of the CHARACTERS in the HEIGHT attribute specification.

See also: MINWIDTH.

Example:

```
01 LAYOUT ( MINWIDTH=60, MINHEIGHT=50 )
02 GRID
03 ...
```

MINWIDTH Attribute

Purpose:

The MINWIDTH attribute defines the minimum width of a form.

Syntax:

```
MINWIDTH = integer
```

Notes:

1. *integer* defines the minimum width of the element, as a number of grid cells.

Usage:

The `MINWIDTH` attribute is used to define a minimum width of the form/window. It must be specified in the attributes of the LAYOUT section.

The unit defaults to a number of grid cells. This is the equivalent of the `CHARACTERS` in the WIDTH attribute specification.

See also: MINHEIGHT.

Example:

```
01 LAYOUT ( MINWIDTH=60, MINHEIGHT=50 )
02 GRID
03 ...
```

NOT NULL Attribute

Purpose:

The `NOT NULL` attribute sets that the field does not accept NULL values.

Syntax:

```
NOT NULL
```

Notes:

1. `NOT NULL` keywords can also be used in the type definition of FORMONLY fields.

Usage:

This attribute requires that the field contains a value. If the field contains a default value, the `NOT NULL` attribute satisfied. To insist on data entry from the user, combine `NOT NULL` with the REQUIRED attribute in the field definition, or make sure the corresponding column is defined as REQUIRED in the database schema file.

See also REQUIRED attribute.

Example:

```
01 EDIT f001 = customer.city, NOT NULL;
```

NOENTRY Attribute

Purpose:

The NOENTRY attribute prevents data entry in the field during an INPUT or INPUT ARRAY statement.

Syntax:

```
NOENTRY
```

Notes:

1. The NOENTRY attribute does *not* prevent data entry into a field during a CONSTRUCT statement.

Example:

```
01 EDIT f001 = order.totamount, NOENTRY;
```

ORIENTATION Attribute

Purpose:

The ORIENTATION attribute defines whether an element displays vertically or horizontally.

Syntax:

```
ORIENTATION = { VERTICAL | HORIZONTAL }
```

Example:

```
01 RADIOGROUP f001 = customer.status, ORIENTATION=HORIZONTAL;
```

PICTURE Attribute

Purpose:

The `PICTURE` attribute specifies a character pattern for data entry in a text field, and prevents entry of values that conflict with the specified pattern.

Syntax:

```
PICTURE = "format-string"
```

Notes:

1. *format-string* defines the data input pattern of the field.
2. *format-string* can be any combination of characters, where the characters "A", "#" and "X" have a special meaning.
3. The character "A" specifies **any letter** (alpha-numeric) character at a given position.
4. The character "#" specifies **any digit** character at a given position.
5. The character "X" specifies **any character** at a given position.
6. Any character different from "A", "X" and "#" is treated as a literal. Such characters automatically appear in the field and do not have to be entered by the user.
7. The `PICTURE` attribute does not require data entry into the entire field. It only requires that whatever characters are entered conform to *format-string*.
8. When `PICTURE` specifies input formats for DATETIME or INTERVAL fields, the form compiler does not check the syntax of *format-string*, but your form will work if the syntax is correct. Any error in *format-string*, however, such as an incorrect field separator, produces a runtime error.

Example:

```
01 EDIT f001 = carinfo.number, PICTURE = "AA####-AA(X)";
```

PROGRAM Attribute TUI Only!

Purpose:

The `PROGRAM` attribute can specify an external application program to work with screen fields of data type TEXT or BYTE.

Syntax:

```
PROGRAM = "editor"
```

Notes:

1. *editor* is the name of the program that must be used to edit the special field data.
2. You can assign the `PROGRAM` attribute to a TEXT or BYTE field to call an external program to work with the BYTE or TEXT values.
3. Users can invoke the external program by pressing the exclamation point (!) key while the screen cursor is in the field.
4. The external program then takes over control of the screen. When the user exits from the external program, the form is redisplayed with any display attributes besides `PROGRAM` in effect.
5. When no `PROGRAM` attribute is used, the DBEDIT environment variable defines the default editor.

Warning: This attribute works in TUI mode only.

QUERYEDITABLE Attribute

Purpose:

The `QUERYEDITABLE` attribute makes a combobox field editable during a CONSTRUCT statement.

Syntax:

```
QUERYEDITABLE
```

Notes:

1. The `QUERYEDITABLE` attribute is effective only during a CONSTRUCT statement..
2. This attribute is useful when the display values match the real values in the ITEMS attribute.

REQUIRED Attribute

Purpose:

The `REQUIRED` attribute forces the user to enter data in the field during an INPUT or INPUT ARRAY statement.

Syntax:

```
REQUIRED
```

Notes:

1. The `REQUIRED` attribute is effective only when the field name appears in the list of screen fields of an INPUT or INPUT ARRAY statement.
2. If both the `REQUIRED` and `DEFAULT` attributes are assigned to the same field, the runtime system assumes that the default value satisfies the `REQUIRED` attribute.

Warnings:

1. If the dialog instruction uses the `WITHOUT DEFAULTS` clause, the current value of the variable linked to the `REQUIRED` field is considered as a default value; the runtime system assumes that the field satisfies the `REQUIRED` attribute, even if the variable value is NULL.

Usage:

This attribute requires only that the user enter a printable character in the field. If the user subsequently erases the entry during the same input, the runtime system considers the `REQUIRED` attribute satisfied. To insist on a non-null entry, combine `REQUIRED` with the NOT NULL attribute in the field definition or make sure the corresponding column is defined as NOT NULL in the database schema file.

See also NOT NULL attribute.

REVERSE Attribute

Purpose:

On character terminals, the `REVERSE` attribute displays any value in the field in reverse video (dark characters in a bright field).

Syntax:

```
REVERSE
```

Notes:

1. With character based terminals, the REVERSE video escape sequences must be defined in the TERMINFO or TERMCAP databases.

SAMPLE Attribute

Purpose:

The SAMPLE attribute defines the text to be used to compute the width of a form field.

Syntax:

```
SAMPLE = "text"
```

Notes:

1. *text* is the sample string that will be used to compute the width of the field.

Warnings:

1. By default the physical width of the fields is:
   if the width is smaller than 6 chars, the pixel width of the character 'M', multiplied by the number of characters the field is designed for,
   if the width is bigger than 6 chars, the pixel width of 6 characters 'M' plus the pixel width of the character '0' , multiplied by the number of characters the field is designed for minus 6.
   The default sample looks like "MMMMMM0000"...

Usage:

By default, form fields are rendered by the client with a size determined by the current font and the number of characters used in the layout grid. The field width is computed so that the largest value can fit in the widget.

Sometimes the default computed width is too wide for the typical values displayed in the field. For example, numeric fields usually need less space as alphanumeric fields. If the values are always smaller, you can use the SAMPLE attribute to provide a hint for the front end to compute the best width for that form field.

You can define a default sample for all fields used in the form, by specifying a DEFAULT SAMPLE option in the INSTRUCTIONS section.

See also: DEFAULT SAMPLE.

Example:

```
01 EDIT cid = customer.custid, SAMPLE="0000";
02 EDIT ccode = customer.ucode, SAMPLE="MMMMMM";
03 DATEEDIT be01 = customer.created, SAMPLE="00-00-0000";
```

SCROLL Attribute

Purpose:

The `SCROLL` attribute can be used to enable horizontal scrolling in a character field.

Syntax:

```
SCROLL
```

Warnings:

1. Applies only to fields with character data input.

Usage:

By default, the maximum data input length is defined by the width of the item-tag of the field. For example, if you define an CHAR field in the form with a length of 3 characters, users can only enter a maximum of 3 characters, even if the program variable used for input is a CHAR(20).

If you want to let the user input more characters than the width of the item-tag of the field, use the `SCROLL` attribute.

See also: Field Input Length.

STRETCH Attribute

Purpose:

The `STRETCH` attribute specifies how a widget must resize when the parent container is resized.

Syntax:

```
STRETCH = { NONE | X | Y | BOTH }
```

Usage:

This attribute is typically used with form items that can be resized like IMAGE or TEXTEDIT fields. By default such form items have a fixed width and height, but in some cases you may want to force the widget to resize vertically, horizontally, or in both directions.

Example:

```
01 IMAGE i01 = FORMONLY.image01, STRETCH=BOTH;
```

STEP Attribute

Purpose:

The `STEP` attribute specifies how a value is increased or decreased in one step (by a mouse click or key up/down).

Syntax:

```
STEP = integer
```

Usage:

This attribute is typically used with form items allowing the user to change the current integer value by a mouse click like SLIDER, SPINEDIT.

Example:

```
01 SLIDER s01 = FORMONLY.slider, STEP=10;
```

TEXT Attribute

Purpose:

The `TEXT` attribute defines the label associated with a form item, such as the text of a checkbox item.

Syntax:

```
TEXT = [%]"string"
```

Notes:

1. *string* defines the label to be associated with the form item.
2. *string* can be a localized string.

Example:

```
01 CHECKBOX cb01 = FORMONLY.checkbox01, TEXT="OK", VALUECHECKED='y',
VALUEUNCHECKED='n';
```

TITLE Attribute

Purpose:

The `TITLE` attribute defines the title of a form item. Use may be restricted to form fields that make up the columns of a table container; see the documentation for the relevant form item.

Syntax:

```
TITLE = [%]"string"
```

Notes:

1. *string* defines the title to be associated with the form item.
2. *string* can be a localized string.

Example:

```
01 EDIT col1 = FORMONLY.column1, TITLE="Num";
```

VALUEMIN Attribute

Purpose:

The `VALUEMIN` attribute defines a lower limit of values displayed in widgets (such as progress bars).

Syntax:

```
VALUEMIN = integer
```

Notes:

1. *integer* is a integer literal.

Usage:

This attribute is typically used in PROGRESSBAR fields, to define the lower limit.

VALUEMAX Attribute

Purpose:

The VALUEMAX attribute defines a upper limit of values displayed in widgets (such as progress bars).

Syntax:

```
VALUEMAX = integer
```

Notes:

1. *integer* is an integer literal.

Usage:

This attribute is typically used in PROGRESSBAR fields, to define the upper limit.

VALUECHECKED Attribute

Purpose:

The VALUECHECKED attribute defines the value associated with a checkbox item when it is checked.

Syntax:

```
VALUECHECKED = value
```

Notes:

1. *value* is a numeric or string literal, or one of the following keywords: NULL, TRUE, FALSE.

Usage:

This attribute is used in conjunction with the VALUEUNCHECKED attribute to define the values corresponding to the states of a CHECKBOX. See CHECKBOX definition for more details.

Example:

```
01 CHECKBOX cb01 = FORMONLY.checkbox01, TEXT="OK", VALUECHECKED=TRUE,
VALUEUNCHECKED=FALSE;
```

VALUEUNCHECKED Attribute

Purpose:

The VALUEUNCHECKED attribute defines the value associated with a checkbox item when it is not checked.

Syntax:

```
VALUEUNCHECKED = value
```

Notes:

1. *value* is a numeric or string literal, or one of the following keywords: NULL, TRUE, FALSE.

Usage:

This attribute is used in conjunction with the VALUECHECKED attribute to define the values corresponding to the states of a CHECKBOX. See CHECKBOX definition for more details.

Example:

```
01 CHECKBOX cb01 = FORMONLY.checkbox01, TEXT="OK", VALUECHECKED="Y",
VALUEUNCHECKED="N";
```

UNSORTABLE Attribute

Purpose:

Indicates that the element cannot be selected by the user for sorting.

Syntax:

```
UNSORTABLE
```

Notes:

1. Makes sense only for a field that is used for the definition of a column in a TABLE container.

Usage:

By default, a TABLE container allows the user to sort the columns by a left-click on the column header. Use this attribute to prevent a sort on a specific column.

Example:

```
01 EDIT c01 = item.comment, UNSORTABLE;
```

UNSORTABLECOLUMNS Attribute

Purpose:

Indicates that the columns of the table cannot be selected by the user for sorting.

Syntax:

```
UNSORTABLECOLUMNS
```

Usage

Same effect as UNSORTABLE, but at the TABLE level, so that none of the table columns can be used for sort.

Example:

```
01 TABLE t1 ( UNSORTABLECOLUMNS )
```

UNSIZABLE Attribute

Purpose:

Indicates that the element cannot be resized by the user.

Syntax:

```
UNSIZABLE
```

Notes:

1. Makes sense only for a field that is used for the definition of a column in a TABLE container.

Usage:

By default, a TABLE container allows the user to resize the columns by a drag-click on the column header. Use this attribute to prevent a resize on a specific column.

Example:

```
01 EDIT c01 = item.comment, UNSIZABLE;
```

UNSIZABLECOLUMNS Attribute

Purpose:

Indicates that the columns of the table cannot be resized by the user.

Syntax:

```
UNSIZABLECOLUMNS
```

Usage

Same effect as UNSIZABLE, but at the TABLE level, to make all columns not resizable.

Example:

```
01 TABLE t1 ( UNSIZABLECOLUMNS )
```

UNHIDABLE Attribute

Purpose:

Indicates that the element cannot be hidden or shown by the user with the context menu.

Syntax:

```
UNHIDABLE
```

Notes:

1. Makes sense only for a field that is used for the definition of a column in a TABLE container.

Usage:

By default, a TABLE container allows the user to hide the columns by a right-click on the column header. Use this attribute to prevent the user from hiding a specific column.

Example:

```
01 EDIT c01 = item.comment, UNHIDABLE;
```

UNHIDABLECOLUMNS Attribute

Purpose:

Indicates that the columns of the table cannot be hidden or shown by the user with the context menu.

Syntax:

```
UNHIDABLECOLUMNS
```

Usage

Same effect as UNHIDABLE, but at the TABLE level, to make all columns not hidable.

Example:

```
01 TABLE t1 ( UNHIDABLECOLUMNS )
```

UNMOVABLE Attribute

Purpose:

The `UNMOVABLE` attribute prevents the user from moving a defined column of a table.

Syntax:

```
UNMOVABLE
```

Usage:

By default, a TABLE container allows the user to move the columns by dragging and dropping the column header. Use this attribute to prevent the user from changing the order of a specific column. Typically, `UNMOVABLE` is used on at least two columns to prevent the user from changing the order of the input on these columns.

UNMOVABLECOLUMNS Attribute

Purpose:

The `UNMOVABLECOLUMNS` attribute prevents the user from moving columns of a table.

Syntax:

```
UNMOVABLECOLUMNS
```

Usage:

By default, a TABLE container allows the user to move the columns by dragging and dropping the column header. Use this attribute to prevent the user from changing the order of columns.

UPSHIFT Attribute

Purpose:

Assign the `UPSHIFT` attribute to a character field when you want the runtime system to convert lowercase letters entered by the user to uppercase letters, both on the screen and in the corresponding program variable.

Syntax:

```
UPSHIFT
```

Notes:

1. Because uppercase and lowercase letters have different values, storing character strings in one or the other format can simplify sorting and querying a database.
2. Characters entered by the user are converted in INPUT, INPUT ARRAY and CONSTRUCT instructions.
3. With single byte runners the conversion of ASCII characters >127 is controlled by the locale settings (the LC_CTYPE environment variable).
4. The results of conversions between uppercase and lowercase letters are based on the locale settings (LANG).

Example:

```
01 EDIT f001 = FORMONLY.thetitle, UPSHIFT;
```

VALIDATE Attribute

Purpose:

`VALIDATE` is an Action Defaults attribute defining the data validation level for a given action.

Syntax:

```
VALIDATE = NO
```

Notes:

1. `NO` indicates that no data validation must occur for this action. However, current input buffer contains the text modified by the user before triggering the action.

VALIDATE LIKE Attribute

Purpose:

The `VALIDATE LIKE` attribute instructs the runtime system to validate the data entered in the field by using the validation rules defined in the database schema file for the column associated with the field.

Syntax:

```
VALIDATE LIKE [table.]column
```

Notes:

1. *table* is the optional table name to qualify the column.
2. *column* is the name of the column used to search for validation rules.
3. Specifying this attribute is equivalent to listing all the attributes that are assigned to *table.column* in the database schema file generated from the **syscolval** table.
4. Validation rules are taken automatically from the schema file if the field is linked to *table.column* in the field name specification.

Warnings:

1. The `VALIDATE LIKE` clause is evaluated at compile time, not at runtime. If the database schema file changes, you might need to recompile a program that uses the `LIKE` clause. Even if all of the fields in the form are `FORMONLY`, this attribute requires the form compiler to access the database schema file that contains the description of *table*.

Example:

```
01 EDIT f001 = FORMONLY.fullname, VALIDATE LIKE customer.custname;
```

INITIALIZER Attribute

Purpose:

The `INITIALIZER` attribute allows you to specify an initialization function that will be automatically called by the runtime system to set up the form item.

Syntax:

```
INITIALIZER = function
```

Notes:

1. *function* is an identifier defining the program function to be called.

Usage:

The initialization function must exist in the program using the form file and must be defined with a ui.ComboBox parameter.

ITEMS Attribute

Purpose:

The `ITEMS` attribute defines a list of possible values that can be used by the form item.

Syntax:

```
ITEMS = { single-value-list | double-value-list }
```

where *single-value-list* is:

```
( value [,...] )
```

where *double-value-list* is:

```
( ( value, label-value ) [,...] )
```

Notes:

1. *single-value-list* is a comma-separated list of single values.
2. *double-value-list* is a comma-separated list of (a, b) values pairs within parentheses.
3. *value* is a numeric or string literal, or one of the following keywords: `NULL`, `TRUE`, `FALSE`.
4. *label-value* is a numeric literal, a string literal, or a localized string.

Warnings:

1. It is only possible to use localized strings for item labels (i.e. not for key values).

Usage:

The list must be delimited by parentheses, and each element of the list can be a simple literal value or a pair of literal values delimited by parentheses.

The following example defines a list of simple values:

```
ITEMS = ("Paris", "London", "New York")
```

The next example defines a list of pairs:

```
ITEMS = ((1,"Paris"),(2,"London"),(3,"New York"))
```

This attribute can be used, for example, to define the list of a `COMBOBOX` form item:

```
01 COMBOBOX cb01 = FORMONLY.combobox01, ITEMS =
((1,"Paris"),(2,"London"),(3,"New York"));
```

In this case, the first value of a pair (1,2,3) defines the data values of the form field and the second value of a pair ("Paris", "London", "New York") defines the value to be displayed in the selection list.

When used in a `RADIOGROUP` form item, this attribute defines the list of radio buttons:

```
01 RADIOGROUP rg01 = FORMONLY.radiogroup01, ITEMS =
((1,"Paris"),(2,"London"),(3,"New York"));
```

In this case, the first value of a pair (1,2,3) defines the data values of the form field and the second value of a pair ("Paris", "London", "New York") defines the value to be displayed as the radio button label.

Localization

You can specify item labels with Localized Strings, but this is only possible when you specify a key and a label:

```
ITEMS = ((1,%"item1"),(2,%"item2"),(3,%"item3"))
```

Using NULL items

It is allowed to define a `NULL` value for an item (note that an empty string is equivalent to `NULL`):

```
ITEMS = ((NULL,"Enter bug status"),(1,"Open"),(2,"Resolved"))
```

In this case, the behavior of the field depends from the item type used. For more details, see field type specific notes for COMBOBOX and RADIOGROUP.

JUSTIFY Attribute

Purpose:

The `JUSTIFY` attribute defines text justification.

Syntax:

```
JUSTIFY = { LEFT | CENTER | RIGHT }
```

Usage:

Default text justification depends on the dialog type, the field data type and the FORMAT attribute. For example, a numeric field value is right aligned, while a string field is left aligned. The type of dialog also defines the default justification. In a CONSTRUCT, all input fields are left aligned, for search criteria input.

With the `JUSTIFY` attribute, you define the display text justification of a field, as `LEFT`, `CENTER` or `RIGHT`. This attribute is ignored for input; only the default text justification rule applies when a field has the focus.

The `JUSTIFY` attribute can only be used for widgets displaying a value as text, like a LABEL, EDIT or BUTTONEDIT.

Example:

```
01 LABEL t01 : TEXT="Hello!", JUSTIFY=RIGHT;
02 EDIT f01 = order.value, JUSTIFY=CENTER;
```

SCROLLBARS Attribute

Purpose:

The SCROLLBARS attribute can be used to specify scrollbars for a form item.

Syntax:

```
SCROLLBARS = { NONE | VERTICAL | HORIZONTAL | BOTH }
```

Usage:

This attribute defines scrollbars for the form item, such as a TEXTEDIT.

Example:

```
01 TEXTEDIT f001 = customer.fname, SCROLLBARS=BOTH;
```

SIZEPOLICY Attribute

Purpose:

The SIZEPOLICY attribute is a sizing directive to display form elements.

Syntax:

```
SIZEPOLICY = { INITIAL | FIXED | DYNAMIC }
```

Usage:

This attribute defines the initial size of some form elements. The default value of SIZEPOLICY is INITIAL.

When the SIZEPOLICY is FIXED, the form elements size is exactly the one defined in the Form Specification File. The width of the element is computed from the defined width and the font used.

For some elements such as COMBOBOX or RADIOGROUP, you may want the size of the widget to fit exactly to its content: When SIZEPOLICY is DYNAMIC, the width of the element grows and shrinks according to the width of the wider item.

When a form element is created from a database (for instance populating a COMBOBOX item list), the width of each element is not known when designing the form. When SIZEPOLICY is INITIAL, the width is computed to display the element correctly the first time it appears on the screen. Once it is displayed, its width is frozen. This behavior is also very useful when using Internationalization.

When SIZEPOLICY is INITIAL, the client behaves differently depending on the form element type:

- Buttons: The width defined in the form is a minimum width. If the text is bigger, the size grows.
- ComboBoxes: The width defined in the form is a minimum width. If one of the items in the value list is bigger, the size grows in order for the combobox to display the largest item fully .
- Labels, Checkboxes and Radio Groups: The width defined in the form is ignored. The fields are sized according to their text.
- Images are using AUTOSCALE and STRETCH attributes in combination of SIZEPOLICY .
- Other items (mostly Edits, or widget without items like ProgressBar) are not sensitive to this attribute

The following table shows the effect of the SIZEPOLICY attribute according to the type of form item; INITIAL corresponds to the first content, while DYNAMIC corresponds to the content at anytime:

Item Type	INITIAL	FIXED	DYNAMIC
EDIT	fixed	fixed	no effect
BUTTONEDIT	fixed	fixed	no effect
TEXTEDIT	fixed	fixed	no effect
DATEEDIT	can shrink	fixed	no effect
COMBOBOX	can grow	fixed	can grow
BUTTON	can grow	fixed	can shrink and grow
LABEL	can shrink and grow	fixed	can shrink and grow
RADIOGROUP	can shrink and grow	fixed	can shrink and grow
CHECKBOX	can shrink and grow	fixed	can shrink and grow
PROGRESSBAR	fixed	fixed	no effect
SLIDER	fixed	fixed	no effect
SPINEDIT	fixed	fixed	no effect
TIMEEDIT	fixed	fixed	no effect
IMAGE	depends on AUTOSCALE and STRETCH attributes		
CANVAS	Non applicable	Non applicable	Non applicable

Example:

```
01 COMBOBOX f001 = customer.city,
ITEMS=((1,"Paris"),(2,"Madrid"),(3,"London")), SIZEPOLICY=DYNAMIC;
```

SPACING Attribute

Purpose:

The SPACING attribute is a spacing directive to display form elements.

Syntax:

```
SPACING = { NORMAL | COMPACT }
```

Usage:

This attribute defines the global distance between two neighboring form elements. By default, forms are displayed with NORMAL spacing. In NORMAL mode, the front end displays form elements consistent with the desktop spacing, which is, for example, 6 and 10 pixels on Microsoft Windows platforms. Some overcrowded forms may need to be displayed with less space between elements, to let them fit to the screen. In this case you can use the COMPACT mode.

Example:

```
01 LAYOUT ( SPACING=COMPACT )
```

SPLITTER Attribute

Purpose:

The SPLITTER attribute forces the container to use a splitter widget between each child element.

Syntax:

```
SPLITTER
```

Usage:

This attribute indicates that the container (typically, a VBOX or HBOX) must have a splitter between each child element held by the container. If a container is defined with a splitter and if the children are stretchable (like TABLE or TEXTEDIT), users can resize the child elements inside the container.

Example:

```
01 VBOX ( SPLITTER )
```

STYLE Attribute

Purpose:

The `STYLE` attribute specifies a presentation style for a form element.

Syntax:

```
STYLE = "string"
```

Notes:

1. *string* is a user-defined style.

Usage:

This attribute specifies a presentation style to be applied to a form element. The presentation style can define decoration attributes such as a background color, a font type, and so on.

TAG Attribute

Purpose:

The `TAG` attribute can be used to identify the form item with a specific string.

Syntax:

```
TAG = "tag-string"
```

Notes:

1. *tag-string* is free text.

Usage:

This attribute is used to identify form items with a specific string. It can be queried in the program to perform specific processing.

You are free to use this attribute as you need. For example, you can define a numeric identifier for each field in the form in order to show context help, or group fields for specific input verification.

If you need to handle multiple data, you can format the text, for example, by using a pipe separator.

Example:

```
01 EDIT f001 = customer.fname, TAG = "name";
02 EDIT f002 = customer.lname, TAG = "name|optional";
```

TABINDEX Attribute

Purpose:

The `TABINDEX` attribute defines the tab order for a form item.

Syntax:

```
TABINDEX = integer
```

Notes:

1. *integer* defines the order of the item in the tab sequence.
2. If *integer* is zero, the item will be excluded from the tagging list.

Usage:

This attribute can be used to define the order in which the form items are selected as the user "tabs" from field to field when the program is using the form field order option.

It can also be used to define which field must get the focus when a folder page is selected.

By default, form items get a tab index according to the order in which they appear in the LAYOUT section.

Tip: `TABINDEX` can be set to zero in order to exclude the item from the tabbing list. The item can still get the focus with the mouse.

Example:

```
01 EDIT f001 = customer.fname, TABINDEX = 1;
02 EDIT f002 = customer.lname, TABINDEX = 2;
03 EDIT f003 = customer.comment, TABINDEX = 0; -- Exclude from tabbing
list
```

VERIFY Attribute

Purpose:

The `VERIFY` attribute requires users to enter data in the field twice to reduce the probability of erroneous data entry.

Syntax:

```
VERIFY
```

Notes:

1. Because some data is critical, this attribute supplies an additional step in data entry to ensure the integrity of your data. After the user enters a value into a `VERIFY` field and presses RETURN, the runtime system erases the field and requests reentry of the value. The user must enter exactly the same data each time, character for character: 15000 is not exactly the same as 15000.00.
2. The `VERIFY` attribute takes effect while INPUT or INPUT ARRAY statements are executing. It has no effect on CONSTRUCT statements.

VERSION Attribute

Purpose:

The `VERSION` attribute is used to specify a user version string for an element.

Syntax:

```
VERSION = { "string" | TIMESTAMP }
```

Notes:

1. *string* is a user-defined version string.

Warnings:

1. Use the `TIMESTAMP` only during development.

Usage:

This attribute specifies a version string to distinguish different versions of a form element. You can specify an explicit version string or use the `TIMESTAMP` keyword to force the form compiler to write a timestamp string into the **42f** file.

Typical usage is to specify a version of the form to indicate if the form content has changed. This attribute is used by the front-end to distinguish different form versions and to avoid reloading window/form settings into a new version of a form.

Example:

```
01 LAYOUT ( TEXT="Orders", VERSION = "1.23" )
```

OPTIONS Attribute

Purpose:

The `OPTION` attribute specifies widget options for the field.

Syntax:

```
OPTIONS = "option [...]"
```

Notes:

1. *option* can be one of: `-nolist` (to indicate that the column should appear as an independent field).

Warnings:

1. This attributes is supported for backward compatibility.

WANTTABS Attribute

Purpose:

The `WANTTABS` attribute forces a text field to insert TAB characters in the text when the user presses the TAB key.

Syntax:

```
WANTTABS
```

Usage:

By default, text fields like TEXTEDIT do not insert a TAB character in the text when the user presses the TAB key, since the TAB key is used to move to the next field. You can force the field to consume TAB keys with this attribute.

WANTNORETURNS Attribute

Purpose:

The WANTNORETURNS attribute forces a text field to reject NewLine characters when the user presses the RETURN key.

Syntax:

```
WANTNORETURNS
```

Usage:

By default, text fields like TEXTEDIT insert a NewLine (ASCII 10) character in the text when the user presses the RETURN key. As the RETURN key is used to validate the dialog, you can force the field to reject RETURN keys with this attribute.

WANTFIXEDPAGESIZE Attribute

Purpose:

The WANTFIXEDPAGESIZE attribute gives a fixed height to a TABLE container.

Syntax:

```
WANTFIXEDPAGESIZE
```

Usage:

By default, the height of a TABLE container is resizable. Use this attribute to freeze the number of rows to the number of screen lines defined by the form file.

WIDTH Attribute

Purpose:

The WIDTH attribute defines an explicit width of a form element.

Syntax:

```
WIDTH = integer [CHARACTERS|COLUMNS|POINTS|PIXELS]
```

Notes:

1. *integer* defines the width of the element.

Usage:

By default, the size of an element is defined in characters and automatically computed by the form compiler according to the size of the form element in the layout.

For items like images, the default width is defined by the number of horizontal characters used in the item tag. You can overwrite this default by specifying the `WIDTH` attribute. You typically give a number of pixels.

For tables, the default width is defined by the columns used in the table layout. You can overwrite this default by specifying the `WIDTH` attribute. You typically give a number of columns. This allows you to use tables with a large number of columns, but a small initial width.

If you don't specify any unit, the size unit defaults to `CHARACTERS`, which defines the number of grid cells.

See also: HEIGHT.

Example:

```
01 TABLE t1 ( WIDTH = 5 COLUMNS )
```

WIDGET Attribute

Purpose:

The `WIDGET` attribute specifies the type of graphical widget to be used for the field.

Syntax:

```
WIDGET = "identifier"
```

Notes:

1. *identifier* defines the type of widget, it can be one of the keywords listed in the table below.
2. The `WIDGET` attribute is used with CONFIG to parameter the field widget.

Warnings:

1. This attribute is supported for backward compatibility.

- Instead of `WIDGET="IMAGE"`, you should now use a IMAGE form item.
- Instead of `WIDGET="CANVAS"`, you should now use a CANVAS form item.
- Instead of `WIDGET="CHECK"`, you should now use a CHECKBOX form item.
- Instead of `WIDGET="COMBO"`, you should now use a COMBOBOX form item.
- Instead of `WIDGET="BMP"`, you should now use a BUTTON form item.
- Instead of `WIDGET="FIELD_BMP"`, you should now use a BUTTONEDIT form item.
- Instead of `WIDGET="RADIO"`, you should now use a RADIOGROUP form item.

2. The *identifier* widget type is case sensitive, only uppercase letters are recognized.
3. When you use the `WIDGET` attribute, the form cannot be properly displayed on character based terminals, it should only be displayed on graphical front ends.

Supported widgets:

Symbol	Effect	Other attributes
`Canvas`	The field is used as a drawing area. Field must be declared as FORMONLY field.	None.
`BUTTON`	The field is presented as a button widget with a label.	CONFIG: The unique parameter defines the key to be sent when the user clicks on the button. Button text is defined in configuration files or from the program with a DISPLAY TO instruction. For example: `CONFIG = "Control-z"`
`BMP`	The field is presented as a button with an image.	CONFIG: First parameter defines the image file to be displayed, second parameter defines the key to be sent when the user clicks on the button. For example: `CONFIG = "smiley.bmp F11"` **Important warning:** Image files are not centralized on the machine where the program is executed; image files must be present on the Workstation. See front end specific documentation for more details.
`CHECK`	The field is presented as a checkbox widget. It can be used with	CONFIG: First and second parameters define the values corresponding respectively to the state "Checked" / "Unchecked" of the check box, while the third parameter defines the label of the

	the CLASS attribute to change the behavior of the widget.	checkbox.
		For example:
		`CONFIG = "Y N Confirmation"`
		If the CLASS attribute is used with the `"KEY"` value, the first and second parameters defines the keys to be sent respectively when the checkbox is "Checked" / "Unchecked", and the third parameter defines the label of the checkbox as with normal checkbox usage.
		For example:
		`CLASS="KEY",CONFIG="F11 F12 Confirmation"`

`COMBO`	The field is presented as a combobox widget.
	
	It can be used with the CLASS attribute to change the behavior of the widget.
		INCLUDE: This attribute defines the list of acceptable values that will be displayed in the combobox list.
		For example:
		`INCLUDE = ("Paris", "London", "Madrid")`
		Important warning: The INCLUDE attribute cannot hold value range definitions, because all items must be explicitly listed to be added to the combobox list.
		The following example is not supported:
		`INCLUDE = ( 1 TO 10 )`

`FIELD_BMP`	The field is presented as a normal editbox, plus a button on the right.	CONFIG: The first parameter defines the image file to be displayed in the button; the second parameter defines the key to be sent when the user clicks on the button. For example: `CONFIG = "combo.bmp Control-z"`
`LABEL`	The field is presented as a simple label, a read-only text.	None.

`RADIO`	The field is presented as a radiogroup widget.	CONFIG: Parameter pairs define respectively the value and the label corresponding to one radio button.
		For example:
		`CONFIG = "AA First BB Second CC Third"`
		If the CLASS attribute is used with the value `"KEY"`, the first element of each pairs represents the key to be sent when the user selects a radio button.
		For example:
		`CLASS="KEY",`
		`CONFIG="F11 First F12 Second F13 Third"`

Controlling V3 widgets activation:

The following list of widgets can be enabled or disabled from programs with a DISPLAY TO instruction:

- Text buttons (`WIDGET="BUTTON"`)
- Image buttons (`WIDGET="BMP"`)
- Checkboxes of class "KEY" (`WIDGET="CHECK", CLASS="KEY"`)
- Radio buttons of class "KEY" (`WIDGET="RADIO", CLASS="KEY"`)

If you display an exclamation mark in such fields, the button is enabled, but if you display a star (*), it is disabled:

```
01 DISPLAY "*" TO button1 # disables the button
02 DISPLAY "!" TO button1 # enables the button
```

Changing the text of V3 buttons:

Text buttons (`WIDGET="BUTTON"`) can be changed from programs with the DISPLAY TO instruction:

```
01 DISPLAY "Click me" TO button1 # Sets text and enables the button
```

Changing the image of V3 buttons:

Image buttons (`WIDGET="BMP"`) can be changed from programs with the DISPLAY TO instruction:

```
01 DISPLAY "smiley.bmp" TO button1 # Sets image and enables the button
```

Warning: Image files are not centralized on the machine where the program is executed; image files must be present on the Workstation. See front end specific documentation for more details.

Changing the text of V3 labels:

The text if label fields (`WIDGET="LABEL"`) can be changed from programs with the DISPLAY TO instruction:

```
01 DISPLAY "Firstname" TO l_firstname # Sets text of the label field
```

Using V3 canvas areas:

The fields declared with the `WIDGET="Canvas"` attribute can be used by the program as drawing areas. Canvas fields must be defined in the LAYOUT section. A set of drawing functions are provided to fill CANVAS fields with graphical elements.

WINDOWSTYLE Attribute

Purpose:

The WINDOWSTYLE attribute defines the style to be used by the parent window of a form.

Syntax:

```
WINDOWSTYLE = "string"
```

Notes:

1. *string* is a user defined style.

Usage:

The WINDOWSTYLE attribute can be used to specify the style of the parent window that will hold the form. This attribute is specific to the LAYOUT element. Do not confuse with the STYLE attribute, which is used to specify decoration style of the form elements.

When a form is loaded by the OPEN WINDOW or DISPLAY FORM instructions, the runtime system automatically assigns the WINDOWSTYLE to the 'style' attribute of the parent window element.

See also: STYLE, Windows and Forms.

Example:

```
01 LAYOUT ( STYLE="BigFont", WINDOWSTYLE="dialog" )
```

WORDWRAP Attribute

Purpose:

The WORDWRAP attribute enables a multiple-line editor in TUI mode.

Syntax:

```
WORDWRAP [ { COMPRESS | NONCOMPRESS } ]
```

Usage:

In TUI mode:

- During input and display, the runtime system treats all segments that have that field tag as segments of a single field.

- The multi-line editor can *wrap* long character strings to the next line of a multiple-segment field for data entry, data editing, and data display.
- The `COMPRESS` option prevents blanks produced by the editor from being included in the program variable. `COMPRESS` is applied by default and can cause truncation to occur if the sum of intentional characters exceeds the field or column size. Because of editing blanks in the `WORDWRAP` field, the stored value might not correspond exactly to its multiple-line display.
- Specifying `NONCOMPRESS` after the `WORDWRAP` keyword causes any editor blanks to be saved when the string value is saved in a database column, in a variable, or in a file.

In GUI mode:

- The `WORDWRAP` attribute is ignored, because text input and display is managed by the text editor widget.
- The text data is NOT automatically modified by the editor by adding blanks to put words on the next line.

Warnings:

1. This attribute is provided for backward compatibility with character-based forms; you should use a TEXTEDIT form item instead in graphical forms.
2. Using `WORDWRAP` fields with character-based terminals results in quite different behavior than with graphical front ends. With character-based terminals, the text input and display is modified by the multi-line editor. This editor can automatically modify the text data by adding blanks to put words to the next line, in order to make the text fit into the form field. In GUI mode, the text input and display is managed by a multi-line edit control.

The maximum number of bytes a user can enter is the width of the form-field multiplied by the height of the form-field. Blank characters may be intentional blanks or fill blanks. Intentional blanks are initially stored in the target variable where entered by the user. Fill blanks are inserted at the end of a line by the editor when a new-line or a word-alignment forces a line-break. It is not possible to set the cursor at a fill blank. Intentional blanks are always displayed (even on the beginning of a line; the word-wrapping method used in reports with PRINT WORDWRAP works differently).

When entering characters with Japanese locales, special characters are prohibited from being the first or the last character on a line. If the last character is prohibited from ending a line, this character is wrapped down to the next line. If the first character is prohibited from beginning a line, the preceding character will also wrap down to the next line. This method is called kinsoku. The test for prohibited characters will be done only once for the first and the last character on each line.

Word-wrapping is disabled on the last row of a WORDWRAP field. The last word on the last row may by truncated. The WORDWRAP COMPRESS attribute instructs the editor to remove fill blanks before assigning the field-buffer to the target variable. The WORDWRAP NONCOMPRESS attribute instructs the editor to store fill blanks to the target variable. The WORDWRAP and WORDWRAP NONCOMPRESS attributes are equivalent.

Form Rendering

Summary:

- A character-based grid
  - Grid layout rules
  - Size computing
  - Complex example
- Grid dependencies
  - A large number of cells for large widgets
  - HBox Tags
    - Mechanism
    - SpacerItems
- Packed Grid
  - General rule
  - Group exception

Genero has introduced a form rendering system. Forms are not based on fixed text-mode screen, but can display complex layouts. In order to support .per files, the rendering system has to manage a character-based definition, which implies very specific graphical rules.
This document explains the graphical rendering of a .per form.

A character-based grid

Warning: Scrollgrid and Groups (without `gridchildreninparent` attribute) behave the same way as Grids.

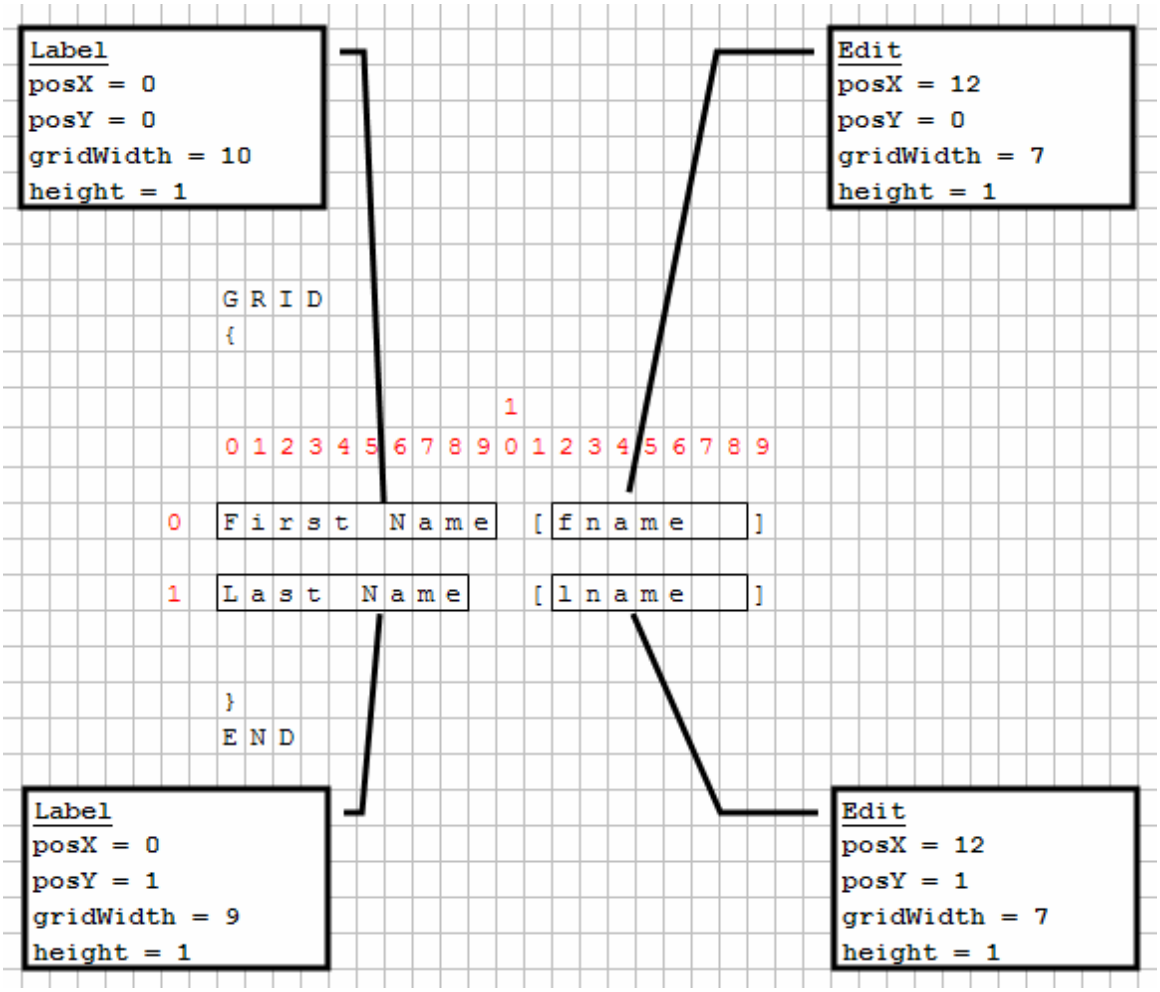
The grid container is the most important container - it contains all ‘final’ widgets (fields, buttons…). The .per file defines a form which is character based; each character defines a cell of the grid:

```
GRID
{
First Name [fname   ]
Last Name  [lname   ]
}
END
```

Above .per file layout specification can be show in a character grid as follows:

```
   GRID
   {

             1
   0123456789012345678 9

0  First  Name  [fname    ]

1  Last  Name   [lname    ]


   }
   END
```

With a fixed-font based front end, there is no problem, but Genero introduced Windows look and feel and proportional fonts. Objects are then created and added to the grid; each object has a starting position (defined by `posX` and `posY` attributes) and the number of cells taken (`gridwidth`, `gridheight` attributes).



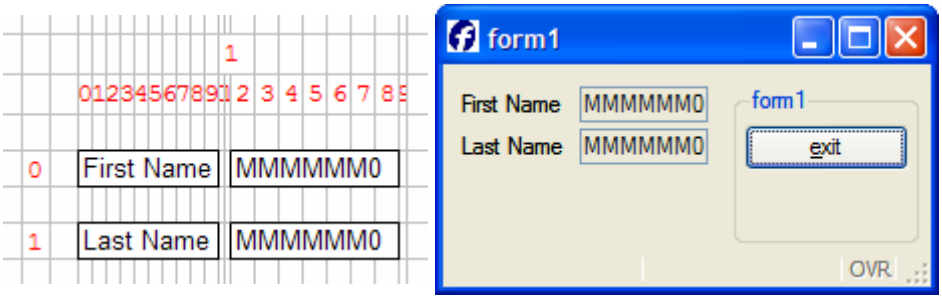
Grid layout rules

Front-ends grid layout follows these important rules:

1. Empty lines and empty columns take 0 pixels.
2. The size of a cell depends on the size of the widgets inside the grid.
3. Widget's minimum size is computed via its size attribute.
4. Widget's real size is computed to completely fill the cells in the GRID (this depends on the `sizepolicy` attribute).
5. A small spacing is applied in non-empty cells.

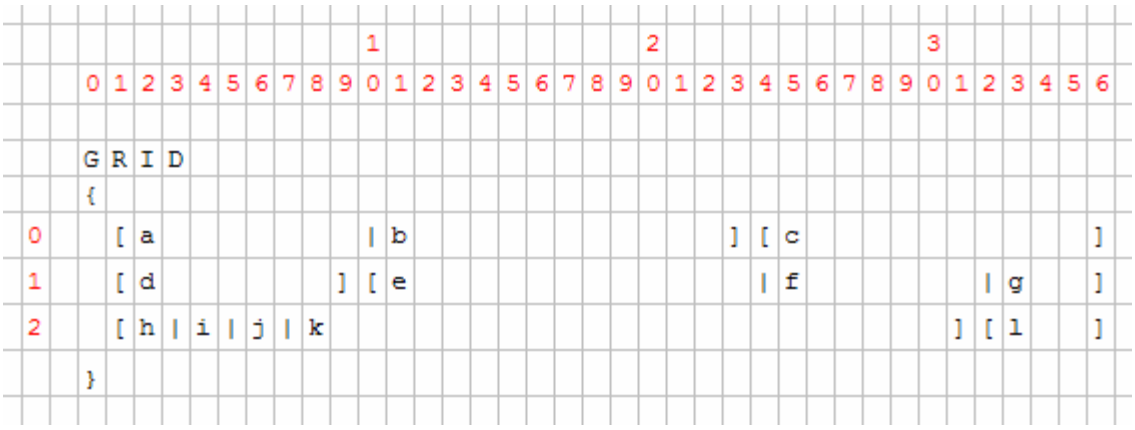
Size computing

Each widget's minimum size is computed according to its `size` and `sizepolicy` (rule #3); the size of cells of the grid is then computed (rule #1 and #2), and the widget's size can change to fill the cells (rule #4).

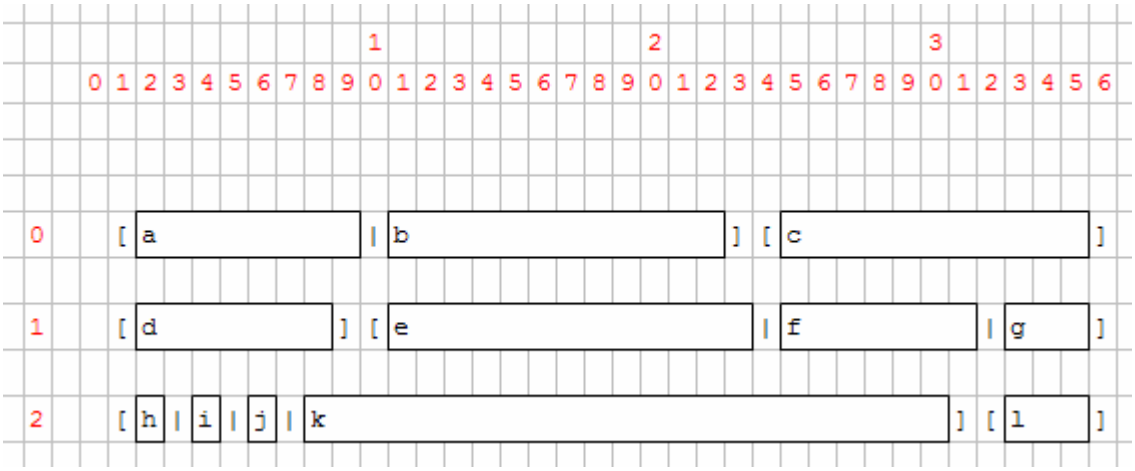


Complex example

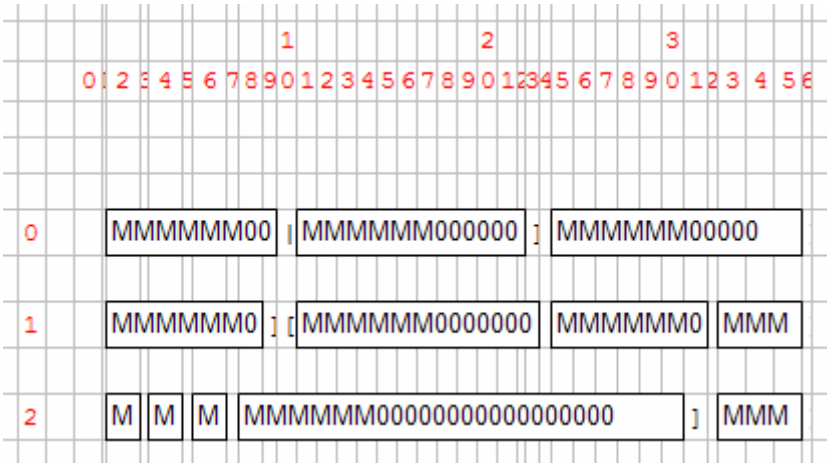
This Grid contains several fields.

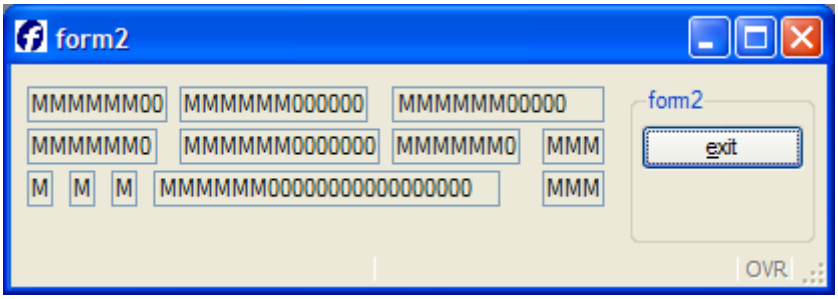


For each field, the position and the number of cells is computed by the form compiler. Then the front-end creates the widgets and sets them on the grid.



Once widgets are on the grid, their minimum size is computed according to their `size` and `sizepolicy` attributes. Then the grid cells are computed.





You can see that fields k and c are much bigger than expected:

- Field g and l make columns 33, 34 and 35 bigger than the other,
- Field f extends columns 25 to 31.
- As field c has to fill columns 25 to 35, its size grows; the same for field k.

Some fields are proportionally bigger than others because some parameters are variable, others fixed. Field width is computed as follows:

> The width of the content (depending on `sizepolicy` and `sample`, but by default a combination of 'M' and '0'), plus the border.

For example, a field of 1 will be as wide as 2 borders + 1 'M'. A field of 10 will be as wide as 2 borders + 6 'M' + 4 '0'. This means that a field of 1 is far from being 10 times smaller than a field of 10.

Grid dependencies

Rule 2 (the size of a cell depends on the size of the widgets inside the grid) is useful to keep text-mode alignment:

```
GRID
{
  [a      ]
  [b      ]
}
END
```

This .per implies that a and b start at the same position and have the same size, whatever a and b are.

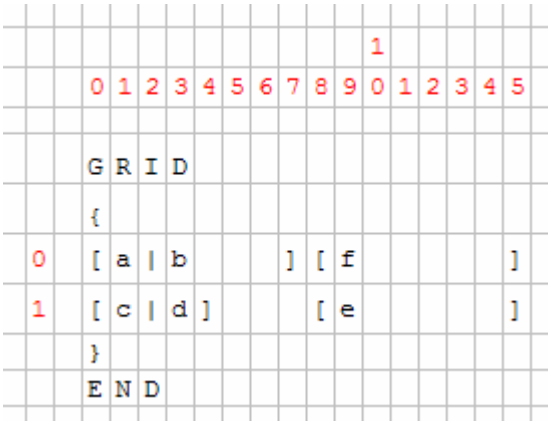
A large number of cells for large widgets

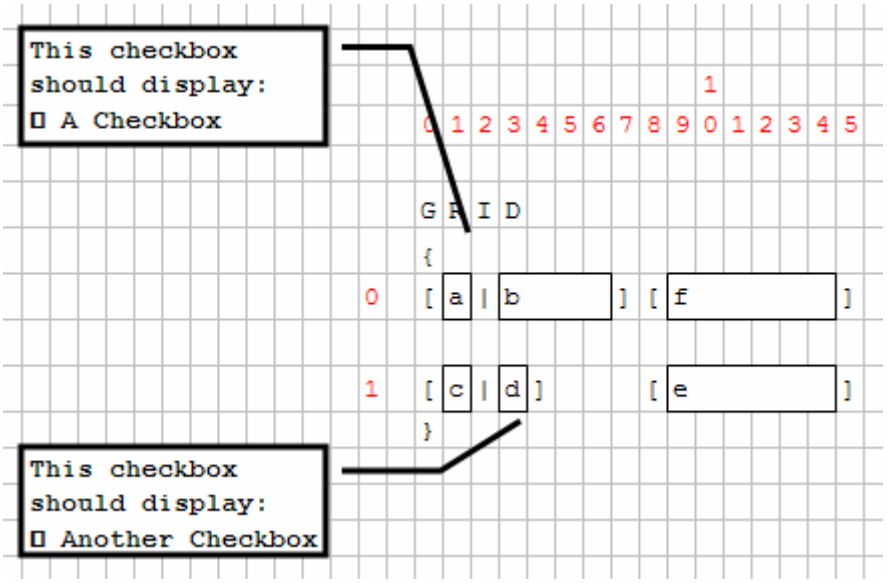
This rule could lead to very different results, especially when a large widget is assigned into a small number of cells.

Example:

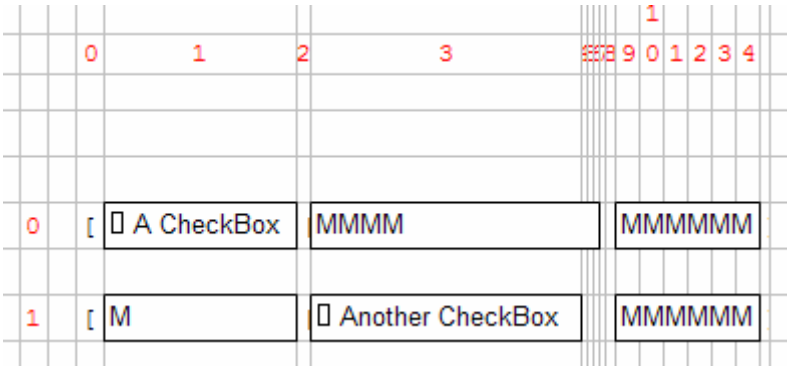
```
LAYOUT
GRID
{
[a|b   ][f     ]
[c|d]   [e     ]
}
END
END

ATTRIBUTES
CHECKBOX a = formonly.a, TEXT="A Checkbox";
EDIT b = formonly.b;
EDIT c = formonly.c;
CHECKBOX d = formonly.d, TEXT="Another Checkbox";
EDIT e = formonly.e;
EDIT f = formonly.f;
END
```

As seen previously, the grid will be computed regarding characters:

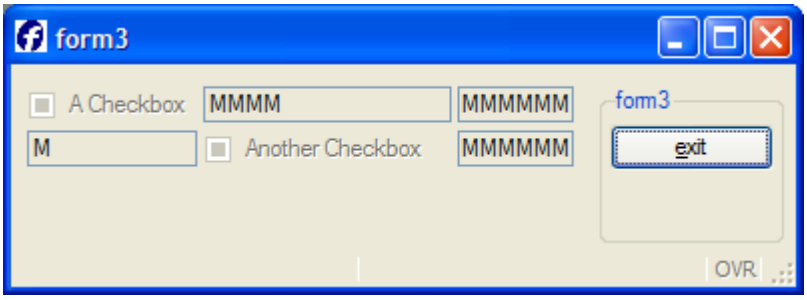




Then the minimum size of each widget and the layout is computed. Cells (0,1) and (1,3) contain a checkbox; these checkboxes will enlarge columns 1 and 3.



As Edit "c" is defined to have the same width as checkbox "a", it will be much larger as expected:

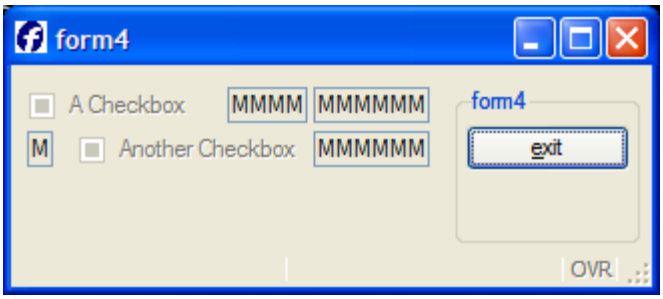


To avoid this “strange” result, the form designer should assign a realistic number of cells for each object:

```
GRID
{
```

```
[a        |b    ][f      ]
[c|d             ][e      ]
}
```

Even if the LAYOUT section is wider, the result will be smaller:



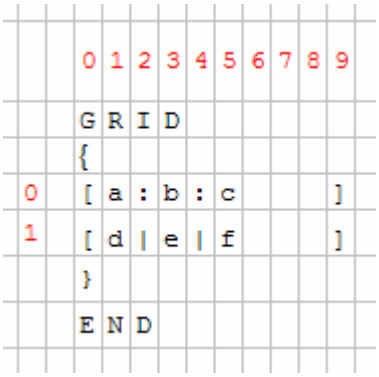
HBox Tags

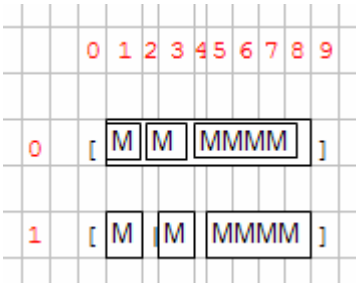
Mechanism

To get rid of the “character-based” grid, HBox Tags have been introduced. This mechanism defines a “widget container” that will gather the widget horizontally, like the HBOX layout container. All widgets inside this container are no longer dependent on the parent grid:

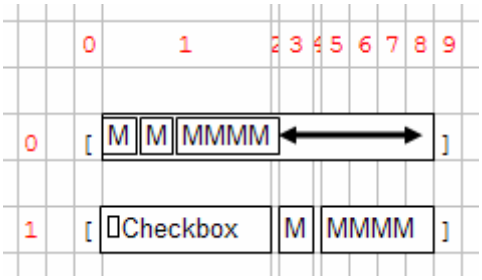
```
GRID
{
[a:b:c   ]
[d|e|f   ]
}
END
```

The notation “:” defines the HBox Tag. A container is created and will contain widgets a, b and c. These widgets won’t be aligned in the Grid:



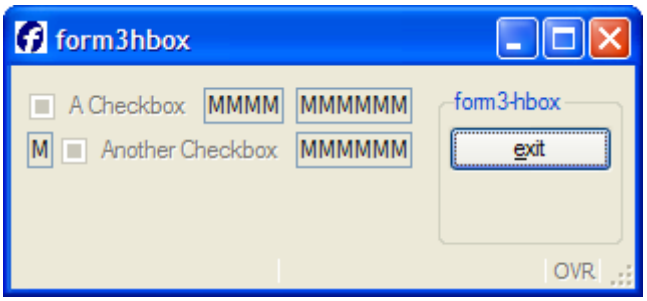


This mechanism is useful when you have large widgets in a small number of cells in one row and don't want to have dependencies:



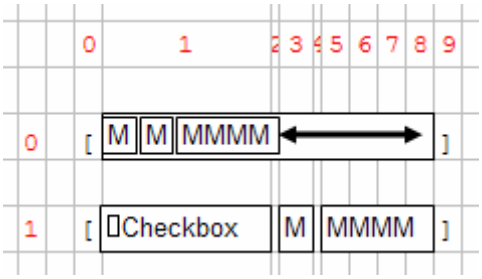
If we take the "form3" example again, and modify it with HBox Tags:

```
GRID
{
[a:b   ][f     ]
[c:d   ][e     ]
}
END
```



Spacer Items in HBox tags

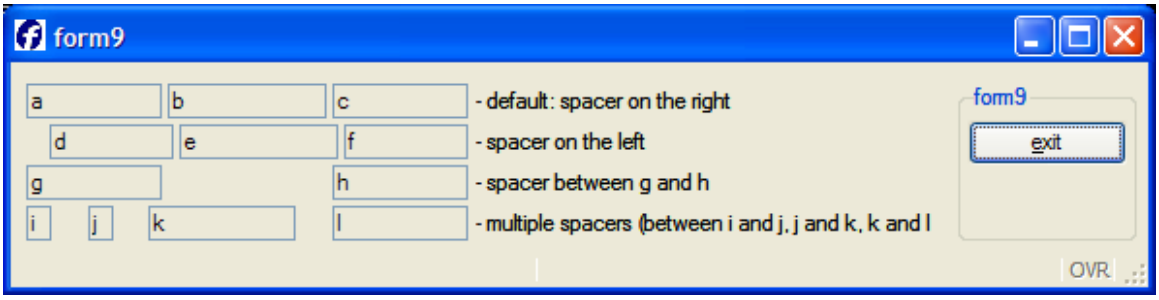
HBox tags also introduces the SpacerItems concept: when a grid HBox is created, the content may be smaller than the container:



Because of the checkbox, the cell 1 is very large, and then the HBox is larger than the three fields. A SpacerItem object is automatically created by the form compiler; the role of the SpacerItem is to take all the free space in the container. Then all the widgets are packed at the left.

By default, a SpacerItem is created at the right of the container, but the spacer can also be defined in another place:

```
GRID
{
[a       :b       :c       ] <- default: spacer on the right
[ :d     :e       :f       ] <- spacer on the left
[g       :        :h       ] <- spacer between g and h
[i: :j: :k      : :l       ] <- multiple spacers (between i and j, j
and k, k and l
}
END
```



Packed Grid

General rule

When you resize a window, the content will either grow with the window or be packed in the top left position. The rule followed by the front-end is that the grid is packed (horizontally / vertically / both) if nothing can grow in that direction.

The following widgets can grow horizontally:

- Tables
- Images (`stretch`=both or `stretch`=`x`)
- TextEdits (`stretch`=both or `stretch`=`x`)

The following widgets can grow vertically:

- Tables (without `wantfixedpagesize`)
- Images (`stretch`=both or `stretch`=`y`)
- TextEdits (`stretch`=both or `stretch`=`y`)

Group's exception

In general, a `GRID` can grow if any object inside the `GRID` can grow. The exception to this rule: If there is only one `GROUP` (defined without the `GRIDCHILDRENINPARENT` attribute) inside a `GRID` and nothing else, the grid can grow.

This exception allows better rendering of a grouped grid:

• A packed grid:

• An unpacked grid:

Menus

Summary:

- Basics
- Syntax
- Usage
  - Programming Steps
  - Instruction Configuration
  - Default Actions
  - Control Blocks
  - Interaction Blocks
  - Control Instructions
  - Default Accelerator Keys
  - Using the COMMAND clause
  - Using the COMMAND KEY clause
- Examples

See also: Dynamic User Interface

Basics

A Menu defines a list of options (also called actions) that can trigger program routines. The `MENU` statement is an interactive instruction defining the possible actions that can be executed in a specific context at a given place in the program. One Menu can only define a set of options for a given level. You cannot define all menu options of your program in a unique Menu; you must use nested Menu calls. A typical application starts with a global Menu, defining general options to access sub-routines, which in turn implement specific Menus with options like 'Append', 'Delete', 'Modify', and 'Search' that trigger sub-routines to manage database records.

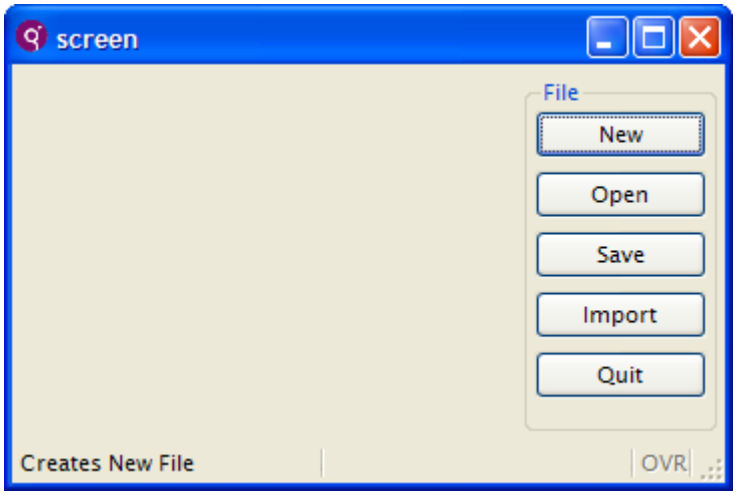
A `MENU` statement is a controller for user actions. You bind actions views with Menu options *by name*. For example, if your Menu contains `ON ACTION sendmail`, you can specify which action view the user would use to trigger the sendmail action by:

- Set the name attribute with the action name (`sendmail`) for a *Toolbar* button (see Toolbars for more details).
- Set the name attribute with the action name (`sendmail`) for a *TopMenu* button (see Topmenus for more details).
- Set the item-name with the action name (`sendmail`) for a *Button* widget (see Form Specification Files for more details).

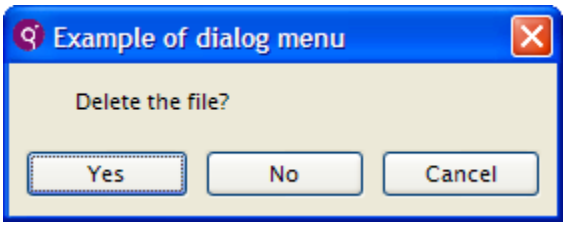
Action views are bound to an action controller *by name*. See the Interaction Model description for more details.

Warning: When binding action views to menu option clauses, the action name is case sensitive. The compiler converts `COMMAND` labels and `ON ACTION` identifiers to lowercase for the action. It is recommended that you use all lowercase letters when providing the action name for the action view.

By default, if no action views are associated with Menu options, the Menu options are displayed as simple push buttons in a specific area, depending on the front end. The following screen is produced by the program shown in Example 2:



You can display a Menu in a modal dialog window by setting some attributes, as shown in the source code of Example 3:



For more details about default and explicit action views, see Interaction Model.

MENU

Purpose:

The `MENU` instruction defines a set of user choices.

Syntax:

```
MENU [title]
   [ ATTRIBUTES ( control-attributes ) ]
   [ BEFORE MENU
```

```
        menu-statement
        [...]
    ]
    menu-option
    [...]
END MENU
```

where *menu-option* is one of:

```
{ COMMAND option-name [option-comment] [ HELP help-number ]
    menu-statement
    [...]
| COMMAND KEY ( key-name ) option-name [option-comment] [ HELP help-
number ]
    menu-statement
    [...]
| COMMAND KEY ( key-name )
    menu-statement
    [...]
| ON ACTION action-name
    menu-statement
    [...]
| ON IDLE idle-seconds
    menu-statement
    [...]
}
```

where *menu-statement* is:

```
{ statement
| CONTINUE MENU
| EXIT MENU
| NEXT OPTION option
| SHOW OPTION { ALL | option [,...] }
| HIDE OPTION { ALL | option [,...] }
}
```

Notes:

1. *title* is a string expression defining the title of the menu.
2. *control-attributes* is a list of attributes that defines the behavior and presentation of the menu.
3. *key-name* is an hot-key identifier (like `F11` or `Control-z`).
4. *option-name* is a string expression defining the label of the menu option and identifying the action that can be executed by the user.
5. *option-comment* is a string expression containing a description for the menu option, displayed when *option-name* is the current.
6. *help-number* is an integer that allows you to associate a help message number with the menu *option.*
7. *action-name* identifies an action that can be executed by the user.
8. *idle-seconds* is an integer literal or variable that defines a number of seconds.
9. *action-name* identifies an action that can be executed by the user.
10. *bool-expr* is an integer expression that must evaluate to 0 or 1.

11. *variable* is a program variable that receives the return of the GET instruction.

The following table shows the *control-attributes* supported by the MENU statement:

Attribute	Description
STYLE = *string*	Defines the type of the menu. Values can be 'default', 'dialog' or 'popup'.
COMMENT = *string*	Defines the message associated with this menu.
IMAGE = *string*	Defines the icon file associated with this menu.

Usage

The MENU instruction can be used to control user actions. See Interaction Model for more details about actions.

Warnings:

1. It is recommended that you do not use the COMMAND KEY clause to handle user actions; use the ON ACTION clause instead.
2. The compiler converts the ON ACTION identifiers to lowercase for internal storage. ON ACTION APPEND is equivalent to ON ACTION append.
3. When using COMMAND "*label*", you can use uppercase letters: The compiler converts the command label to lowercase for the action name. COMMAND "Append" is equivalent to ON ACTION append, except that the text and comment attributes are defined at the program level.
4. For backward compatibility, it is possible to use simple characters and multiple keys in the COMMAND KEY clause, but this is not recommended. If multiple keys are defined in the COMMAND KEY clause, only the last element will be used as accelerator key.

Tips:

1. Window close events can be trapped with COMMAND KEY(INTERRUPT) clause. See Windows and Forms for more details.
2. Default presentation of menu buttons can be shown in different positions in the window, according to the Window Style.

Programming Steps

The typical usage of a MENU instruction is show in the following example, using a set of ON ACTION control blocks:

```
01 MENU
02    ON ACTION new
03      CALL newFile()
```

```
04    ON ACTION open
05      CALL openFile()
06    ON ACTION quit
07      EXIT PROGRAM
08 END MENU
```

The COMMAND clause defines both the action name and the label of the menu option, which is by default decorated on the front end side as a push-button in a specific area. The ON ACTION clause defines an action trigger clause as described in Interaction Model, Controlling User Actions section. To write abstract code, we recommend that you use the ON ACTION clause instead of COMMAND. However, when using 'dialog' menus, you might only need to provide the title of the buttons; in such situations, you can use COMMAND clauses.

Instruction Configuration

When the STYLE instruction attribute is set to 'default' or when you do not specify the menu type, the runtime system generates a default decoration as a set of buttons in a specific area of the current window. When this attribute is set to 'dialog', the menu options appear as buttons at the bottom in a temporary modal window, in which you can define the message and the icon with the COMMENT and IMAGE attributes. When the STYLE is set to 'popup', the menu appears as a popup menu (contextual menu).

Warning: If the menu is a "dialog" or "popup", the dialog is automatically exited after any action clause such as ON ACTION, COMMAND or ON IDLE.

Default Actions

When an MENU instruction executes, the runtime system creates a set of default actions. The following table lists the default actions created for this dialog:

Default action	Control Block execution order
close	By default, generates a cancel key press (COMMAND KEY(INTERRUPT)) Default action view is hidden. See Windows closed by the user.
help	Shows the help topic defined by the HELP clause. Default action view is hidden.

Control Blocks

If the menu block contains a BEFORE MENU clause, statements within this clause will be executed before the menu is displayed.

Interaction Blocks

The ON ACTION *action-name* clause defines a set of instructions to be executed when an action is fired.

When you enter a MENU statement, you are entering an interactive instruction, also known as a *dialog*. During a dialog, you can enable or disable an action with the setActionActive() method of the DIALOG object. You can also hide and show the default action view with the setActionHidden() method of the DIALOG object.

```
01 ...
02    BEFORE MENU
03       CALL DIALOG.setActionActive("query",FALSE)
04       CALL DIALOG.setActionHidden("adduser",TRUE)
05 ...
```

The ON IDLE *idle-seconds* clause defines a set of instructions that must be executed after *idle-seconds* of inactivity. This can be used to quit the dialog after the user has not interacted with the program for a specified period of time. The parameter *idle-seconds* must be an integer literal or variable. If it evaluates to zero, the timeout is disabled.

```
01 ...
02    ON IDLE 10
03       IF ask_question("Do you want to leave the dialog?") THEN
04          EXIT MENU
05       END IF
06 ...
```

Control Instructions

CONTINUE MENU statement causes the runtime system to ignore the remaining instructions in the current block and redisplay the menu.

EXIT MENU statement terminates the MENU block without executing any other statement.

The NEXT OPTION *option* statement defines *option* as the default. This cannot apply to a hidden option, and works only with default action views created when an explicit view is not used.

The SHOW OPTION and HIDE OPTION statements are provided for backward compatibility only. The SHOW OPTION statement is used to make the default action view visible and the explicit action views enabled. The HIDE OPTION statement is used to make the default action view invisible and the explicit action views disabled. The ALL clause can be used to specify all options. It is now recommended that you hide and show action views with the setActionHidden() method of the DIALOG object.

Default Accelerator Keys

When a Menu instruction executes, the first letter of the display text on the action view is, by default, underscored. For an action views where the underscored letter is not shared with other action views, pressing the key corresponding to that letter will execute that action. For an action view where the underscored letter is shared with other action views surfaced by the Menu statement, pressing the key corresponding to that letter will toggle the focus between all action views that share the same letter. For example:

```
01 MENU
02   COMMAND "Start"
03     DISPLAY "Start"
04   COMMAND "cmdone"
05     DISPLAY "Command 1"
06   COMMAND "cmdtwo"
07     DISPLAY "Command 2"
08   COMMAND "Quit"
09     EXIT MENU
10 END MENU
```

In this example, if you press "S", the action "`start`" will be performed. If you press "C", the focus will toggle between "`cmdone`" and "`cmdtwo`". If you press "Q", the action "`quit`" will be performed.

For information on specifying an alternate accelerator key using the ampersand (&), see "Using the COMMAND clause" below.

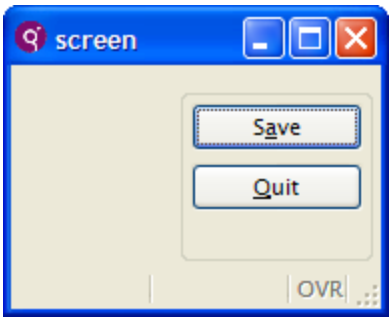
Using the COMMAND clause

When using the `COMMAND` clause, the name of the action will be the option text converted to lowercase letters. The *text* and *comment* decoration attributes for the default action view will get the value of the option text and comment text of the menu clause. As these attributes are defined by the program, the corresponding *text* and *comment* attributes in Action Defaults are not applied to these menu options. For example, when you define a `COMMAND "Hello" "This is the Hello option"` menu option, the name of the action will be "`hello`", the button text will be "`Hello`", and the comment will be "`This is the Hello option`", even if an action default defines a different text or comment for the action "`hello`".

With the COMMAND clause, if the user includes an ampersand (&) in the option text, the system transforms it as an accelerator for the next character. The ampersand is not considered when determining the name of the action. For example:

```
01 MENU
02    COMMAND "S&ave"
03      ...
04    COMMAND "&Quit"
05       EXIT MENU
06 END MENU
```

In the first COMMAND clause of this example, the name of the action will be "`save`", the button text will be "Save" (with the "a" underscored), and the accelerator key for this command will be Alt+A. In the second COMMAND clause in this example, the name of the action will be "`quit`", the button text will be "Quit" (with the "Q" underscored), and the accelerator key for this command will be Alt+Q.



Using the COMMAND KEY clause

When the `COMMAND KEY` specifies an option text (as in `COMMAND KEY(F10,F12) "Hello"`) the name of the action is defined by the option text, in lowercase letters (i.e. "`hello`"). If the `COMMAND KEY` does not specify the option text, the action name defaults to the last key of the list in lowercase letters. For example, if you write `COMMAND KEY(F10,F12,Control-Z)`, the name of the action will be "`control-z`". So if you want to define action defaults for a `COMMAND KEY` using multiple keys, you must use the last key for the name of the action.

A `COMMAND KEY` clause implicitly defines the accelerator attributes for the action and the corresponding action default accelerators will be ignored. For backward compatibility, the `COMMAND KEY` instruction supports up to four keys. Each four keys are used to initialize the *acceleratorName*, *acceleratorName2*, *acceleratorName3* and *acceleratorName4* attributes of the action. For example, when you define a `COMMAND KEY(F10,F12) "Hello"` menu option, *acceleratorName* will be "F10" and *acceleratorName2* will be "F12", even if an Action Defaults for the action "`hello`" defines *acceleratorName* as "F5" and *acceelratorName2* as "Control=-F". However, you can set the third accelerator with the *acceleratorName3* attribute in action defaults.

Warning: In TUI mode, actions created with COMMAND [KEY] do not get accelerators of Action defaults; Only actions defined with ON ACTION will get accelerators of Action Defaults.

Examples

Example 1: Abstract action controller

```
01 MENU
02    ON ACTION new
03      CALL newFile()
04    ON ACTION open
05      CALL openFile()
06    ON ACTION save
07      CALL saveFile()
08    ON ACTION import
09      LOAD FROM "infile.dat" INSERT INTO table
10    ON ACTION quit
```

```
11      EXIT PROGRAM
12 END MENU
```

Example 2: Simple menu (old Informix 4GL style)

```
01 MENU "File"
02    COMMAND KEY ( CONTROL-N ) "New" "Creates New File" HELP 101
03      CALL newFile()
04    COMMAND KEY ( CONTROL-O ) "Open" "Open existing File" HELP 102
05      CALL openFile()
06    COMMAND KEY ( CONTROL-S ) "Save" "Save Current File" HELP 103
07      CALL saveFile()
08    COMMAND "Import"
09      LOAD FROM "infile.dat" INSERT INTO table
10    COMMAND KEY ( CONTROL-Q ) "Quit" "Quit Program" HELP 201
11      EXIT PROGRAM
12 END MENU
```

Example 3: Using a modal menu

```
01 MAIN
02    MENU "Example of dialog menu"
03       ATTRIBUTES ( STYLE="dialog", COMMENT="Delete the file?" )
04       COMMAND "Yes"
05         DISPLAY "Yes"
06       COMMAND "No"
07         DISPLAY "No"
08       COMMAND "Cancel"
09         DISPLAY "Cancel"
10    END MENU
11 END MAIN
```

Displaying Data to Forms

Summary:

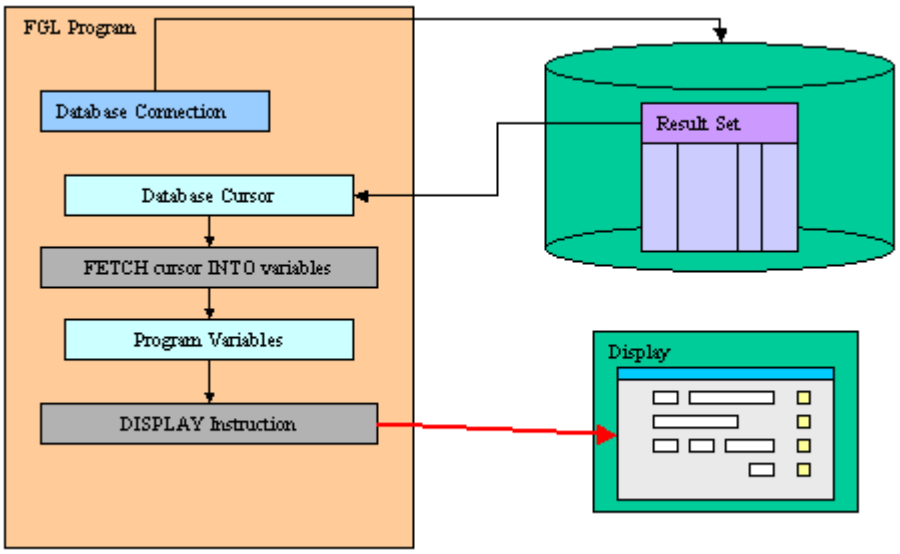
- How to display data to form fields
- Displaying data to specific form fields (`DISPLAY TO`)
- Displaying data to form fields by name (`DISPLAY BY NAME`)
- Clearing all form fields (`CLEAR FORM`)
- Clearing specific form fields (`CLEAR field`)

See also: Variables, Records, Windows, Forms, Record Input, Display Array

---

How to display data to form fields

Programs retrieve data from the database into variables with a cursor or a static SELECT statement, and display the variable values to the current form with the `DISPLAY` instruction:



DISPLAY TO

Purpose:

The `DISPLAY TO` instruction displays data to form fields explicitly.

Syntax:

```
DISPLAY expression [,...] TO field-list [,...]
  [ ATTRIBUTE ( display-attribute [,...] ) ]
```

where *field-list* is :

```
{ field-name
| table-name.*
| table-name.field-name
| screen-array[line].*
| screen-array[line].field-name
| screen-record.*
| screen-record.field-name
} [,...]
```

Notes:

1. *expression* is any expression supported by the language.
   This is typically a list of variables or a record with the `.*` notation.
2. *field-name* is the identifier of a field of the current form.
3. *table-name* is the identifier of a database table of the current form.
4. *screen-record* is the identifier of a screen record of the current form.
5. *screen-array* is the screen array that will be used in the form.
6. *display-attribute* is one of the display attributes supported in this instruction. See below for more details.

Warning: The `DISPLAY TO` statement changes the 'touched' status of the target fields. When you modify a field value with this instruction, the FIELD_TOUCHED() operator returns TRUE and the ON CHANGE trigger may be fired if the current field value was changed with a `DISPLAY TO`. During an `INPUT` or `INPUT ARRAY`, we recommend you to use the `UNBUFFERED` attribute to display data automatically to fields without changing the 'touched' status of fields.

Usage:

If the variables do not have the same names as the form fields, you must use the `TO` clause to explicitly map the variables to fields. You can list the fields individually, or you can use the *`screen-record`*`.*` or *`screen-record`*`[n].*` notation, where *`screen-record`*`[n].*` specifies all the fields in line `n` of a screen array.

In a `DISPLAY TO` statement, any screen attributes specified in the `ATTRIBUTE` clause apply to all the fields that you specify after the `TO` keyword.

In the following example, the values in the **p_items** program record are displayed in the first row of the **s_items** screen array:

```
01 DISPLAY p_items.* TO s_items[1].*
```

The expanded list of screen fields must correspond in order and in number to the expanded list of identifiers after the DISPLAY keyword. Identifiers and their corresponding fields must have the same or compatible data types. For example, the next DISPLAY statement displays the values in the **p_customer** program record in fields of the **s_customer** screen record:

```
01 DISPLAY p_customer.* TO s_customer.*
```

For this example, the **p_customer** program record and the **s_customer** screen record require compatible declarations. The following DEFINE statement declares the **p_customer** program record:

```
01 DEFINE p_customer RECORD
02   customer_num LIKE customer.customer_num,
03   fname LIKE customer.fname,
04   lname LIKE customer.lname,
05   phone LIKE customer.phone
05 END RECORD
```

This fragment of a form specification declares the **s_customer** screen record:

```
01 ATTRIBUTES
02  f000 = customer.customer_num;
03  f001 = customer.fname;
04  f002 = customer.lname;
05  f003 = customer.phone;
06 END
```

The ATTRIBUTE clause temporarily overrides any default display attributes or any attributes specified in the OPTIONS or OPEN WINDOW statements for the fields. When the DISPLAY statement completes execution, the default display attributes are restored.

The following table shows the *display-attributes* supported by the DISPLAY TO statement. The *display-attributes* affect console-based applications only, they do not affect GUI-based applications.

Attribute	Description
BLACK, BLUE, CYAN, GREEN, MAGENTA, RED, WHITE, YELLOW	The color of the displayed data.
BOLD, DIM, NORMAL	The font attribute of the displayed data.
REVERSE, BLINK, UNDERLINE	The video attribute of the displayed data.

The REVERSE, BLINK, INVISIBLE, and UNDERLINE attributes are not sensitive to the color or monochrome status of the terminal, if the terminal is capable of displaying these intensity modes. The ATTRIBUTE clause can include zero or more of the BLINK, REVERSE, and UNDERLINE attributes, and zero or one of the other attributes. That is, all of the attributes except BLINK, REVERSE, and UNDERLINE are mutually exclusive.

The DISPLAY statement ignores the INVISIBLE attribute, regardless of whether you specify it in the ATTRIBUTE clause.

DISPLAY BY NAME

Purpose:

The DISPLAY BY NAME instruction displays data to form fields explicitly *by name*.

Syntax:

```
DISPLAY BY NAME { variable | record.* } [,...]
  [ ATTRIBUTE ( display-attribute [,...] ) ]
```

Notes:

1. *variable* is a program variable that has the same name as a form field.
2. *record.** is a record variable that has members with the same names as form fields. The record name prefix is ignored.
3. *display-attribute* is one of the display attributes supported in this instruction. See below for more details.

Warning: The DISPLAY BY NAME statement changes the 'touched' status of the target fields. When you modify a field value with this instruction, the FIELD_TOUCHED() operator returns TRUE and the ON CHANGE trigger may be fired if the current field value was changed with a DISPLAY BY NAME. During an INPUT or INPUT ARRAY, we recommend that you use the UNBUFFERED attribute to display data to fields automatically without changing the 'touched' status of fields.

Usage:

If the variables to be displayed have the same name as form fields, you can use the BY NAME clause. The BY NAME clause binds the fields to variables. To use this clause, you must define variables with the same name as the form fields where they will be displayed. The language ignores any record name prefix when matching the names. The names must be unique and unambiguous; if not, this option results in an error, and the runtime system sets STATUS to a negative value.

For example, the following statement displays the values for the specified variables in the form fields with corresponding names (company and address1):

```
01 DISPLAY BY NAME p_customer.company, p_customer.address1
```

This BY NAME clause displays data to the screen fields of the default screen records. The default screen records are those having the names of the tables defined in the TABLES section of the form specification file. To use a screen array, you define a screen array in

addition to the default screen record. This default screen record holds only the first line of the screen array.

For example, the following DISPLAY statement displays the **ordno** variable only in the first line of the screen array (the default screen record):

```
01 DISPLAY BY NAME p_stock[1].ordno
```

To display **ordno** in all elements of the screen array, you can use the DISPLAY ARRAY statement, or `DISPLAY TO`, as in the next example:

```
01 FOR i=1 TO 10
02    DISPLAY p_stock[i].ordno TO sc.stock[i].ordno
03    ...
04 END FOR
```

The following table shows the *display-attributes* supported by the `DISPLAY BY NAME` statement:

Attribute	Description
`BLACK, BLUE, CYAN, GREEN, MAGENTA, RED, WHITE, YELLOW`	The color of the displayed data.
`BOLD, DIM, NORMAL`	The font attribute of the displayed data.
`REVERSE, BLINK, UNDERLINE`	The video attribute of the displayed data.

CLEAR FORM

Purpose:

The `CLEAR FORM` instruction clears all fields in the current form.

Syntax:

```
CLEAR FORM
```

Notes:

1. This instruction has no effect on any part of the screen display except the form fields.

Example:

```
01 MAIN
02    OPEN WINDOW w1 AT 1,1 WITH FORM "custlist"
03    CLEAR FORM
```

```
04    CLOSE WINDOW w1
05 END FOR
```

CLEAR *field*

Purpose:

The `CLEAR field` instruction clears specific fields in the current form.

Syntax:

```
CLEAR field-list
```

where *field-list* is :

```
{ field-name
| table-name.*
| table-name.field-name
| screen-array[line].*
| screen-array[line].field-name
| screen-record.*
| screen-record.field-name
} [,...]
```

Notes:

1. *field-name* is the identifier of a field of the current form.
2. *table-name* is the identifier of a database table of the current form.
3. *screen-record* is the identifier of a screen record of the current form.
4. *screen-array* is the screen array that will be used in the form.

Example:

```
01 FOR i=1 TO 10
02    CLEAR s_items[i].*
03 END FOR
```

Record Input

Summary:

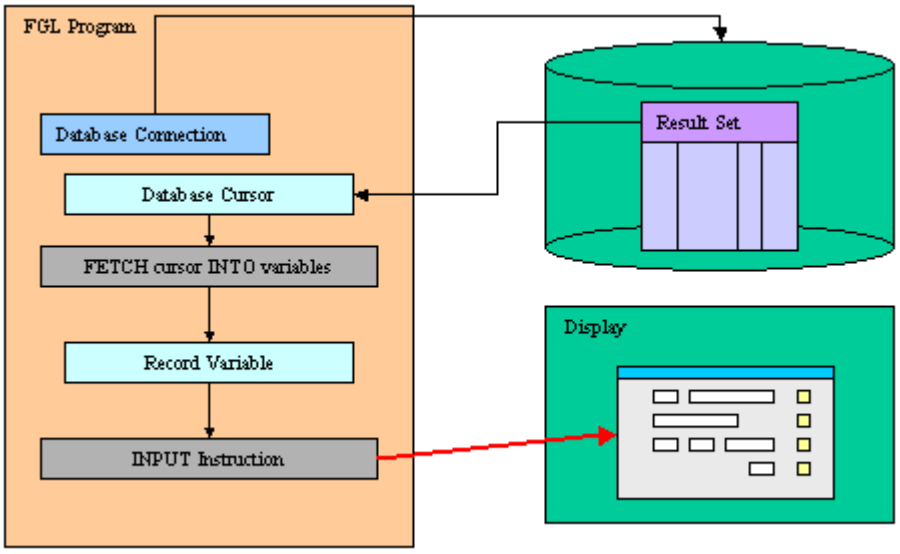
- Basics
- Syntax
- Usage
  - Programming Steps
  - Instruction Configuration
  - Default Actions
  - Control Blocks
  - Control Blocks Execution Order
  - Interaction Blocks
  - Control Instructions
  - Control Class
  - Control Functions
- Examples
  - Example 1: Simple INPUT statement using screen record specification
  - Example 2: Complex INPUT statement using the BY NAME clause and control blocks

See also: Variables, Records, Windows, Forms, Record Display, Display Array

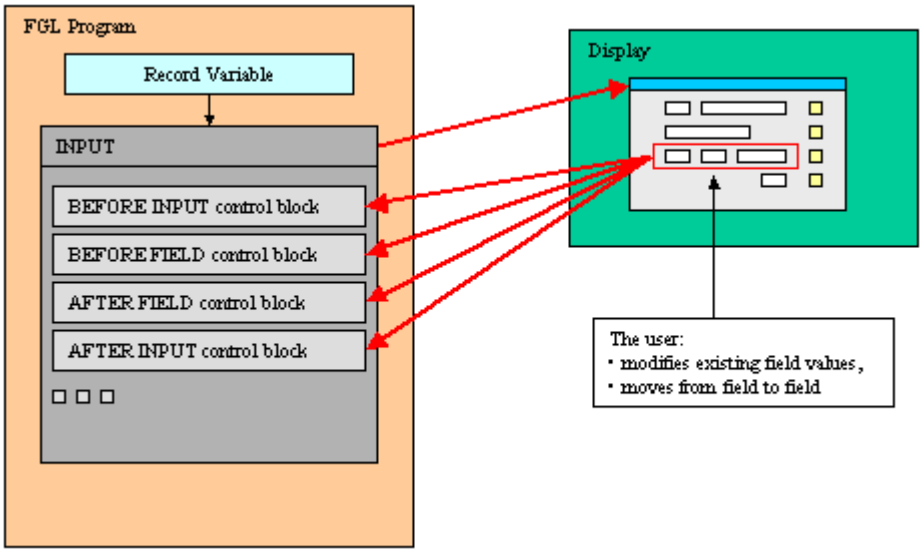
---

Basics

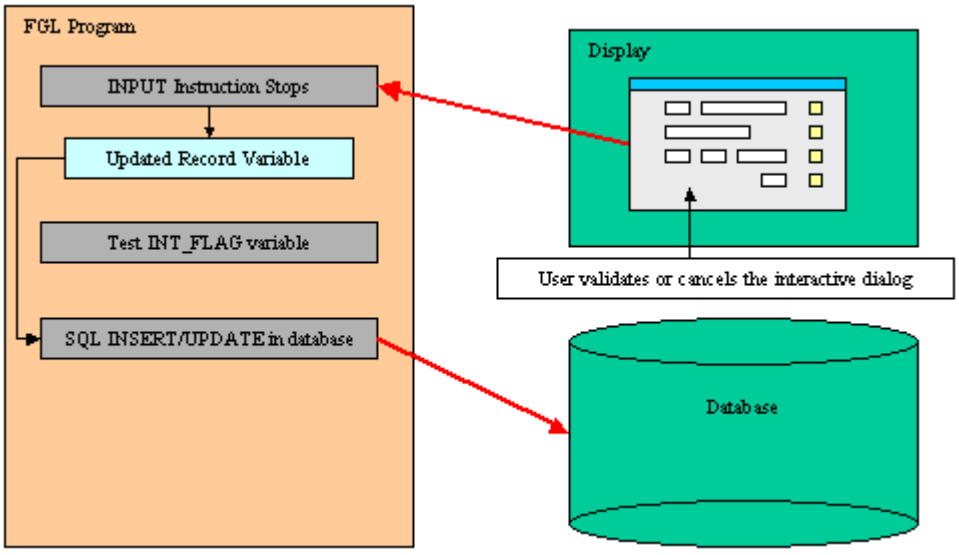
The programs maintain data in variables and use the `INPUT` statement to bind variables to screen-records or screen-arrays of forms for data entry into form fields. The `INPUT` statement activates the current form (the form that was most recently displayed or the form in the current window):



During the `INPUT` statement execution, the user can edit the record fields, while the program controls the behavior of the instruction with control blocks:



To terminate the `INPUT` execution, the user can validate (or cancel) the dialog to commit (or invalidate) the modifications made in the record:



When the statement completes execution, the form is de-activated. After the user terminates the input (for example, with the Accept key), the program must test the INT_FLAG variable to check if the dialog was validated (or canceled), and then can use the INSERT or UPDATE SQL statements to modify the appropriate database tables.

INPUT

Purpose:

The INPUT statement supports data entry into the fields of the current form.

Syntax 1: Implicit field-to-variable mapping

```
INPUT BY NAME { variable | record.* } [,...]
  [ WITHOUT DEFAULTS ]
  [ ATTRIBUTES ( { display-attribute | control-attribute } [,...] ) ]
  [ HELP help-number ]
[ dialog-control-block
  [...]
END INPUT ]
```

Syntax 2: Explicit field-to-variable mapping

```
INPUT { variable | record.* } [,...]
  [ WITHOUT DEFAULTS ]
  FROM field-list
  [ ATTRIBUTES ( { display-attribute | control-attribute } [,...] ) ]
  [ HELP help-number ]
[ dialog-control-block
  [...]
END INPUT ]
```

where *dialog-control-block* is one of:

```
{ BEFORE INPUT
| AFTER INPUT
| BEFORE FIELD field-spec
| AFTER FIELD field-spec
| ON CHANGE field-spec
| ON IDLE idle-seconds
| ON ACTION action-name
| ON KEY ( key-name [,...] )
}
    dialog-statement
    [...]
```

where *dialog-statement* is one of:

```
{ statement
| ACCEPT INPUT
| CONTINUE INPUT
| EXIT INPUT
| NEXT FIELD { CURRENT | NEXT | PREVIOUS | field-name }
}
```

where *field-list* is:

```
{ field-name
| table-name.*
| table-name.field-name
| screen-array[line].*
| screen-array[line].field-name
| screen-record.*
| screen-record.field-name
} [,...]
```

where *field-spec* is:

```
{ field-name
| table-name.field-name
| screen-array.field-name
| screen-record.field-name
} [,...]
```

Notes:

1. *variable* is a program variable that will be filled by the `INPUT` statement.
2. *record.** is a record variable that will be filled by the `INPUT` statement.
3. *help-number* is an integer that allows you to associate a help message number with the instruction.
4. *field-name* is the identifier of a field of the current form.
5. *table-name* is the identifier of a database table of the current form.
6. *screen-record* is the identifier of a screen record of the current form.
7. *screen-array* is the screen array that will be used in the form.
8. *line* is a screen array line in the form.
9. *key-name* is a hot-key identifier (like `F11` or `Control-z`).
10. *idle-seconds* is an integer literal or variable that defines a number of seconds.
11. *action-name* identifies an action that can be executed by the user.
12. *statement* is any instruction supported by the language.

The following table shows the *options* supported by the `INPUT` statement:

Attribute	Description
`HELP help-number`	Defines the help number when help is invoked by the user, where *help-number* is an integer literal or a program variable. **See Warning below!**
`WITHOUT DEFAULTS`	Indicates that the data rows are not filled (TRUE) with the column default values defined in the form specification file or the database schema files (Default is FALSE). **See Warning below!**

The following table shows the *display-attributes* supported by the `INPUT` statement. The *display-attributes* affect console-based applications only, they do not affect GUI-based applications.

Attribute	Description
BLACK, BLUE, CYAN, GREEN, MAGENTA, RED, WHITE, YELLOW	The color of the displayed data.
BOLD, DIM, INVISIBLE, NORMAL	The font attribute of the displayed data.
REVERSE, BLINK, UNDERLINE	The video attribute of the displayed data.

The following table shows the *control-attributes* supported by the INPUT statement:

Attribute	Description
NAME = *string*	Identifies the dialog statement with a clear name.
HELP = *help-number*	Defines the help number when help is invoked by the user, where *help-number* is an integer literal or a program variable. **See Warning below!**
WITHOUT DEFAULTS [=*bool*]	Indicates if the data rows must be filled (FALSE) or not (TRUE) with the column default values defined in the form specification file or the database schema files. The *bool* parameter can be an integer literal or a program variable. **See Warning below!**
FIELD ORDER FORM	Indicates that the tabbing order of fields is defined by the TABINDEX attribute of form fields. The default order in which the focus moves from field to field in the screen array is determined by the order of the variables used by the INPUT statement. The program options instruction can also change this behavior with FIELD ORDER FORM options.
WRAP	
UNBUFFERED [=*bool*]	Indicates that the dialog must be sensitive to program variable changes. The *bool* parameter can be an integer literal or a program variable.
CANCEL = *bool*	Indicates if the default *cancel* action should be added to the dialog. If not specified, the action is registered. The *bool* parameter can be an integer literal or a program variable.
ACCEPT = *bool*	Indicates if the default *accept* action should be added to the dialog. If not specified, the action is registered. The *bool* parameter can be an integer literal or a program variable.

Usage

Warnings:

1. Although the `ON KEY` block is supported for backward compatibility, it is recommended that you use `ON ACTION` instead.
2. For new programs, specify `UNBUFFERED` mode rather than using the default buffered mode.
3. For the `HELP` and `WITHOUT DEFAULTS` options, the appearance order is important : **the last option overrides the previous one!**

Programming Steps

The following steps describe how to use the `INPUT` statement:

1. Create a form specification file, with an optional screen record. The screen record identifies the presentation elements to be used by the runtime system to display the records. if you omit the declaration of the screen record in the form file, the runtime system will use the default screen records created by the form compiler for each table listed in the TABLES section and for the `FORMONLY` pseudo-table.
2. Make sure that the program controls interruption handling with DEFER INTERRUPT, to manage the validation/cancellation of the interactive dialog.
3. Define a program record with the DEFINE instruction. The members of the program record must correspond to the elements of the screen record, by number and data types.
4. Open and display the form, using an OPEN WINDOW with the `WITH FORM` clause or the OPEN FORM / DISPLAY FORM instructions.
5. If needed, fill the program record with data, for example with a result set cursor.
6. Write the `INPUT` statement to handle data input.
7. Inside the `INPUT` statement, control the behavior of the instruction with `BEFORE INPUT`, `BEFORE FIELD`, `AFTER FIELD`, `AFTER INPUT` and `ON KEY` blocks.
8. After the `INPUT ARRAY` statement, test the INT_FLAG pre-defined variable to check if the interactive dialog was canceled (INT_FLAG = TRUE) or validated (INT_FLAG = FALSE).

Variable Binding

The program record member variables are bound to the fields of a screen record, so the `INPUT` instruction can manipulate the values that the user enters in the form fields.

The `BY NAME` clause implicitly binds the fields to the variables that have the same identifiers as the field names. You must first declare variables with the same names as the fields from which they accept input. The runtime system ignores any record name

prefix when making the match. The unqualified names of the variables and of the fields must be unique and unambiguous within their respective domains. If they are not, the runtime system generates an exception, and sets the STATUS variable to a negative value.

The FROM clause explicitly binds the fields in the screen record to a list of program variables that can be simple variables or records. The form can include other fields that are not part of the specified variable list, but the number of variables or record members must equal the number of fields listed in the FROM clause. Each variable must be of the same (or a compatible) data type as the corresponding screen field. When the user enters data, the runtime system checks the entered value against the data type of the variable, not the data type of the screen field.

The program variables can be of any data type: The runtime system will adapt input and display rules to the variable type. If a variable is declared LIKE a SERIAL column, however, the runtime system does not allow the screen cursor to stop in the field. (Values in SERIAL columns are automatically generated by the database server, not by the runtime system.)

The variables act as data model to display data or to get user input through the INPUT instruction. Always use the variables if you want to change some field values programmatically. When using the UNBUFFERED attribute, the instruction is sensitive to program variable changes: If you need to display new data during the INPUT execution, just assign the values to the program variables; the runtime system will automatically display the values to the screen:

```
01 INPUT p_items.* FROM s_items.* ATTRIBUTES (UNBUFFERED)
02    ON CHANGE code
03       IF p_items.code = "A34" THEN
04          LET p_items.desc = "Item A34"
05       END IF
06 END INPUT
```

When the INPUT instruction executes, any column default values are displayed in the screen fields, unless you specify the WITHOUT DEFAULTS keywords. The column default values are specified in the form specification file with the DEFAULT attribute, or in the database schema files.

If you specify the WITHOUT DEFAULTS option, however, the form fields display the current values of the variables when the INPUT statement begins. This option is available with both the BY NAME and the FROM binding clauses.

```
01 LET p_items.code = "A34"
02 INPUT p_items.* FROM s_items.* WITHOUT DEFAULTS
03    BEFORE INPUT
04       MESSAGE "You should see A34 in field 'code'..."
05 END INPUT
```

Instruction Configuration

The `ATTRIBUTES` clause specifications override all default attributes and temporarily override any display attributes that the OPTIONS or the OPEN WINDOW statement specified for these fields. While the `INPUT` statement is executing, the runtime system ignores the `INVISIBLE` attribute.

- HELP option
- WITHOUT DEFAULTS option
- FIELD ORDER FORM option
- WRAP option
- UNBUFFERED option
- ACCEPT option
- CANCEL option

HELP option

The `HELP` clause specifies the number of a help message to display if the user invokes the help while the focus is in any field used by the instruction. The predefined 'help' action is automatically created by the runtime system. You can bind action views to the 'help' action.

Warning: The HELP *option* overrides the HELP *attribute*!

WITHOUT DEFAULTS option

Indicates if the data rows must be filled (FALSE) or not (TRUE) with the column default values defined in the form specification file or the database schema files.

FIELD ORDER FORM option

By default, the tabbing order is defined by the variable binding list in the instruction description. You can control the tabbing order by using the `FIELD ORDER FORM` attribute: When this attribute is used, the tabbing order is defined by the TABINDEX attribute of the form fields. If this attribute is used, the Dialog.fieldOrder FGLPROFILE entry is ignored.

WRAP option

The default order in which the focus moves from field to field in the screen array is determined by the order of the variables used by the `INPUT` statement. The program options instruction can also change the behavior of the `INPUT` instruction, with the `WRAP` or `FIELD ORDER FORM` options.

UNBUFFERED option

Indicates that the dialog must be sensitive to program variable changes. When using this option, you bypass the traditional "buffered" mode.

When using the traditional "buffered" mode, program variable changes are not automatically displayed to form fields; You need to execute a `DISPLAY TO` or `DISPLAY BY NAME`. Additionally, if an action is triggered, the value of the current field is not validated and is not copied into the corresponding program variable. The only way to get the text of the current field is to use GET_FLDBUF().

If the "unbuffered" mode is used, program variables and form fields are automatically synchronized. You don't need to display explicitly values with a `DISPLAY TO` or `DISPLAY BY NAME`. When an action is triggered, the value of the current field is validated and is copied into the corresponding program variable.

ACCEPT option

The `ACCEPT` attribute can be set to FALSE to avoid the automatic creation of the *accept* default action. This option can be used for example when you want to write a specific validation procedure, by using ACCEPT INPUT.

CANCEL option

The `CANCEL` attribute can be set to FALSE to avoid the automatic creation of the *cancel* default action. This is useful for example when you only need a validation action (accept), or when you want to write a specific cancellation procedure, by using EXIT INPUT.

Note that if the `CANCEL=FALSE` option is set, no *close* action will be created, and you must write an `ON ACTION close` control block to create an explicit action.

Default Actions

When an `INPUT` instruction executes, the runtime system creates a set of default actions. See the control block execution order to understand what control blocks are executed when a specific action is fired.

The following table lists the default actions created for this dialog:

Default action	Description
`accept`	Validates the `INPUT` dialog (validates fields) *Creation can be avoided with `ACCEPT` attribute.*
`cancel`	Cancels the `INPUT` dialog (no validation, int_flag is set) *Creation can be avoided with `CANCEL` attribute.*
`close`	By default, cancels the `INPUT` dialog (no validation, int_flag is set) Default action view is hidden. See Windows closed by the user.
`help`	Shows the help topic defined by the `HELP` clause.

Only created when a HELP clause is defined.

The accept and cancel default actions can be avoided with the ACCEPT and CANCEL dialog control attributes:

```
01  INPUT BY NAME field1 ATTRIBUTES ( CANCEL=FALSE )
02        ...
```

Control Blocks

- BEFORE INPUT block
- AFTER INPUT block
- BEFORE FIELD block
- ON CHANGE block
- AFTER FIELD block

BEFORE INPUT block

The BEFORE INPUT block is executed one time, before the runtime system gives control to the user. You can implement initialization in this block.

AFTER INPUT block

The AFTER INPUT block is executed one time, after the user has validated or canceled the dialog, and before the runtime system executes the instruction that appears just after the INPUT block. You typically implement dialog finalization in this block. The AFTER INPUT block is not executed if EXIT INPUT executes.

BEFORE FIELD block

A BEFORE FIELD block is executed each time the cursor enters into the specified field, when moving the focus from field to field.

Such block is also executed when using NEXT FIELD.

ON CHANGE block

The ON CHANGE block is executed when another field is selected, if the value of the specified field has changed since the field got the focus and if the 'touched' field flag is set. The 'touched' flag is set on user input or when doing a DISPLAY TO or a DISPLAY BY NAME. Once set, the 'touched' flag is not reset until the end of the dialog.

For fields defined as RadioGroup, ComboBox, SpinEdit, Slider, and CheckBox views, the ON CHANGE block is fired immediately when the user changes the value. For other type of fields (like Edits), the ON CHANGE block is fired when leaving the field. You leave the field when you validate the dialog, when you move to another field, or when you

move to another row in an INPUT ARRAY. Note that the dialogtouched predefined action can also be used to detect field changes immediately, but with this action you can't get the data in the target variables (should only be used to detect that the user has started to modify data)

If both an `ON CHANGE` block and `AFTER FIELD` block are defined for a field, the `ON CHANGE` block is executed before the `AFTER FIELD` block.

When changing the value of the current field by program in an `ON ACTION` block, the `ON CHANGE` block will be executed when leaving the field if the value is different from the reference value and if the 'touched' flag is set (after previous user input or DISPLAY TO / DISPLAY BY NAME).

When using the NEXT FIELD instruction, the comparison value is re-assigned as if the user had leaved and re-entered the field. Therefore, when using `NEXT FIELD` in `ON CHANGE` block or in an `ON ACTION` block, the `ON CHANGE` block will only be fired again if the value is different from the reference value. This denies to do field validation in `ON CHANGE` blocks: you better do validations in `AFTER FIELD` blocks and/or `AFTER INPUT` blocks.

AFTER FIELD block

An `AFTER FIELD` block is executed each time the cursor leaves the specified field, when moving the focus from field to field.

The `AFTER FIELD` block is also executed when you validate the dialog or when you move to another row in an INPUT ARRAY.

Control Block Execution Order

The following table shows the order in which the runtime system executes the control blocks in the `INPUT` instruction, according to the user action:

Context / User action	Control Block execution order
Entering the dialog	1. `BEFORE INPUT` 2. `BEFORE FIELD` (first field)
Moving from field A to field B	1. `ON CHANGE` (if value has changed for field A) 2. `AFTER FIELD` (for field A) 3. `BEFORE FIELD` (for field B)
Changing the value of a field with a specific field like checkbox	1. `ON CHANGE`
Validating the dialog	1. `ON CHANGE` (if value has changed in current

	field) 2. AFTER FIELD 3. AFTER INPUT
Canceling the dialog	1. AFTER INPUT

Interaction Blocks

- ON IDLE block
- ON ACTION block
- ON KEY block

ON IDLE block

The ON IDLE *idle-seconds* clause defines a set of instructions that must be executed after *idle-seconds* of inactivity. This can be used, for example, to quit the dialog after the user has not interacted with the program for a specified period of time. The parameter *idle-seconds* must be an integer literal or variable. If it evaluates to zero, the timeout is disabled.

```
01 ...
02    ON IDLE 10
03       IF ask_question("Do you want to leave the dialog?") THEN
04          EXIT INPUT
05       END IF
06 ...
```

ON ACTION block

You can use ON ACTION blocks to execute a sequence of instructions when the user raises a specific action. This is the preferred solution compared to ON KEY blocks, because ON ACTION blocks use abstract names to control user interaction.

```
01 ...
02    ON ACTION zoom
03       CALL zoom_customers() RETURNING st, cust_id, cust_name
04       ...
```

ON KEY block

For backward compatibility, you can use ON KEY blocks to execute a sequence of instructions when the user presses a specific key. The following key names are accepted by the compiler:

Key Name	Description

`ACCEPT`	The validation key.
`INTERRUPT`	The interruption key.
`ESC` or `ESCAPE`	The ESC key (not recommended, use `ACCEPT` instead).
`TAB`	The TAB key (not recommended).
`Control-`*`char`*	A control key where *char* can be any character except A, D, H, I, J, K, L, M, R, or X.
`F1` through `F255`	A function key.
`DELETE`	The key used to delete a new row in an array.
`INSERT`	The key used to delete a new row in an array.
`HELP`	The help key.
`LEFT`	The left arrow key.
`RIGHT`	The right arrow key.
`DOWN`	The down arrow key.
`UP`	The up arrow key.
`PREVIOUS` or `PREVPAGE`	The previous page key.
`NEXT` or `NEXTPAGE`	The next page key.

Control Instructions

- CONTINUE INPUT instruction
- EXIT INPUT instruction
- ACCEPT INPUT instruction
- NEXT FIELD instruction
- CLEAR field-list instruction

Continuing the dialog: CONTINUE INPUT

`CONTINUE INPUT` skips all subsequent statements in the current control block and gives the control back to the dialog. This instruction is useful when program control is nested within multiple conditional statements, and you want to return the control to the dialog. Note that if this instruction is called in a control block that is not `AFTER INPUT`, further control blocks might be executed according to the context. Actually, `CONTINUE INPUT` just instructs the dialog to continue as if the code in the control block was terminated (i.e. it's a kind of `GOTO end_of_control_block`). However, when executed in `AFTER INPUT`, the focus returns to the most recently occupied field in the current form, giving the user another chance to enter data in that field. In this case the `BEFORE FIELD` of the current field will be fired.

Note that you can also use the `NEXT FIELD` control instruction to give the focus to a specific field and force the dialog to continue. However, unlike `CONTINUE INPUT`, the `NEXT FIELD` instruction will also skip the further control blocks that are normally executed.

Leaving the dialog: EXIT INPUT

You can use the `EXIT INPUT` statement to terminate the `INPUT` instruction and resume the program execution at the instruction following the `INPUT` block.

Validating the dialog: ACCEPT INPUT

The `ACCEPT INPUT` instruction validates the `INPUT` instruction and exits the `INPUT` instruction if no error is raised. The `AFTER FIELD`, `ON CHANGE`, etc. control blocks will be executed. Statements after the `ACCEPT INPUT` will not be executed.

Moving to a field: NEXT FIELD

The `NEXT FIELD` *`field-name`* instruction gives the focus to the specified field. You typically use this instruction to control field input dynamically, in `BEFORE FIELD` or `AFTER FIELD` blocks.

Abstract field identification is supported with the `CURRENT`, `NEXT` and `PREVIOUS` keywords. These keywords represent respectively the current, next and previous fields, corresponding to variables as defined in the input binding list (with the `FROM` or `BY NAME` clause).

Non-editable fields are fields defined with NOENTRY attribute or using a widget that does not allow input, such as a LABEL. If a `NEXT FIELD` instruction selects a non-editable field, the next editable field gets the focus (defined by the FIELD ORDER FORM mode used by the dialog). However, the `BEFORE FIELD` and `AFTER FIELD` blocks of non-editable fields are executed when a `NEXT FIELD` instruction selects such a field.

Clearing the form fields: CLEAR field-list

The `CLEAR` *`field-list`* instruction can be used to clear a specific field or all fields in a line of the screen record. You can specify the screen record as described in the following table:

CLEAR instruction	Result
`CLEAR` *`field-name`*	Clears the specified field.
`CLEAR` *`screen-record.*`*	Clears all fields members of the screen record.

Warning: When using the UNBUFFERED attribute, it is not recommended that you use the CLEAR instruction; always use program variables to set field values to NULL.

Control Class

Inside the dialog instruction, the predefined keyword `DIALOG` represents the current dialog object. It can be used to execute methods provided in the dialog built-in class.

For example, you can enable or disable an action with the ui.Dialog.setActionActive() dialog method, or you can hide or show the default action view with ui.Dialog.setActionHidden():

```
01 ...
02    BEFORE INPUT
03       CALL DIALOG.setActionActive("zoom",FALSE)
04    AFTER FIELD field1
05       CALL DIALOG.setActionHidden("zoom",1)
06 ...
```

The ui.Dialog.setFieldActive() method can be used to enable or disable a field during the dialog. This instruction takes an integer expression as argument.

```
01 ...
02    ON CHANGE custname
03       CALL DIALOG.setFieldActive( "custaddr", (rec.custname IS NOT
NULL) )
04 ...
```

Control Functions

The language provides several built-in functions and operators to use in a `INPUT` statement. For example: FIELD_TOUCHED(), FGL_DIALOG_GETFIELDNAME(), FGL_DIALOG_GETBUFFER().

Examples

Example 1: Simple INPUT statement using screen record specification

Form definition file (FormFile.per):

```
01 DATABASE stores
02
03 LAYOUT
04 GRID
05 {
06   Customer : [f001    ]
07   Name     : [f002                  ]
08   Last Name: [f003                  ]
09 }
10 END
```

```
END

TABLES
  customer
END

ATTRIBUTES
  f001 = customer.customer_num ;
  f002 = customer.fname, default = "<no name>", upshift ;
  f003 = customer.lname ;
END

INSTRUCTIONS
  SCREEN RECORD sr_cust(
    customer.customer_num,
    customer.fname,
    customer.lname);
END
```

Program source code:

```
MAIN

    DEFINE custrec RECORD
            id INTEGER,
            first_name CHAR(30),
            last_name CHAR(30)
    END RECORD

    OPTIONS INPUT WRAP

    OPEN FORM f FROM "FormFile"
    DISPLAY FORM f

    INPUT custrec.* FROM sr_cust.*

    IF INT_FLAG = FALSE THEN
       DISPLAY custrec.*
    END IF

END MAIN
```

Example 2: Complex INPUT statement using the BY NAME clause and control blocks

Form definition file (FormFile.per):

```
DATABASE stores

LAYOUT
GRID
{
  Customer : [f001    ]
  Name     : [f002                  ]
  Last Name: [f003                  ]
```

```
09 }
10 END
11 END
12
13 TABLES
14   customer
15 END
16
17 ATTRIBUTES
18   f001 = customer.customer_num ;
19   f002 = customer.fname, upshift ;
20   f003 = customer.lname ;
21 END
```

Program source code:

```
01 MAIN
02
03     DEFINE custrec RECORD
04             customer_num INTEGER,
05             fname CHAR(30),
06             lname CHAR(30)
07     END RECORD
09
10     OPTIONS INPUT WRAP
11
12     OPEN FORM f FROM "FormFile"
13     DISPLAY FORM f
14
15     LET custrec.customer_num = 0
16     LET custrec.fname = "<no name>"
17     LET custrec.lname = NULL
18     INPUT BY NAME custrec.* WITHOUT DEFAULTS
19        BEFORE INPUT
20          MESSAGE "Enter customer details..."
21        AFTER FIELD fname
23          IF FIELD_TOUCHED(custrec.fname)
23             AND custrec.fname IS NULL THEN
24             LET custrec.lname = NULL
25             DISPLAY BY NAME custrec.lname
26          END IF
27        BEFORE FIELD lname
28          IF NOT canEditLastName() THEN
29            NEXT FIELD fname
30          END IF
31        AFTER INPUT
32          MESSAGE "Input terminated..."
33     END INPUT
34
35     IF INT_FLAG = FALSE THEN
36        DISPLAY custrec.*
37     END IF
38
40 END MAIN
```

Array Display

Summary:

- Basics
  - The full list mode
  - The paged mode
- Syntax
- Usage
  - Programming Steps
  - Variable Binding
  - Instruction Configuration
  - Default Actions
  - Control Blocks
  - Control Blocks Execution Order
  - Interaction Blocks
  - Control Instructions
  - Control Class
  - Control Functions
- Scrolling Rows Up and Down (`SCROLL`)
- Examples
  - Full list mode example
  - Paged mode example

See also: Arrays, Records, Result Sets, Programs, Windows, Forms, Input Array

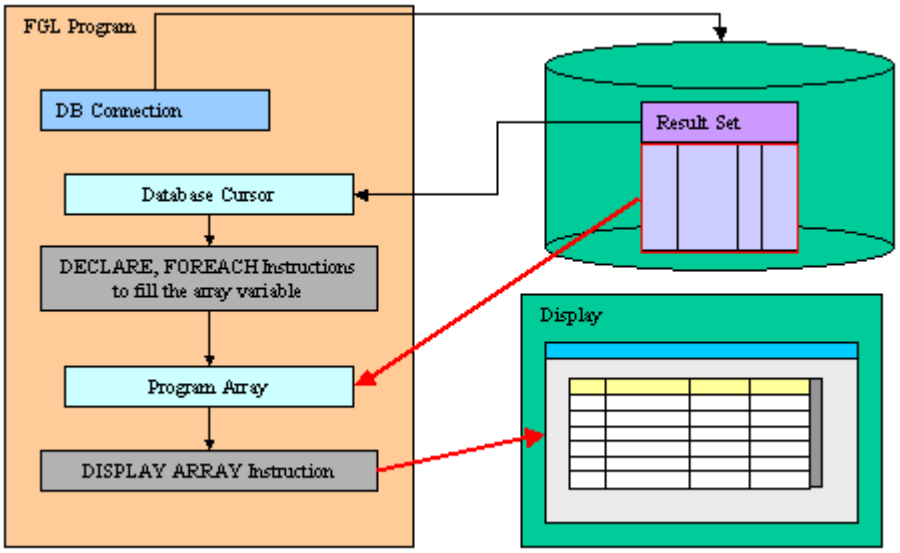
---

Basics

With `DISPLAY ARRAY`, you can let the user browse a list of records, using a static or dynamic array as the data buffer. The `DISPLAY ARRAY` instruction can work in *full list* mode or in *paged* mode. In *full list* mode, you must copy all the data you want to display into the array. In *paged* mode, you provide data rows dynamically during the dialog, using a dynamic array to hold one page of data. The *full list* mode should be used for a short and static list of rows, while the *paged* mode can be used for an infinite number of rows. Additionally, the *paged* mode allows you to fetch fresh data from the database.

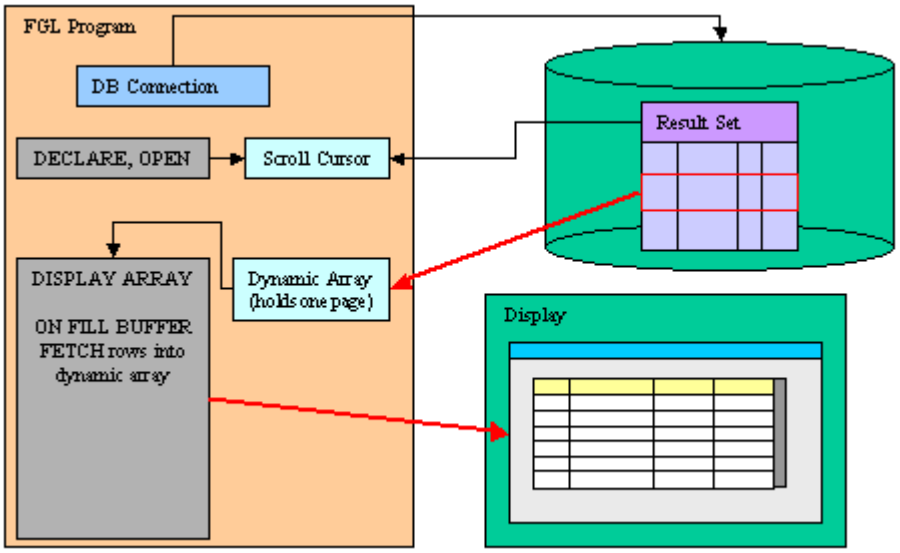
The full list mode

In *full list* mode, the `DISPLAY ARRAY` instruction uses a static or dynamic program array defined with a record structure corresponding to (or to a part of) a screen-array of a form. The program array is filled with data rows before `DISPLAY ARRAY` is executed. In this case, the list is static and cannot be updated until the instruction is exited.



The paged mode

In *paged* mode, the `DISPLAY ARRAY` instruction uses a dynamic program array defined with a record structure corresponding to (or to a part of) a screen-array of a form. The total number of rows is defined by the `COUNT` attribute. The program array is filled dynamically with data rows *as needed* during the `DISPLAY ARRAY` execution. The `ON FILL BUFFER` clause is required, to feed the `DISPLAY ARRAY` instruction with pages of data. The statements in the `ON FILL BUFFER` clause are executed automatically by the runtime system each time a new page of data is needed.



Warnings:

1. With paged mode, the user cannot sort data by clicking on a column header.

DISPLAY ARRAY

Purpose:

The `DISPLAY ARRAY` instruction controls the display of a program array on the screen.

Syntax:

```
DISPLAY ARRAY array TO screen-array.*
  [ HELP help-number ]
  [ ATTRIBUTES ( { display-attribute | control-attribute } [,...] ) ]
[ dialog-control-block
  [...]
END DISPLAY ]
```

where *dialog-control-block* is one of :

```
{ BEFORE DISPLAY
| AFTER DISPLAY
| BEFORE ROW
| AFTER ROW
| ON IDLE idle-seconds
| ON ACTION action-name
| ON FILL BUFFER
| ON KEY ( key-name [,...] )
}
    dialog-statement
    [...]
```

where *dialog-statement* is one of :

```
{ statement
| EXIT DISPLAY
| CONTINUE DISPLAY
| ACCEPT DISPLAY
}
```

Notes:

1. *array* is a static or dynamic array containing the records you want to display.
2. *screen-array* is the name of the screen array used to display data.
3. *help-number* is an integer that associates a help message number with the instruction.
4. *key-name* is an hot-key identifier (such as `F11` or `Control-z`).
5. *action-name* identifies an action that can be executed by the user.
6. *idle-seconds* is an integer literal or variable that defines a number of seconds.
7. *statement* is any instruction supported by the language.

The following table shows the *display-attributes* supported by the DISPLAY ARRAY statement. The *display-attributes* affect console-based applications only, they do not affect GUI-based applications.

Attribute	Description
BLACK, BLUE, CYAN, GREEN, MAGENTA, RED, WHITE, YELLOW	The color of the displayed data.
BOLD, DIM, NORMAL	The font attribute of the displayed data.
REVERSE, BLINK, UNDERLINE	The video attribute of the displayed data.

The following table shows the *control-attributes* supported by the DISPLAY ARRAY statement:

Attribute	Description
COUNT = *row-count*	Defines the number of data rows when using a static array or the paged mode. *row-count* can be an integer literal or a program variable. This is the equivalent of the SET_COUNT() built-in function.
HELP = *int-expr*	Defines the help number when help is invoked by the user.
KEEP CURRENT ROW [=*bool*]	Keeps current row highlighted after execution of the instruction.
UNBUFFERED [=*bool*]	Indicates that the dialog must be sensitive to program variable changes. The *bool* parameter can be an integer literal or a program variable.
CANCEL = *bool*	Indicates if the default *cancel* action should be added to the dialog. If not specified, the action is registered.
ACCEPT = *bool*	Indicates if the default *accept* action should be added to the dialog. If not specified, the action is registered.

Usage:

Programming Steps

The following steps describe how to use the DISPLAY ARRAY statement:

1. Create a form specification file containing a screen array. The screen array identifies the presentation elements to be used by the runtime system to display the rows.

2. Make sure that the program controls interruption handling with DEFER INTERRUPT, to manage the validation/cancellation of the interactive dialog.
3. Define an array of records with the DEFINE instruction. The members of the program array must correspond to the elements of the screen array, by number and data types. You can use a static or a dynamic array for a full list mode, but you must use a dynamic array for a paged mode.
4. Open and display the form, using an OPEN WINDOW with the `WITH FORM` clause or the OPEN FORM / DISPLAY FORM instructions.
5. If you want to use the full list mode, fill the program array with data, for example with a result set cursor, counting the number of program records being filled with retrieved data.
6. Use the `DISPLAY ARRAY` statement to display the values. When using a static array, specifying the number of rows with the `COUNT` attribute in the `ATTRIBUTES` clause.
7. If you want to use the paged mode, add the `ON FILL BUFFER` clause inside the instruction, and write the code to fill the dynamic array with the expected rows from `fgl_dialog_getBufferStart()` to `fgl_dialog_getBufferLength()`.
8. Inside the `DISPLAY ARRAY` statement, control the behavior of the selection list with `BEFORE DISPLAY`, `BEFORE ROW`, `AFTER ROW`, `AFTER DISPLAY` and `ON KEY` blocks.
9. After the `DISPLAY ARRAY` statement, test the INT_FLAG predefined variable to check if the interactive dialog was canceled (INT_FLAG = TRUE) or validated (INT_FLAG = FALSE).
10. If needed, get the selected row with the ARR_CURR() built-in function.

Tips:

1. If you want to display data to a reduced set of columns, you must define a second screen array in the form file, containing the limited list of form fields. Then you can use the second screen array in a `DISPLAY ARRAY a TO sa.*` instruction.

Warnings:

1. The `INVISIBLE` attribute is ignored.
2. While the `ON KEY` block is supported for backward compatibility, it is recommended that you use `ON ACTION` instead.

Variable Binding

The `DISPLAY ARRAY` statement binds the members of the array of record to the screen array fields specified with the `TO` keyword. The number of variables in each record of the program array must be the same as the number of fields in each screen record (that is, in a single row of the screen array).

When using the `UNBUFFERED` attribute, the instruction is sensitive to program variable changes. This means that you do not have to `DISPLAY` the values; setting the program

variable used by the dialog automatically displays the data into the corresponding form field.

```
01 ...
02    ON ACTION change
03       LET arr[arr_curr()].field1 = newValue()
04 ...
```

Instruction Configuration

The ATTRIBUTES clause specifications override all default attributes and temporarily override any display attributes that the OPTIONS or the OPEN WINDOW statement specified for these fields. While the DISPLAY ARRAY statement is executing, the runtime system ignores the INVISIBLE attribute.

- HELP option
- COUNT option
- KEEP CURRENT ROW option
- ACCEPT option
- CANCEL option

HELP option

The HELP clause specifies the number of a help message to display if the user invokes the help while executing the instruction. The predefined *help* action is automatically created by the runtime system. You can bind action views to the 'help' action.

Warnings:

1. The HELP *option* overrides the HELP *attribute*!

COUNT option

When using a static array or the paged mode, the number of rows to be displayed is defined by the COUNT attribute. You can also use the SET_COUNT() built-in function, but it is supported for backward compatibility only. When using a dynamic array, the number of rows to be displayed is defined by the number of elements in the dynamic array; the COUNT attribute is ignored.

KEEP CURRENT ROW option

Depending on the list container used in the form, the current row may be highlighted during the execution of the dialog, and cleared when the instruction ends. You can change this default behavior by using the KEEP CURRENT ROW attribute, to force the runtime system to keep the current row highlighted.

ACCEPT option

The ACCEPT attribute can be set to FALSE to avoid the automatic creation of the *accept* default action. Use this attribute when you want to write a specific validation procedure by using ACCEPT DISPLAY.

CANCEL option

The CANCEL attribute can be set to FALSE to avoid the automatic creation of the *cancel* default action. Use this attribute when you only need a validation action (accept), or when you want to write a specific cancellation procedure by using EXIT DISPLAY.

Note that if the CANCEL=FALSE option is set, no *close* action will be created, and you must write an ON ACTION close control block to create an explicit action.

Default Actions

When an DISPLAY ARRAY instruction executes, the runtime system creates a set of default actions. See the control block execution order to understand what control blocks are executed when a specific action is fired.

The following table lists the default actions created for this dialog:

Default action	Description
accept	Validates the DISPLAY ARRAY dialog (validates current row selection) *Creation can be avoided with ACCEPT attribute.*
cancel	Cancels the DISPLAY ARRAY dialog (no validation, INT_FLAG is set) *Creation can be avoided with CANCEL attribute.*
close	By default, cancels the DISPLAY ARRAY dialog (no validation, INT_FLAG is set) Default action view is hidden. See Windows closed by the user.
help	Shows the help topic defined by the HELP clause. *Only created when a HELP clause is defined.*
firstrow	Moves to the first row in the list.
lastrow	Moves to the last row in the list.
nextrow	Moves to the next row in the list.
prevrow	Moves to the previous row in the list.

The accept and cancel default actions can be avoided with the ACCEPT and CANCEL dialog control attributes:

```
01  DISPLAY ARRAY arr TO sr.* ATTRIBUTES( CANCEL=FALSE, ... )
02        ...
```

Control Blocks

- BEFORE DISPLAY block
- AFTER DISPLAY block
- BEFORE ROW block
- AFTER ROW block
- ON FILL BUFFER block

BEFORE DISPLAY block

The `BEFORE DISPLAY` block is executed one time, before the runtime system gives control to the user. You can implement dialog initialization tasks in this block.

AFTER DISPLAY block

The `AFTER DISPLAY` block is executed one time, after the user has validated or canceled the dialog and before the runtime system executes the instruction that appears just after the `DISPLAY ARRAY` block. You typically implement dialog finalization in this block.

BEFORE ROW block

The `BEFORE ROW` block is executed each time the user moves to another row, after the destination row is made the current one. When called in this block, the ARR_CURR() function returns the index of the current row.

AFTER ROW block

The `AFTER ROW` block is executed each time the user moves to another row, before the current row is left. When called in this block, the ARR_CURR() function returns the index of the current row.

ON FILL BUFFER block

The `ON FILL BUFFER` clause is provided to fill a page of rows into the dynamic array, according to an offset and a number of rows. The offset can be retrieved with the FGL_DIALOG_GETBUFFERSTART() built-in function and the number of rows to provide is defined by the FGL_DIALOG_GETBUFFERLENGTH() built-in function.

A typical paged display array consists of a scroll cursor providing the list of records to be displayed. Scroll cursors use a static result set. If you want to display fresh data, you can write advanced paged display array instructions by using a scroll cursor providing the primary keys of the reference result set, plus a prepared cursor used to fetch rows on demand in the `ON FILL BUFFER` clause. In this case, you may need to check if a row still exists when fetching a record with the second cursor.

See Example 2 for a typical paged mode implementation.

Control Block Execution Order

The following table shows the order in which the runtime system executes the control blocks in the `DISPLAY ARRAY` instruction, according to the user action:

Context / User action	Control Block execution order
Entering the dialog	1. `BEFORE DISPLAY` 2. `BEFORE ROW`
Moving to a different row	1. `AFTER ROW` (the current row) 2. `BEFORE ROW` (the new row)
Validating the dialog	1. `AFTER ROW` 2. `AFTER DISPLAY`
Canceling the dialog	1. `AFTER ROW` 2. `AFTER INPUT`

Interaction Blocks

- ON IDLE block
- ON ACTION block
- ON KEY block

ON IDLE block

The `ON IDLE` *`idle-seconds`* clause defines a set of instructions that must be executed after *idle-seconds* of inactivity. This can be used, for example, to quit the dialog after the user has not interacted with the program for a specified period of time. The parameter *idle-seconds* must be an integer literal or variable. If it evaluates to zero, the timeout is disabled.

```
01 ...
02    ON IDLE 10
03       IF ask_question("Do you want to leave the dialog?") THEN
04          EXIT DISPLAY
05       END IF
06 ...
```

ON ACTION block

You can use `ON ACTION` blocks to execute a sequence of instructions when the user raises a specific action. This is the preferred solution compared to `ON KEY` blocks, because `ON ACTION` blocks use abstract names to control user interaction.

```
01 ...
```

```
02    ON ACTION zoom
03       CALL zoom_customers() RETURNING st, cust_id, cust_name
04       ...
```

ON KEY block

For backward compatibility, you can use `ON KEY` blocks to execute a sequence of instructions when the user presses a specific key. The following key names are accepted by the compiler:

Key Name	Description
`ACCEPT`	The validation key.
`INTERRUPT`	The interruption key.
`ESC` or `ESCAPE`	The ESC key (not recommended, use `ACCEPT` instead).
`TAB`	The TAB key (not recommended).
`Control-`*`char`*	A control key where *char* can be any character except A, D, H, I, J, K, L, M, R, or X.
`F1` through `F255`	A function key.
`DELETE`	The key used to delete a new row in an array.
`INSERT`	The key used to insert a new row in an array.
`HELP`	The help key.
`LEFT`	The left arrow key.
`RIGHT`	The right arrow key.
`DOWN`	The down arrow key.
`UP`	The up arrow key.
`PREVIOUS` or `PREVPAGE`	The previous page key.
`NEXT` or `NEXTPAGE`	The next page key.

Control Instructions

- CONTINUE DISPLAY instruction
- EXIT DISPLAY instruction
- ACCEPT DISPLAY instruction

Continuing the dialog: CONTINUE DISPLAY

`CONTINUE DISPLAY` skips all subsequent statements in the current control block and gives the control back to the dialog. This instruction is useful when program control is nested within multiple conditional statements, and you want to return the control to the dialog. Note that if this instruction is called in a control block that is not `AFTER DISPLAY`, further control blocks might be executed according to the context. Actually, `CONTINUE DISPLAY` just instructs the dialog to continue as if the code in the control block was terminated (i.e. it's a kind of `GOTO end_of_control_block`). However, when executed in `AFTER DISPLAY`, the focus returns to the current row in the list, giving the user another

chance to browse and select a row. In this case the `BEFORE ROW` of the current row will be fired.

Leaving the dialog: EXIT DISPLAY

You can use the `EXIT DISPLAY` to terminate the `DISPLAY ARRAY` instruction and resume the program execution at the instruction immediately following the `DISPLAY ARRAY` block.

Validating the dialog: ACCEPT DISPLAY

The `ACCEPT DISPLAY` instruction validates the `DISPLAY ARRAY` instruction and exits the `DISPLAY ARRAY` instruction. The `AFTER DISPLAY` control block will be executed. Statements after `ACCEPT DISPLAY` will not be executed.

Control Class

Inside the dialog instruction, the predefined keyword `DIALOG` represents the current dialog object. It can be used to execute methods provided in the DIALOG built-in class.

For example, you can enable or disable an action with the ui.Dialog.setActionActive() dialog method, or you can hide and show a default action view with ui.Dialog.setActionHidden():

```
01 ...
02    BEFORE DISPLAY
03       CALL DIALOG.setActionActive("refresh",FALSE)
```

Control Functions

The language provides several built-in functions and operators to use in a `DISPLAY ARRAY` statement. You can use the following built-in functions to keep track of the relative states of the current row, the program array, and the screen array or to access the field buffers and keystroke buffers. These functions and operators are:

- ARR_CURR()
- FGL_SET_ARR_CURR()
- ARR_COUNT()
- SCR_LINE()
- SET_COUNT()

SCROLL

Purpose:

The SCROLL instruction specifies vertical movements of displayed values in all or some of the fields of a screen array within the current form.

Syntax:

```
SCROLL field-list { UP | DOWN } [ BY lines ]
```

where *field-list* is :

```
{ field-name
| table-name.*
| table-name.field-name
| screen-array[line].*
| screen-array[line].field-name
| screen-record.*
| screen-record.field-name
} [,...]
```

Notes:

1. *field-name* is the identifier of a field of the current form.
2. *table-name* is the identifier of a database table of the current form.
3. *screen-record* is the identifier of a screen record of the current form.
4. *screen-array* is the name of the screen array used of the current form.
5. *lines* is an integer literal or variables that specifies how far (in lines) to scroll the display.

Warnings:

1. It is recommended that you NOT use this instruction in GUI mode.

Examples

Example 1: Full list mode

Form definition file "custlist.per":

```
01 DATABASE stores
02
03 LAYOUT
04 TABLE
05 {
06  Id       Name         LastName
07 [f001    |f002        |f003        ]
```

```
[f001    |f002        |f003        ]
[f001    |f002        |f003        ]
[f001    |f002        |f003        ]
[f001    |f002        |f003        ]
[f001    |f002        |f003        ]
}
END
END

TABLES
customer
END

ATTRIBUTES
f001 = customer.customer_num;
f002 = customer.fname;
f003 = customer.lname;
END

INSTRUCTIONS
DELIMITERS "||";
SCREEN RECORD srec[6] (
       customer.customer_num,
       customer.fname,
       customer.lname);
END
```

Application:

```
MAIN
  DEFINE cnt INTEGER
  DEFINE arr ARRAY[500] OF RECORD
             id INTEGER,
             fname CHAR(30),
             lname CHAR(30)
         END RECORD

  DATABASE stores7

  OPEN FORM f1 FROM "custlist"
  DISPLAY FORM f1

  DECLARE c1 CURSOR FOR
    SELECT customer_num, fname, lname FROM customer
  LET cnt = 1
  FOREACH c1 INTO arr[cnt].*
    LET cnt = cnt + 1
  END FOREACH
  LET cnt = cnt - 1
  DISPLAY ARRAY arr TO srec.* ATTRIBUTES(COUNT=cnt)
    ON ACTION print
       DISPLAY "Print a report"
  END DISPLAY
END MAIN
```

Example 2: Paged mode

Form definition file "custlist.per" (same as example 1)

Application:

```
01 MAIN
02   DEFINE arr DYNAMIC ARRAY OF RECORD
03              id INTEGER,
04              fname CHAR(30),
05              lname CHAR(30)
06          END RECORD
07   DEFINE cnt, ofs, len, i INTEGER
08
09   DATABASE stores7
10
11   OPEN FORM f1 FROM "custlist"
12   DISPLAY FORM f1
13
14   SELECT COUNT(*) INTO cnt FROM customer
15   DECLARE c1 SCROLL CURSOR FOR
16          SELECT customer_num, fname, lname FROM customer
17   DISPLAY ARRAY arr TO srec.* ATTRIBUTES(COUNT=cnt)
18     ON FILL BUFFER
19        OPEN c1
20        LET ofs = fgl_dialog_getBufferStart()
21        LET len = fgl_dialog_getBufferLength()
22        -- Warning: Fill the array from index 1 to len!
23        FOR i=1 TO len
24           FETCH ABSOLUTE ofs+i-1 c1 INTO arr[i].*
25        END FOR
26        CLOSE c1
27     AFTER DISPLAY
28        IF NOT int_flag THEN
29           DISPLAY "Selected customer is #"
30                   || arr[arr_curr()-ofs+1].id
31        END IF
32   END DISPLAY
33 END MAIN
```

Array Input

Summary:

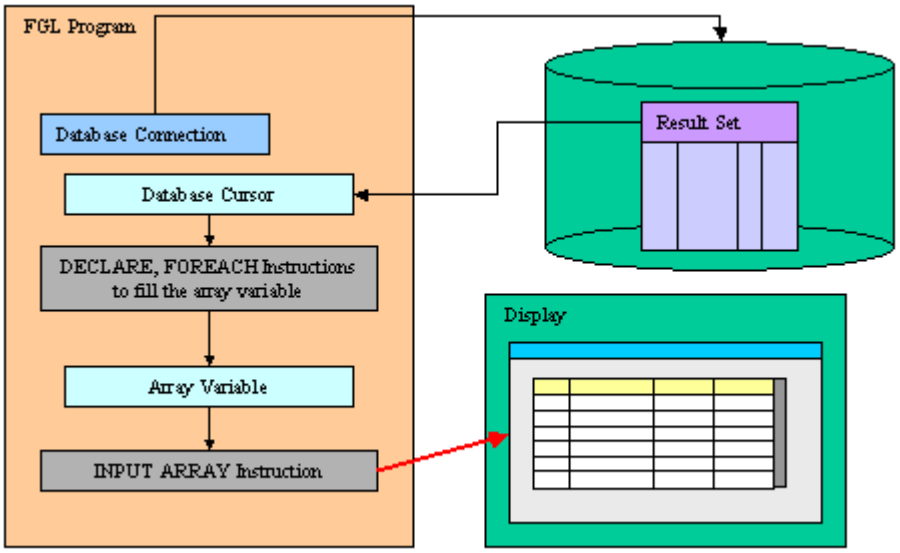
- Basics
- Syntax
- Usage
  - Programming Steps
  - Variable Binding
  - Instruction Configuration
  - Default Actions
  - Control Blocks
  - Control Blocks Execution Order
  - Interaction Blocks
  - Control Instructions
  - Control Class
  - Control Functions
- Examples
  - Basic INPUT ARRAY
  - INPUT ARRAY using default values
  - INPUT ARRAY using dynamic array
  - INPUT ARRAY updating database table

See also: Arrays, Records, Result Sets, Programs, Windows, Forms, Display Array

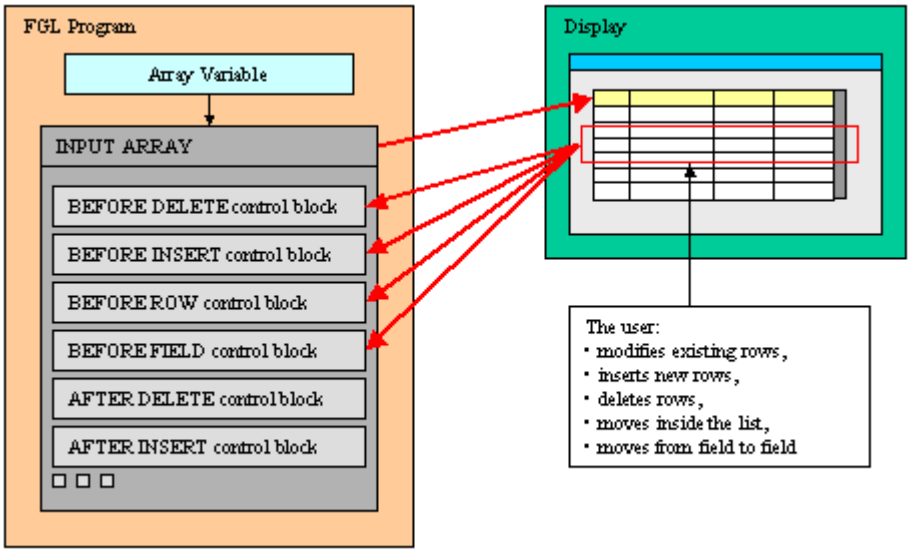
---

Basics

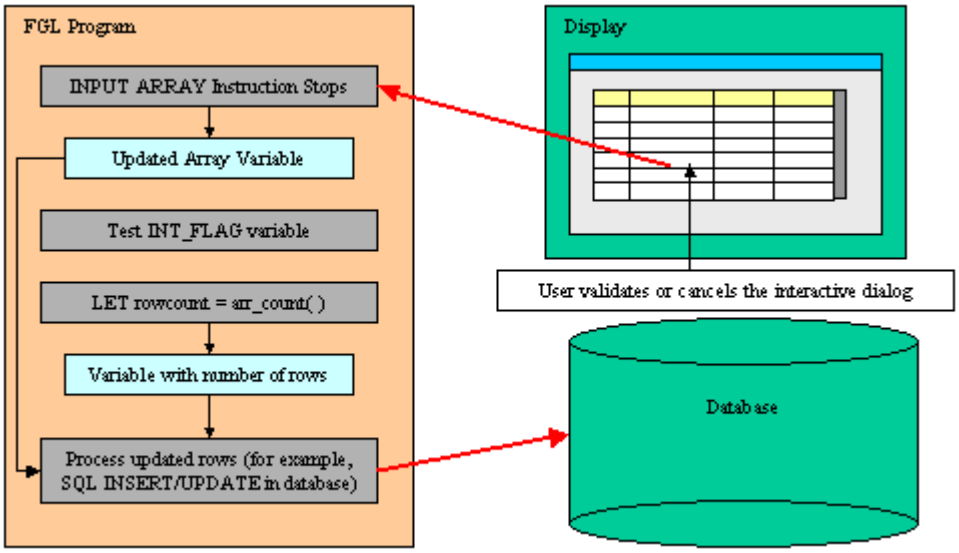
The `INPUT ARRAY` instruction associates a program array of records with a screen-array defined in a form so that the user can update the list of records. The `INPUT ARRAY` statement activates the current form (the form that was most recently displayed or the form in the current window):



During the `INPUT ARRAY` execution, the user can edit or delete existing rows, insert new rows, and move inside the list of records. The user can insert new rows with the insert key, which is by default `F1`, or delete existing rows with the delete key, which is by default `F2`. The program controls the behavior of the instruction with control blocks:



To terminate the `INPUT ARRAY` execution, the user can validate (or cancel) the dialog to commit (or invalidate) the modifications made in the list of records:



When the statement completes execution, the form is de-activated. After the user terminates the input (for example, with the Accept key), the program must test the INT_FLAG variable to check if the dialog was validated (or canceled) and then use INSERT, DELETE, or UPDATE SQL statements to modify the appropriate database tables. The database can also be updated during the execution of the `INPUT ARRAY` statement.

INPUT ARRAY

Purpose:

The `INPUT ARRAY` supports data entry by users into a screen array and stores the entered data in an array of records.

Syntax:

```
INPUT ARRAY array
  [ WITHOUT DEFAULTS ]
  FROM screen-array.*
  [ ATTRIBUTES ( { display-attribute | control-attribute } [,...] ) ]
  [ HELP help-number ]
[ dialog-control-block
  [...]
END INPUT ]
```

where *dialog-control-block* is one of:

```
{ BEFORE INPUT
| AFTER INPUT
| AFTER DELETE
| BEFORE ROW
```

```
| AFTER ROW
| BEFORE FIELD field-spec
| AFTER FIELD field-spec
| ON ROW CHANGE
| ON CHANGE field-spec
| ON IDLE idle-seconds
| ON ACTION action-name
| ON KEY ( key-name [,...] )
| BEFORE INSERT
| AFTER INSERT
| BEFORE DELETE
}
    dialog-statement
    [...]
```

where *dialog-statement* is one of:

```
{ statement
| ACCEPT INPUT
| CONTINUE INPUT
| EXIT INPUT
| NEXT FIELD { CURRENT | NEXT | PREVIOUS | field-name }
| CANCEL DELETE
| CANCEL INSERT
}
```

where *field-spec* is:

```
{ field-name
| table-name.field-name
| screen-array.field-name
| screen-record.field-name
} [,...]
```

Notes:

1. *array* is the array of records that will be filled by the `INPUT ARRAY` statement.
2. *help-number* is an integer that allows you to associate a help message number with the instruction.
3. *field-name* is the identifier of a field of the current form.
4. *table-name* is the identifier of a database table of the current form.
5. *screen-record* is the identifier of a screen record of the current form.
6. *screen-array* is the screen array that will be used in the form.
7. *action-name* identifies an action that can be executed by the user.
8. *idle-seconds* is an integer literal or variable that defines a number of seconds.
9. *key-name* is a hot-key identifier (like `F11` or `Control-z`).
10. *statement* is any instruction supported by the language.

The following table shows the *options* supported by the `INPUT ARRAY` statement:

Attribute	Description
`HELP help-number`	Defines the help number when help is

	invoked by the user, where *help-number* is an integer literal or a program variable. **See Warning below!**
`WITHOUT DEFAULTS`	Indicates that the data rows are not filled (TRUE) with the column default values defined in the form specification file or the database schema files (Default is FALSE). **See Warning below!**

The following table shows the *display-attributes* supported by the `INPUT ARRAY` statement. The *display-attributes* affect console-based applications only, they do not affect GUI-based applications.

Attribute	Description
`BLACK, BLUE, CYAN, GREEN, MAGENTA, RED, WHITE, YELLOW`	The color of the displayed data.
`BOLD, DIM, INVISIBLE, NORMAL`	The font attribute of the displayed data.
`REVERSE, BLINK, UNDERLINE`	The video attribute of the displayed data.

The following table shows the *control-attributes* supported by the `INPUT ARRAY` statement:

Attribute	Description
`HELP = help-number`	Defines the help number when help is invoked by the user, where *help-number* is an integer literal or a program variable. **See Warning below!**
`WITHOUT DEFAULTS [=bool]`	Indicates if the data rows must be filled (FALSE) or not (TRUE) with the column default values defined in the form specification file or the database schema files. The *bool* parameter can be an integer literal or a program variable that evaluates to TRUE or FALSE. **See Warning below!**
`FIELD ORDER FORM`	Indicates that the tabbing order of fields is defined by the TABINDEX attribute of form fields. The default order in which the focus moves from field to field in the screen array is determined by the order of the members in the array variable used by the `INPUT ARRAY` statement. The program options instruction can also change this behavior with `FIELD ORDER FORM` options.
`UNBUFFERED [ =bool]`	Indicates that the dialog must be sensitive to

	program variable changes. The *bool* parameter can be an integer literal or a program variable.
`COUNT = row-count`	Defines the number of data rows in the static array. The *row-count* can be an integer literal or a program variable. This is the equivalent of the SET_COUNT() built-in function.
`MAXCOUNT = row-count`	Defines the maximum number of data rows that can be entered in the program array, where *row-count* can be an integer literal or a program variable.
`ACCEPT = bool`	Indicates if the default *accept* action should be added to the dialog. If not specified, the action is registered. The *bool* parameter can be an integer literal or a program variable.
`CANCEL = bool`	Indicates if the default *cancel* action should be added to the dialog. If not specified, the action is registered. The *bool* parameter can be an integer literal or a program variable.
`APPEND ROW [ =bool]`	Defines if the user can append new rows at the end of the list. The *bool* parameter can be an integer literal or a program variable that evaluates to TRUE or FALSE.
`INSERT ROW [ =bool]`	Defines if the user can insert new rows inside the list. The *bool* parameter can be an integer literal or a program variable that evaluates to TRUE or FALSE.
`DELETE ROW [ =bool]`	Defines if the user can delete rows. The *bool* parameter can be an integer literal or a program variable that evaluates to TRUE or FALSE.
`AUTO APPEND [ =bool]`	Defines if a temporary row will be created automatically when needed. The *bool* parameter can be an integer literal or a program variable that evaluates to TRUE or FALSE.
`KEEP CURRENT ROW [=bool]`	Keeps current row highlighted after execution of the instruction.

Usage

Programming Steps

The following steps describe how to use the `INPUT ARRAY` statement:

1. Create a form specification file containing a screen array. The screen array identifies the presentation elements to be used by the runtime system to display the rows.
2. Make sure that the program controls interruption handling with DEFER INTERRUPT, to manage the validation/cancellation of the interactive dialog.
3. Define an array of records with the DEFINE instruction. The members of the program array must correspond to the elements of the screen array, by number and data types. If you want to input data from a reduced set of columns, you must define a second screen array, containing the limited list of form fields, in the form file. You can then use the second screen array in an `INPUT ARRAY a FROM sa.*` instruction.
4. Open and display the form, using a OPEN WINDOW with the `WITH FORM` clause or the OPEN FORM / DISPLAY FORM instructions.
5. If needed, fill the program array with data, for example with a result set cursor, counting the number of program records being filled with retrieved data.
6. Inside the `INPUT ARRAY` statement, control the behavior of the instruction with control blocks such as `BEFORE INPUT`, `BEFORE INSERT`, `BEFORE DELETE`, `BEFORE ROW`, `BEFORE FIELD`, `AFTER INSERT`, `AFTER DELETE`, `AFTER FIELD`, `AFTER ROW`, `AFTER INPUT` and `ON ACTION` blocks.
7. After the `INPUT ARRAY` statement, test the INT_FLAG pre-defined variable to check if the interactive dialog was canceled (INT_FLAG = TRUE) or validated (INT_FLAG = FALSE).
8. Get the new number of rows with the ARR_COUNT() built-in function.
9. If needed, get the current row with the ARR_CURR() built-in function.

Warnings:

1. You can only use the `CANCEL INSERT` keyword in the `BEFORE INSERT` or `AFTER INSERT` blocks.
2. You can only use the `CANCEL DELETE` keyword in the `BEFORE DELETE` block.
3. For the `HELP` and `WITHOUT DEFAULTS` options, the order of appearance is important : **the last option overrides the previous one!**

Variable Binding

The `INPUT ARRAY` statement binds the members of the array of record to the screen array fields specified with the `FROM` keyword. The number of variables in each record of the program array must be the same as the number of fields in each screen record (that is, in a single row of the screen array).

When using a static array, the initial number of rows is defined by the `COUNT` attribute and the size of the array determines how many rows can be inserted. When using a dynamic array, the initial number of rows is defined by the number of elements in the dynamic array (the `COUNT` attribute is ignored). The size of the input array is unlimited. In both cases, the maximum number of rows the user can enter can be defined with the `MAXCOUNT` attribute.

The FROM clause binds the screen records in the screen array to the program records of the program array. The form can include other fields that are not part of the specified screen array, but the number of member variables in each record of the program array must equal the number of fields in each row of the screen array. When the user enters data, the runtime system checks the entered value against the data type of the variable, not the data type of the screen field.

The variables of the record array are the interface to display data or to get the user input through the INPUT ARRAY instruction. Always use the variables if you want to change some field values programmatically. When using the UNBUFFERED attribute, the instruction is sensitive to program variable changes. If you need to display new data during the INPUT ARRAY execution, use the UNBUFFERED attribute and assign the values to the program array row; the runtime system will automatically display the values to the screen:

```
01 INPUT ARRAY p_items FROM s_items.* ATTRIBUTES(UNBUFFERED)
02    ON CHANGE code
03       IF p_items[arr_curr()].code = "A34" THEN
04          LET p_items[arr_curr()].desc = "Item A34"
05       END IF
06 END INPUT
```

The member variables of the records in a program array can be of any data type. If a variable is declared LIKE a SERIAL column, the runtime system does not allow the screen cursor to stop in the field, as values in SERIAL columns are automatically generated by the database server, not by the runtime system.

The default order in which the focus moves from field to field in the screen array is determined by the declared order of the corresponding member variables, in the array of the record definition. The program options instruction can also change the behavior of the INPUT ARRAY instruction, with the INPUT WRAP or FIELD ORDER FORM options.

When the INPUT ARRAY instruction executes, it displays any column default values in the screen fields, unless you specify the WITHOUT DEFAULTS keywords. The column default values are specified in the form specification file with the DEFAULT attribute or in the database schema files.

Instruction Configuration

The ATTRIBUTES clause specifications override all default attributes and temporarily override any display attributes that the OPTIONS or the OPEN WINDOW statement specified for these fields. While the INPUT ARRAY statement is executing, the runtime system ignores the INVISIBLE attribute.

- HELP option
- WITHOUT DEFAULTS option
- FIELD ORDER FORM option
- UNBUFFERED option

- COUNT option
- MAXCOUNT option
- ACCEPT option
- CANCEL option
- APPEND ROW option
- INSERT ROW option
- DELETE ROW option
- AUTO APPEND option
- KEEP CURRENT ROW option

HELP option

The `HELP` clause specifies the number of a help message to display if the user invokes the help while the focus is in any field used by the instruction. The predefined *help* action is automatically created by the runtime system. You can bind action views to the 'help' action.

Warnings:

1. The HELP *option* overrides the HELP *attribute*!

WITHOUT DEFAULTS option

Indicates if the data rows must be filled (FALSE) or not (TRUE) with the column default values defined in the form specification file or the database schema files. The *bool* parameter can be an integer literal or a program variable that evaluates to TRUE or FALSE.

FIELD ORDER FORM option

By default, the form tabbing order is defined by the variable list in the binding specification. You can control the tabbing order by using the `FIELD ORDER FORM` attribute. When this attribute is used, the tabbing order is defined by the TABINDEX attribute of the form items. With `FIELD ORDER FORM`, if you jump from one field to a another with the mouse, the `BEFORE FIELD` / `AFTER FIELD` triggers of intermediate fields are not executed (actually, the Dialog.fieldOrder FGLPROFILE entry is ignored)

If the form uses a TABLE container, the front-end resets the tab indexes when the user moves columns around. This way, the visual column order always corresponds to the input tabbing order. The order of the columns in an editable list can be important; you may want to freeze the table columns with the UNMOVABLECOLUMNS attribute.

UNBUFFERED option

The `UNBUFFERED` attribute indicates that the dialog must be sensitive to program variable changes. When using this option, you bypass the traditional "buffered" mode.

When using the traditional " buffered" mode, program variable changes are not automatically displayed to form fields; You need to execute a `DISPLAY TO` or `DISPLAY BY NAME`. Additionally, if an action is triggered, the value of the current field is not validated and is not copied into the corresponding program variable. The only way to get the text of the current field is to use GET_FLDBUF().

If the "unbuffered" mode is used, program variables and form fields are automatically synchronized. You don't need to display explicitly values with a `DISPLAY TO` or `DISPLAY BY NAME`. When an action is triggered, the value of the current field is validated and is copied into the corresponding program variable.

COUNT option

The `COUNT` attribute defines the number of valid rows in the static array to be displayed as default rows. If you do not use the `COUNT` attribute, the runtime system cannot determine how much data to display, so the screen array remains empty. You can also use the SET_COUNT() built-in function, but it is supported for backward compatibility only. The `COUNT` option is ignored when using a dynamic array. If you specify the `COUNT` attribute, the `WITHOUT DEFAULTS` option is not required because it is implicit. If the `COUNT` attribute is greater as `MAXCOUNT`, the runtime system will take `MAXCOUNT` as the actual number of rows. If the value of `COUNT` is negative or zero, it defines an empty list.

MAXCOUNT option

The `MAXCOUNT` attribute defines the maximum number of rows that can be inserted in the program array. This attribute allows you to give an upper limit. When using a dynamic array, the user can enter an infinite number of rows unless the `MAXCOUNT` attribute is used. When using a static array and `MAXCOUNT` is greater than the size of the declared static array, the original static array size is used as the upper limit. If `MAXCOUNT` is negative or equal to zero, user cannot insert rows.

ACCEPT option

The `ACCEPT` attribute can be set to FALSE to avoid the automatic creation of the *accept* default action. This option can be used for example when you want to write a specific validation procedure, by using ACCEPT INPUT.

CANCEL option

The `CANCEL` attribute can be set to FALSE to avoid the automatic creation of the *cancel* default action. This is useful for example when you only need a validation action (accept), or when you want to write a specific cancellation procedure, by using EXIT INPUT.

Note that if the `CANCEL=FALSE` option is set, no *close* action will be created, and you must write an `ON ACTION close` control block to create an explicit action.

APPEND ROW option

The `APPEND ROW` attribute can be set to FALSE to avoid the automatic creation of the *append* default action, and deny the user to add rows at the end of the list. If `APPEND ROW = FALSE`, the user can still insert rows in the middle of the list. Use the `INSERT ROW` attribute to disallow the user from inserting rows.

INSERT ROW option

Using the `INSERT ROW` attribute, you can define with a Boolean value whether the user is allowed to insert new rows in the middle of the list. However, even if `INSERT ROW` is FALSE, the user can still append rows at the end of the list. Use the `APPEND ROW` attribute to disallow the user from appending rows.

DELETE ROW option

Using the `DELETE ROW` attribute, you can define with a Boolean value whether the user is allowed to delete rows (TRUE) or not allowed to delete rows (FALSE).

AUTO APPEND option

By default, an `INPUT ARRAY` controller creates a temporary row when needed (for example, when the user deletes the last row of the list, an new row will be automatically created). You can prevent this default behavior by setting the `AUTO APPEND` attribute to FALSE. If this attribute is set to TRUE, the only way to create a new temporary row is to execute the *append* action.

KEEP CURRENT ROW option

Depending on the list container used in the form, the current row may be highlighted during the execution of the dialog, and cleared when the instruction ends. You can change this default behavior by using the `KEEP CURRENT ROW` attribute, to force the runtime system to keep the current row highlighted.

Default Actions

When an `INPUT ARRAY` instruction executes, the runtime system creates a set of default actions. See the control block execution order to understand what control blocks are executed when a specific action is fired.

The following table lists the default actions created for this dialog:

Default action	Description
`accept`	Validates the `INPUT ARRAY` dialog (validates fields) *Creation can be avoided with* `ACCEPT` *attribute.*
`cancel`	Cancels the `INPUT ARRAY` dialog (no validation, INT_FLAG is set) *Creation can be avoided with* `CANCEL` *attribute.*
`close`	By default, cancels the `INPUT ARRAY` dialog (no validation,

	INT_FLAG is set) Default action view is hidden. See Windows closed by the user.
`insert`	Inserts a new row before current row. *Creation can be avoided with* `INSERT ROW = FALSE` *attribute.*
`append`	Appends a new row at the end of the list. *Creation can be avoided with* `APPEND ROW = FALSE` *attribute.*
`delete`	Deletes the current row. *Creation can be avoided with* `DELETE ROW = FALSE` *attribute.*
`help`	Shows the help topic defined by the `HELP` clause. *Only created when a* `HELP` *clause is defined.*
`firstrow`	Moves to the first row in the list.
`lastrow`	Moves to the last row in the list.
`nextrow`	Moves to the next row in the list.
`prevrow`	Moves to the previous row in the list.

The `insert`, `append`, `delete`, `accept` and `cancel` default actions can be avoided with dialog control attributes:

```
01  INPUT ARRAY arr TO sr.* ATTRIBUTES( INSERT ROW=FALSE, CANCEL=FALSE,
... )
02        ...
```

Control Blocks

- BEFORE INPUT block
- AFTER INPUT block
- BEFORE FIELD block
- AFTER FIELD block
- ON CHANGE block
- BEFORE ROW block
- ON ROW CHANGE block
- AFTER ROW block
- BEFORE INSERT block
- AFTER INSERT block
- BEFORE DELETE block
- AFTER DELETE block

BEFORE INPUT block

The `BEFORE INPUT` block is executed one time, before the runtime system gives control to the user. You can implement initialization in this block.

AFTER INPUT block

The `AFTER INPUT` block is executed one time, after the user has validated or canceled the dialog, and before the runtime system executes the instruction that appears just after

the INPUT ARRAY block. You typically implement dialog finalization in this block. The AFTER INPUT block is not executed if EXIT INPUT executes.

BEFORE ROW block

The BEFORE ROW block is executed each time the user moves to another row. This trigger can also be executed in other situations, such as when you delete a row, or when the user tries to insert a row but the maximum number of rows in the list is reached (see Control Blocks Execution Order for more details).

You typically do some dialog setup / message display in the BEFORE ROW block, because it indicates that the user selected a new row or entered in the list.

When the dialog starts, BEFORE ROW will be executed for the current row.

When called in this block, the ARR_CURR() function returns the index of the current row.

In the following example, the BEFORE ROW block gets the new row number and displays it in a message:

```
01 INPUT ARRAY ...
02    ...
03    BEFORE ROW
04      MESSAGE "We are on row # ", arr_curr()
05    ...
06 END INPUT
```

ON ROW CHANGE block

An ON ROW CHANGE block is executed when the user moves to another row after modifications have been done in the current row. It is triggered if the value of a field has changed since the row was entered and if the 'touched' flag of the corresponding field is set. The field might not be the current field, and several field values can be changed.

The 'touched' flag is set on user input or when doing a DISPLAY TO or a DISPLAY BY NAME. Note that the 'touched' flag is reset for all fields when entering another row, or when leaving the dialog instruction.

You can, for example, code database modifications (UPDATE) in this block. This block is executed before the AFTER ROW block if defined. When called in this block, the ARR_CURR() function returns the index of the current row where values have been changed.

```
01 INPUT ARRAY p_items FROM s_items.*
02    ON ROW CHANGE
03       UPDATE items SET
04              items.code        = p_items[arr_curr()].code,
05              items.description = p_items[arr_curr()].description,
06              items.price       = p_items[arr_curr()].price,
07              items.updatedate  = TODAY
```

```
08              WHERE items.num = p_items[arr_curr()].num
09 ...
```

AFTER ROW block

The AFTER ROW block is executed each time the user moves to another row, before the current row is left. This trigger can also be executed in other situations, such as when you delete a row, or when the user inserts a new row. see Control Blocks Execution Order for more details.

A NEXT FIELD executed in the AFTER ROW control block will keep the user entry in the current row. Thus you can use this to implement row input validation and prevent the user from leaving the list or moving to another row.

When called in this block, the ARR_CURR() function return the index of the row that you are leaving.

Warnings:

1. When leaving a temporary row that will be removed because user goes to a previous row in the list, AFTER ROW is executed for the temporary row, but ARR_CURR() will be one row greater as ARR_COUNT(). You should not access a dynamic array with a row index that is greater as the total number of rows, otherwise the runtime system will adapt the total number of rows to the actual number of rows in the program array.

In the following example, the AFTER ROW block checks a variable value and forces the user to stay in the current row if the value is wrong:

```
01 INPUT ARRAY p_items FROM s_items.*
02    AFTER ROW
03       IF arr_curr()>0 AND arr_curr() <= arr_count() THEN
04         IF NOT
item_is_valid_quantity(p_item[arr_curr()].item_quantity) THEN
05            ERROR "Item quentity is not valid"
06            NEXT FIELD item_quantity
07       END IF
08      END IF
09 ...
```

BEFORE FIELD block

A BEFORE FIELD block is executed each time the cursor enters into the specified field, when moving the focus from field to field in the same row, or when moving to another row.

The BEFORE FIELD block is also executed when using NEXT FIELD.

ON CHANGE block

The ON CHANGE block is executed when another field is selected, if the value of the specified field has changed since the field got the focus and if the 'touched' field flag is set. The 'touched' flag is set when a user modification is done or when doing a DISPLAY TO or a DISPLAY BY NAME. Once set, the 'touched' flag is not reset until the end of the dialog.

For fields defined as RadioGroup, ComboBox, SpinEdit, Slider, and CheckBox views, the ON CHANGE block is fired immediately when the user changes the value. For other type of fields (like Edits), the ON CHANGE block is fired when leaving the field. You leave the field when you validate the dialog, when you move to another field, or when you move to another row in an INPUT ARRAY. Note that the dialogtouched predefined action can also be used to detect field changes immediately, but with this action you can't get the data in the target variables (and it should only be used to detect that the user has started to modify data)

If both an ON CHANGE block and AFTER FIELD block are defined for a field, the ON CHANGE block is executed before the AFTER FIELD block.

When changing the value of the current field by program in an ON ACTION block, the ON CHANGE block will be executed when leaving the field if the value is different from the reference value and if the 'touched' flag is set (after previous user input or DISPLAY TO / DISPLAY BY NAME).

When using the NEXT FIELD instruction, the comparison value is re-assigned as if the user had left and re-entered the field. Therefore, when using NEXT FIELD in an ON CHANGE block or in an ON ACTION block, the ON CHANGE block will only be fired again if the value is different from the reference value. Therefore, it is recommended not to attempt field validation in ON CHANGE blocks: you would do better to perform validations in AFTER FIELD blocks and/or AFTER INPUT blocks.

AFTER FIELD block

An AFTER FIELD block is executed each time the cursor leaves the specified field, when moving the focus from field to field in the same row, or when moving to another row.

BEFORE INSERT block

The BEFORE INSERT block is executed each time the user inserts a new row, before the new row is created and made the current one.

You typically assign default values to the array variables of the newly created row, before the user gets control to enter more values and validates the row creation.

When called in this block, the ARR_CURR() function returns the index of the newly created row. The row in the program array can be referenced with this index, since the new element is already created in the array. Note that the BEFORE ROW block is also executed (just before BEFORE INSERT) when inserting a new row, but the current row index returned by ARR_CURR() is one higher than the actual number of rows in the list (arr_count()).

If needed, you can cancel the insert operation with the CANCEL INSERT instruction. Note that this control instruction can only be used in a BEFORE INSERT or AFTER INSERT block. When a CANCEL INSERT is performed in BEFORE INSERT, the dialog will execute some control blocks such as AFTER ROW / BEFORE ROW / BEFORE FIELD for the current row, even if no new row was inserted.

In the following example, the BEFORE INSERT block sets some default values and displays a message:

```
01 INPUT ARRAY p_items FROM s_items.*
02    BEFORE INSERT
03       LET r = DIALOG.getCurrentRow("s_items")
04       LET p_items[r].item_num = getNewSerial("items")
05       LET p_items[r].item_code = "C" || p_items[r].item_num
06       LET p_items[r].item_price = 100.00
07       MESSAGE "You are creating a new record..."
08 ...
```

AFTER INSERT block

The AFTER INSERT block is executed each time the user leaves a new created row. This block is typically used to implement SQL command that inserts a new row in the database. You can cancel the operation with the CANCEL INSERT instruction.

Warning: When the the user appends a new row at the end of the list, then moves UP to another row or validates the dialog, the AFTER INSERT block is only executed if at least one field was edited. If not data entry is detected, the dialog automatically removes the new appended row and thus does not trigger the AFTER INSERT block.

In the following example, the AFTER INSERT block inserts a new row in the database and cancels the operation if the SQL command fails:

```
01 INPUT ARRAY
02    ...
03    AFTER INSERT s_items
04       LET r = DIALOG.getCurrentRow("s_items")
05       INSERT INTO items VALUES ( p_items[r]. * )
06       IF SQLCA.SQLCODE < 0 THEN
07         ERROR "Could not insert row into database"
08         CANCEL INSERT
09       END IF
10    ...
11 END INPUT
```

BEFORE DELETE block

The BEFORE DELETE block is executed each time the user deletes a row, before the row is removed from the list.

You typically code the database table synchronization in the BEFORE DELETE block, by executing a DELETE SQL statement using the primary key of the current row. In the

BEFORE DELETE block, the row to be deleted still exists in the program array, so you can access its data to identify what record needs to be removed.

If needed, the deletion can be canceled with the CANCEL DELETE instruction.

When called in this block, the ARR_CURR() function returns the index of the row that will be deleted.

In the following example, the BEFORE DELETE block removes the row from the database table and cancels the deletion operation if an SQL error occurs:

```
01 INPUT ARRAY p_items FROM s_items.*
02    BEFORE DELETE
03       WHENEVER ERROR CONTINUE
04       DELETE FROM items WHERE item_num =
p_items[arr_curr()].item_num
05       WHENEVER ERROR STOP
06       IF SQLCA.SQLCODE<>0 VALUES
07          ERROR SQLERRMESSAGE
08          CANCEL DELETE
09       END IF
10 ...
```

AFTER DELETE block

The AFTER DELETE block is executed each time the user deletes a row, after the row has been deleted from the list.

When an AFTER DELETE block executes, the program array has already been modified; the deleted row no longer exists in the array. Note that the ARR_CURR() function returns the same index as in BEFORE ROW, but it is the index of the new current row. Note that the AFTER ROW block is also executed. Pay particular attention when deleting the last row in the list; in this case, the current row index returned by ARR_CURR() is one higher than the actual number of rows in the list (ARR_COUNT()).

Warnings:

1. When deleting the last row of the list, AFTER DELETE is executed for the delete row, and ARR_CURR() will be one row greater as ARR_COUNT(). You should not access a dynamic array with a row index that is greater as the total number of rows, otherwise the runtime system will adapt the total number of rows to the actual number of rows in the program array.

In the following example, the AFTER DELETE block is used to re-number the rows with a new item line number (note ARR_COUNT() may return zero):

```
01 INPUT ARRAY p_items FROM s_items.*
02    AFTER DELETE
03       LET r = arr_curr()
04       FOR i=r TO arr_count()
05           LET p_items[i].item_lineno = i
```

```
06       END FOR
07 ...
```

Control Block Execution Order

The following table shows the order in which the runtime system executes the control blocks in the `INPUT ARRAY` instruction, according to the user action:

Context / User action	Control Block execution order
Entering the dialog	1. `BEFORE INPUT` 2. `BEFORE ROW` 3. `BEFORE FIELD`
Moving to a different row from field A to field B	1. `ON CHANGE` (if value has changed for field A) 2. `AFTER FIELD` (for field A in the row you leave) 3. `AFTER INSERT` (if the row you leave was inserted or appended) or `ON ROW CHANGE` (if values have changed in the row you leave) 4. `AFTER ROW` (for the row you leave) 5. `BEFORE ROW` (the new current row) 6. `BEFORE FIELD` (for field B in the new current row)
Moving from field A to field B in the same row	1. `ON CHANGE` (if value has changed for field A) 2. `AFTER FIELD` (for field A) 3. `BEFORE FIELD` (for field B)
Deleting a row	1. `BEFORE DELETE` (for the row to be deleted) 2. `AFTER DELETE` (for the deleted row) 3. `AFTER ROW` (for the deleted row) 4. `BEFORE ROW` (for the new current row) 5. `BEFORE FIELD` (field in the new current row)
Inserting a new row between rows	1. `ON CHANGE` (if value has changed in the field you leave) 2. `AFTER FIELD` (for the row you leave) 3. `AFTER INSERT` (if the row you leave was inserted or appended) or `ON ROW CHANGE` (if values have changed in the row you leave) 4. `AFTER ROW` (for the row you leave) 5. `BEFORE INSERT` (for the new created row)

	6. `BEFORE FIELD` (for the new created row)
Appending a new row at the end	1. `ON CHANGE` (if value has changed in the current field) 2. `AFTER FIELD` (for the row you leave) 3. `AFTER INSERT` (if the row you leave was inserted or appended) or `ON ROW CHANGE` (if values have changed in the row you leave) 4. `AFTER ROW` (for the row you leave) 5. `BEFORE ROW` (for the new created row) 6. `BEFORE INSERT` (for the new created row) 7. `BEFORE FIELD` (for the new created row)
Validating the dialog	1. `ON CHANGE` 2. `AFTER FIELD` 3. `AFTER INSERT` (if the current row was inserted or appended) or `ON ROW CHANGE` (if values have changed in the current row) 4. `AFTER ROW` 5. `AFTER INPUT`
Canceling the dialog	1. `AFTER ROW` 2. `AFTER INPUT`

Interaction Blocks

- ON IDLE block
- ON ACTION block
- ON KEY block

ON IDLE block

The `ON IDLE` *`idle-seconds`* clause defines a set of instructions that must be executed after *idle-seconds* of inactivity. For example, this can be used to quit the dialog after the user has not interacted with the program for a specified period of time. The parameter *idle-seconds* must be an integer literal or variable. If it evaluates to zero, the timeout is disabled.

```
01 ...
02    ON IDLE 10
```

```
03       IF ask_question("Do you want to leave the dialog?") THEN
04          EXIT INPUT
05       END IF
06 ...
```

ON ACTION block

You can use `ON ACTION` blocks to execute a sequence of instructions when the user raises a specific action. This is the preferred solution compared to `ON KEY` blocks, because `ON ACTION` blocks use abstract names to control user interaction.

```
01 ...
02    ON ACTION zoom
03       CALL zoom_customers() RETURNING st, cust_id, cust_name
04       ...
```

ON KEY block

For backward compatibility, you can use `ON KEY` blocks to execute a sequence of instructions when the user presses a specific key. The following key names are accepted by the compiler:

Key Name	Description
`ACCEPT`	The validation key.
`INTERRUPT`	The interruption key.
`ESC` or `ESCAPE`	The ESC key (not recommended, use `ACCEPT` instead).
`TAB`	The TAB key (not recommended).
`Control-`*`char`*	A control key where *char* can be any character except A, D, H, I, J, K, L, M, R, or X.
`F1` through `F255`	A function key.
`DELETE`	The key used to delete a new row in an array.
`INSERT`	The key used to insert a new row in an array.
`HELP`	The help key.
`LEFT`	The left arrow key.
`RIGHT`	The right arrow key.
`DOWN`	The down arrow key.
`UP`	The up arrow key.
`PREVIOUS` or `PREVPAGE`	The previous page key.
`NEXT` or `NEXTPAGE`	The next page key.

Control Instructions

- CONTINUE INPUT instruction
- EXIT INPUT instruction

- ACCEPT INPUT instruction
- CANCEL INSERT instruction
- CANCEL DELETE instruction
- NEXT FIELD instruction
- CLEAR field-list instruction

Continuing the dialog: CONTINUE INPUT

`CONTINUE INPUT` skips all subsequent statements in the current control block and gives the control back to the dialog. This instruction is useful when program control is nested within multiple conditional statements, and you want to return the control to the dialog. Note that if this instruction is called in a control block that is not `AFTER INPUT`, further control blocks might be executed according to the context. Actually, `CONTINUE INPUT` just instructs the dialog to continue as if the code in the control block was terminated (i.e. it's a kind of `GOTO end_of_control_block`). However, when executed in `AFTER INPUT`, the focus returns to the current row and current field in the list, giving the user another chance to enter data in that field. In this case the `BEFORE ROW` and `BEFORE FIELD` triggers will be fired.

In the following example, an `ON ACTION` block gives control back to the dialog, skipping all instructions below line 04:

```
01    ON ACTION zoom
02       IF p_cust.cust_id IS NULL OR p_cust.cust_name IS NULL THEN
03         ERROR "Zoom window cannot be opened if there is no info to
identify the customer"
04         CONTINUE INPUT
05       END IF
06       IF p_cust.cust_address IS NULL THEN
07       ...
```

Note that you can also use the `NEXT FIELD` control instruction to give the focus to a specific field and force the dialog to continue. However, unlike `CONTINUE INPUT`, the `NEXT FIELD` instruction will also skip the further control blocks that are normally executed.

Leaving the dialog: EXIT INPUT

You can use the `EXIT INPUT` to terminate the `INPUT ARRAY` instruction and resume the program execution at the instruction following the `INPUT ARRAY` block.

```
01    ON ACTION quit
02       EXIT DIALOG
```

When leaving the `INPUT ARRAY` instruction, all form items used by the dialog will be disabled until another interactive statement takes control.

Validating the dialog: ACCEPT INPUT

The ACCEPT INPUT instruction validates the INPUT instruction and exits the INPUT ARRAY instruction if no error is raised. The AFTER FIELD, ON CHANGE, etc. control blocks will be executed. Statements after the ACCEPT INPUT will not be executed.

Input field validation is a process that does several successive validation tasks, as listed below:

1. The current field value is checked, according to the program variable data type (for example, the user must input a valid date in a DATE field).
2. NOT NULL field attributes are checked for all input fields. This attribute forces the field to have a value set by program or entered by the user. If the field contains no value, the constraint is not satisfied. Note that input values are right-trimmed, so if the user inputs only spaces, this corresponds to a NULL value which does not fulfill the NOT NULL constraint.
3. INCLUDE field attributes are checked for all input fields. This attribute forces the field to contain a value that is listed in the include list. If the field contains a value that is not in the list, the constraint is not satisfied.
4. REQUIRED field attributes are checked for all input fields. This attribute forces the field to have a default value, or to be 'touched' by the user or programmatically. If the field was not edited during the dialog, the constraint is not satisfied.

If a field does not satisfy one of the above constraints, dialog termination is canceled, an error message is displayed, and the focus goes to the first field causing a problem.

Canceling row insertion: CANCEL INSERT

Insertion can be canceled, by using the CANCEL INSERT instruction, in the BEFORE INSERT or AFTER INSERT blocks. Using this instruction in a different place will generate a compilation error.

The instructions that appear after CANCEL INSERT will be skipped.

A CANCEL INSERT executed inside a BEFORE INSERT block prevents the new row creation. The following tasks are performed:

1. No new row will be created (the new row is not yet shown to the user).
2. The BEFORE INSERT block is terminated (further instructions are skipped).
3. The BEFORE ROW and BEFORE FIELD triggers are executed.
4. Control goes back to the user.

For example, you can cancel a row insertion if the user is not allowed to create rows:

```
01    BEFORE INSERT
02       IF user_can_insert() == FALSE THEN
03          ERROR "You are not allowed to insert rows"
04          CANCEL INSERT
05       END IF
```

A CANCEL INSERT executed inside an AFTER INSERT block removes the newly created row. The following tasks are performed:

1. The newly created row is removed from the list (the row exists now and user has entered data).
2. The AFTER INSERT block is terminated (further instructions are skipped).
3. The BEFORE ROW and BEFORE FIELD triggers are executed.
4. The control goes back to the user.

For example, you can cancel a row insertion if a database error occurs when you try to insert the row into a database table:

```
01    AFTER INSERT
02       WHENEVER ERROR CONTINUE
03       INSERT INTO customer VALUES ( arr[arr_curr()].* )
04       WHENEVER ERROR STOP
05       IF SQLCA.SQLCODE<>0 THEN
06          ERROR SQLERRMESSAGE
07          CANCEL INSERT
08       END IF
```

If the CANCEL INSERT is done while on a new row that was appended to the end of the list, the new row will be removed and the previous row will get the focus. If there are no more existing rows, the list loses the focus because no row can be edited. The next time the user clicks in a cell, DIALOG will automatically create a new row.

You can also disable the *insert* and *append* actions to prevent the user from performing these actions with:

```
01    CALL DIALOG.setActionActive("insert", FALSE)
02    CALL DIALOG.setActionActive("append", FALSE)
```

Canceling row deletion: CANCEL DELETE

Deletion can be canceled, by using the CANCEL DELETE instruction in the BEFORE DELETE block. Using this instruction in a different place will generate a compilation error.

When the CANCEL DELETE instruction is executed, the current BEFORE DELETE block is terminated without any other trigger execution (no BEFORE ROW or BEFORE FIELD is executed), and the program execution continues in the user event loop.

For example, you can prevent row deletion based on some condition:

```
01    BEFORE DELETE
02       IF user_can_delete() == FALSE THEN
03          ERROR "You are not allowed to delete rows"
04          CANCEL DELETE
05       END IF
```

The instructions that appear after CANCEL DELETE will be skipped.

You can also disable the *delete* action to prevent the user from performing a delete row action with:

```
01    CALL DIALOG.setActionActive("delete", FALSE)
```

Moving to a field: NEXT FIELD

The `NEXT FIELD field-name` instruction gives the focus to the specified field. You typically use this instruction to control field input dynamically, in `BEFORE FIELD` or `AFTER FIELD` blocks, or to control row validation in `AFTER ROW`.

Abstract field identification is supported with the `CURRENT`, `NEXT` and `PREVIOUS` keywords. These keywords represent respectively the current, next and previous fields, corresponding to variables as defined in the input binding list (with the `FROM` or `BY NAME` clause).

Non-editable fields are fields defined with NOENTRY attribute or using a widget that does not allow input, such as a LABEL. If a `NEXT FIELD` instruction selects a non-editable field, the next editable field gets the focus (defined by the FIELD ORDER FORM mode used by the dialog). However, the `BEFORE FIELD` and `AFTER FIELD` blocks of non-editable fields are executed when a `NEXT FIELD` instruction selects such a field.

When using `NEXT FIELD` in `AFTER ROW` or in `ON ROW CHANGE`, the dialog will stay in the current row and give the control back to the user. This behavior allows to implement data input rules:

```
01 ...
02    AFTER ROW
03       IF NOT int_flag AND arr_curr() <= arr_count() THEN
04         IF arr[arr_curr()].it_count * arr[arr_curr()].it_value >
maxval THEN
05            ERROR "Amount of line exceeds max value."
06            NEXT FIELD item_count
07       END IF
09 ...
```

For compatibility reasons, `NEXT FIELD` in `AFTER INSERT` will not stay in the current row.

Clearing the form fields: CLEAR field-list

For backward compatibility, the `CLEAR field-list` instruction is provided to clear a specific field or all fields in a line of the screen array. You can specify the screen array as described in the following table:

CLEAR instruction	Result
`CLEAR field-name`	Clears the specified field in the current line of the screen array.
`CLEAR screen-array.*`	Clears all fields in the current line of the screen array.

`CLEAR screen-array[n].*`	Clears all fields in line *n* of the screen array.
`CLEAR screen-array[n].field-name`	Clears the specified field in line *n* of the screen array.

Warnings:

1. When using the UNBUFFERED attribute, it is recommended that you do NOT use the CLEAR instruction; always use program variables to set field values to NULL.

Control Class

Inside the dialog instruction, the predefined keyword `DIALOG` represents the current dialog object. It can be used to execute methods provided in the dialog built-in class.

For example, you can enable or disable an action with the ui.Dialog.setActionActive() dialog method, and you can hide and show the default action view with ui.Dialog.setActionHidden():

```
01 ...
02    BEFORE INPUT
03       CALL DIALOG.setActionActive("zoom",FALSE)
04    AFTER FIELD field1
05       CALL DIALOG.setActionHidden("zoom",1)
06 ...
```

The ui.Dialog.setFieldActive() method can be used to enable or disable a field during the dialog. This instruction takes an integer expression as argument.

```
01 ...
02    ON CHANGE custname
03       CALL DIALOG.setFieldActive( "custaddr",
(cust_arr[arr_curr()].custname IS NOT NULL) )
04 ...
```

Control Functions

The language provides several built-in functions and operators to use in an `INPUT ARRAY` statement. You can use the following built-in functions to keep track of the relative states of the current row, the program array, and the screen array, or to access the field buffers and keystroke buffers:

- ARR_CURR()
- ARR_COUNT()
- FGL_SET_ARR_CURR()
- SET_COUNT()

- FIELD_TOUCHED()
- GET_FLDBUF()
- INFIELD()
- FGL_DIALOG_GETFIELDNAME()
- FGL_DIALOG_GETBUFFER()

Examples

Example 1: Basic INPUT ARRAY

Form definition file (custlist.per):

```
01 DATABASE stores
02 LAYOUT
03 TABLE
04 {
05  Id       First name   Last name
06 [f001    |f002        |f003        ]
07 [f001    |f002        |f003        ]
08 [f001    |f002        |f003        ]
09 [f001    |f002        |f003        ]
10 [f001    |f002        |f003        ]
11 [f001    |f002        |f003        ]
12 }
13 END
14 END
15 TABLES
16   customer
17 END
18 ATTRIBUTES
19   f001 = customer.customer_num ;
20   f002 = customer.fname ;
21   f003 = customer.lname ;
22 END
23 INSTRUCTIONS
24   SCREEN RECORD sr_cust[6]( customer.* );
25 END
```

Program source code:

```
01 MAIN
02     DEFINE custarr ARRAY[500] OF RECORD
03            id INTEGER,
04            fname CHAR(30),
05            lname CHAR(30)
06     END RECORD
07
08     OPEN FORM f FROM "custlist"
09     DISPLAY FORM f
10
11     INPUT ARRAY custarr WITHOUT DEFAULTS FROM sr_cust.*
12 END MAIN
```

Example 2: INPUT ARRAY using default values

The form definition file is the same as in example 1.

```
01 MAIN
02     DEFINE allow_insert INTEGER
03     DEFINE size INTEGER
04     DEFINE custarr ARRAY[500] OF RECORD
05             id INTEGER,
06             fname CHAR(30),
07             lname CHAR(30)
08     END RECORD
09     LET custarr[1].id = 1
10     LET custarr[1].fname = "John"
11     LET custarr[1].lname = "SMITH"
12     LET custarr[2].id = 2
13     LET custarr[2].fname = "Mike"
14     LET custarr[2].lname = "STONE"
15     LET size = 2
16     LET allow_insert = TRUE
17
18     OPEN FORM f1 FROM "custlist"
19     DISPLAY FORM f1
20
21     INPUT ARRAY custarr WITHOUT DEFAULTS FROM sr_cust.*
22         ATTRIBUTES (COUNT=size, MAXCOUNT=50, UNBUFFERED, INSERT
ROW=allow_insert)
23         BEFORE INPUT
24           MESSAGE "Editing the customer table"
25         BEFORE INSERT
26           IF arr_curr()=4 THEN
27              CANCEL INSERT
28           END IF
29         BEFORE FIELD fname
30           MESSAGE "Enter First Name"
31         BEFORE FIELD lname
32           MESSAGE "Enter Last Name"
33         AFTER FIELD lname
34           IF custarr[arr_curr()].lname IS NULL THEN
35              LET custarr[arr_curr()].fname = NULL
36           END IF
37     END INPUT
38 END MAIN
```

Example 3: INPUT ARRAY using a dynamic array

The form definition file is the same as in example 1.

```
01 MAIN
02     DEFINE counter INTEGER
03     DEFINE custarr DYNAMIC ARRAY OF RECORD
04             id INTEGER,
05             fname CHAR(30),
06             lname CHAR(30)
07     END RECORD
```

```
10      FOR counter = 1 TO 500
11        LET custarr[1].id = counter
12        LET custarr[1].fname = "ff"||counter
13        LET custarr[1].lname = "NNN"||counter
14      END FOR
15
16      OPEN FORM f FROM "custlist"
17      DISPLAY FORM f
18
19      INPUT ARRAY custarr WITHOUT DEFAULTS FROM sr_cust.*
20          ATTRIBUTES ( UNBUFFERED )
21          ON ROW CHANGE
22            MESSAGE "Row #"||arr_curr()||" has been updated."
23      END INPUT
24 END MAIN
```

Example 4: INPUT ARRAY updating the database table

The form definition file is the same as in example 1.

```
01 MAIN
02
03    DEFINE custarr DYNAMIC ARRAY OF RECORD
04          id INTEGER,
05          fname CHAR(30),
06          lname CHAR(30)
07    END RECORD
08
09    DEFINE op CHAR(1)
10
11    OPEN FORM f1 FROM "custlist"
12    DISPLAY FORM f1
13
14    INPUT ARRAY custarr FROM sr1.* ATTRIBUTES(WITHOUT DEFAULTS,
UNBUFFERED)
15
16          BEFORE DELETE
17            IF op == "N" THEN -- No real SQL delete for new inserted
rows
18               IF NOT mbox_yn("List", "Are you sure you want to delete
this record?", "question") THEN
19                 CANCEL DELETE -- Keeps row in list
20               END IF
21               DELETE FROM customer WHERE customer_num =
custarr[arr_curr()].id
22               IF SQLCA.SQLCODE<0 THEN
23                 ERROR "Could not delete the record from database!"
24                 CANCEL DELETE -- Keeps row in list
25               END IF
26            END IF
27
28          AFTER DELETE
29            MESSAGE "Record has been deleted successfully"
30
31          AFTER FIELD fname
32            IF custarr[arr_curr()].fname MATCHES "*@#$%^&()*" THEN
```

```
33              ERROR "This field contains invalid characters"
34              NEXT FIELD CURRENT
35            END IF
36
37          ON ROW CHANGE
38            -- Warning: ON ROW CHANGE can occur if the SQL INSERT
failed...
39            IF op != "I" THEN LET op = "M" END IF
40
41          BEFORE INSERT
42            LET op = "T"
43            LET custarr[arr_curr()].id = arr_count()
44
45          AFTER INSERT
46            LET op = "I"
47
48          BEFORE ROW
49            LET op = "N"
50
51          AFTER ROW
52            IF int_flag THEN EXIT INPUT END IF
53            IF op == "M" OR op == "I" THEN
54               IF custarr[arr_curr()].fname IS NULL OR
custarr[arr_curr()].fname IS NULL
55                  OR custarr[arr_curr()].fname ==
custarr[arr_curr()].lname THEN
56                  ERROR "First name and last name are equal..."
57                  NEXT FIELD fname
58               END IF
59            END IF
60            IF op == "I" THEN
61               INSERT INTO customer VALUES ( custarr[arr_curr()].* )
62               IF SQLCA.SQLCODE<0 THEN
63                   ERROR "Could not insert the record into database!"
64                   NEXT FIELD CURRENT
65               END IF
66            END IF
67            IF op == "M" THEN
68               UPDATE customer SET fname = custarr[arr_curr()].fname,
lname = custarr[arr_curr()].lname
69                    WHERE customer_num = custarr[arr_curr()].id
70               IF SQLCA.SQLCODE<0 THEN
71                  ERROR "Could not update the record in database!"
72                  NEXT FIELD CURRENT
73               END IF
74            END IF
75
76    END INPUT
77
78 END MAIN
```

Query By Example

Summary:

- Basics
  - Query Operators
- Syntax
- Usage
  - Programming Steps
  - Instruction Configuration
  - Default Actions
  - Control Blocks
  - Control Blocks Execution Order
  - Interaction Blocks
  - Control Instructions
  - Control Class
  - Control Functions
- Examples

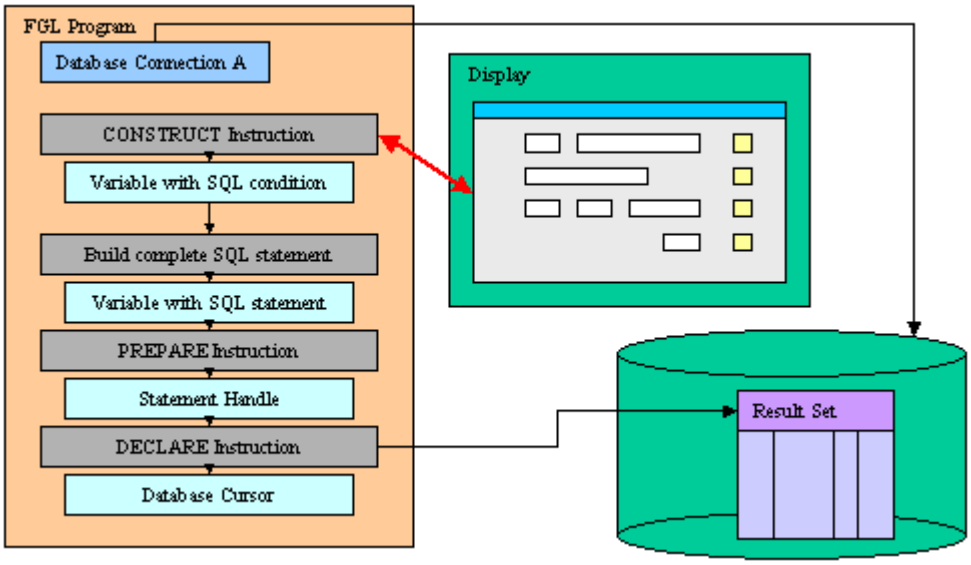
See also: Flow Control, Dynamic SQL, Result Sets

Basics

What is Query By Example (QBE)?

Query By Example enables a user to query a database by specifying values (or ranges of values) for screen fields that correspond to the database. The runtime system converts the search filters entered by the user into a Boolean SQL condition that can be used in the `WHERE` clause of a prepared `SELECT` statement.

The `CONSTRUCT` instruction handles Query By Example input in the current open form and generates the SQL condition in a string variable. You can then use Dynamic SQL instructions to execute the SQL statement to produce a result set:



Query Operators

The following table lists all relational operators that can be used during a Query By Example input:

Symbol	Meaning	Pattern
Any simple data type		
`=`	Is Null	=
`==`	Equal to	== *value*
`>`	Greater than	> *value*
`>=`	Greater than or equal to	>= *value*
`<`	Less than	< *value*
`<=`	Less than or equal to	<= *value*
`<> or !=`	Not equal to	!= *value*, <> *value*
`: or ..`	Range	*value1*:*value2*, *value1*..*value2*
`|`	List of values	*value1* | *value2*
Character data types only		
`*`	Wildcard for any string	*x, x*, *x*
`?`	Single-character wildcard	?x, x?, ?x?, x??
`[c]`	A set of characters	[a-z]*, [xy]?

Syntax

Purpose:

The `CONSTRUCT` instruction handles Query By Example input.

Syntax 1: Implicit field-to-column mapping

```
CONSTRUCT BY NAME variable ON column-list
  [ ATTRIBUTES ( { display-attribute | control-attribute } [,...] ) ]
  [ HELP help-number ]
[ dialog-control-block
  [...]
END CONSTRUCT ]
```

Syntax 2: Explicit field-to-column mapping

```
CONSTRUCT variable ON column-list FROM field-list
  [ ATTRIBUTES ( { display-attribute | control-attribute } [,...] ) ]
  [ HELP help-number ]
[ dialog-control-block
  [...]
END CONSTRUCT ]
```

where *column-list* is :

```
{ column-name
| table-name.*
| table-name.column-name
} [,...]
```

where *field-list* is :

```
{ field-name
| table-name.*
| table-name.field-name
| screen-array[line].*
| screen-array[line].field-name
| screen-record.*
| screen-record.field-name
} [,...]
```

where *dialog-control-block* is one of:

```
{ BEFORE CONSTRUCT
| AFTER CONSTRUCT
| BEFORE FIELD field-name [,...]
| AFTER FIELD field-name [,...]
| ON IDLE idle-seconds
| ON ACTION action-name
| ON KEY ( key-name [,...] )
}
    dialog-statement
    [...]
```

where *dialog-statement* is one of :

```
{ statement
| NEXT FIELD { NEXT | PREVIOUS | field-name }
```

```
| CONTINUE CONSTRUCT
| EXIT CONSTRUCT
}
```

Notes:

1. *variable* is the variable that will contain the SQL condition built by the `CONSTRUCT` instruction.
2. The `ON` clause defines the list of form fields in which the user can enter search criteria.
3. *column-name* is the identifier of a database column of the current form.
4. *table-name* is the identifier of a database table of the current form.
5. *field-name* is the identifier of a field of the current form.
6. The `BY NAME` keyword implicitly maps form fields to the database columns listed in the `ON` clause.
7. Use the `FROM field-list` clause if you need to map form fields to database columns explicitly.
8. *screen-array* is the screen array that will be used in the current form.
9. *line* is a screen array line in the form.
10. *screen-record* is the identifier of a screen record of the current form.
11. *help-number* is an integer that allows you to associate a help message number with the instruction.
12. *key-name* is a hot-key identifier (like `F11` or `Control-z`).
13. *action-name* identifies an action that can be executed by the user.
14. *idle-seconds* is an integer literal or variable that defines a number of seconds.
15. *statement* is any instruction supported by the language.

The following table shows the *options* supported by the `CONSTRUCT` statement:

Attribute	Description
`HELP` *`help-number`*	Defines the help number when help is invoked by the user, where *help-number* is an integer literal or a program variable. **Warning:** The HELP *option* overrides the HELP *attribute*!

The following table shows the *display-attributes* supported by the `CONSTRUCT` statement. The *display-attributes* affect console-based applications only, they do not affect GUI-based applications.

Attribute	Description
`BLACK, BLUE, CYAN, GREEN, MAGENTA, RED, WHITE, YELLOW`	The color of the entered text.. **TUI Only!**
`BOLD, DIM, INVISIBLE, NORMAL`	The font attribute of the entered text.. **TUI Only!**
`REVERSE, BLINK, UNDERLINE`	The video attribute of the entered text.. **TUI**

Only!

The following table shows the *control-attributes* supported by the CONSTRUCT statement:

Attribute	Description
NAME = *string*	Identifies the dialog statement with a clear name.
HELP = *help-number*	Defines the help number when help is invoked by the user, where *help-number* is an integer literal or a program variable. **Warning:** The HELP *option* overrides the HELP *attribute*!
FIELD ORDER FORM	Indicates that the tabbing order of fields is defined by the TABINDEX attribute of form fields.
ACCEPT = *bool*	Indicates if the default *accept* action should be added to the dialog. If not specified, the action is registered.
CANCEL = *bool*	Indicates if the default *cancel* action should be added to the dialog. If not specified, the action is registered.

Usage

The CONSTRUCT statement produces an SQL condition corresponding to all search criterion that a user specifies in the fields. The instruction fills a character variable with that SQL condition, and you can use the content of this variable to create the WHERE clause of a SELECT statement.

Warnings:

1. The SQL condition is generated according to the current database session, which defines the type of the database server. Therefore, the program must be connected to a database server before entering the CONSTRUCT block. The generated SQL condition is specific to the database server and may not be used with other types of servers.
2. There can be only one BEFORE CONSTRUCT and one AFTER CONSTRUCT in a CONSTRUCT block.
3. AFTER CONSTRUCT is not performed if an EXIT CONSTRUCT is performed, or if the Interrupt or Quit key is pressed and a DEFER INTERRUPT or DEFER QUIT statement is not in effect..
4. The WORDWRAP field attribute is not used by the CONSTRUCT instruction.
5. Make sure that the INT_FLAG variable is set to FALSE before entering the CONSTRUCT block.

6. The `ON KEY` blocks are supported for backward compatibility; use `ON ACTION` instead.

Programming Steps

To use the `CONSTRUCT` statement, you must do the following:

1. Declare a character variable with the DEFINE statement.
2. Open and display the form, using an OPEN WINDOW with the `WITH FORM` clause or the OPEN FORM / DISPLAY FORM instructions.
3. Describe the `CONSTRUCT` statement if needed, with *dialog-control-blocks* to control the environment in which the user enters search criteria.
4. After executing the `CONSTRUCT`, check the INT_FLAG variable to verify if the input was validated or canceled by the user.
5. Execute the query in the database (see below).

The `CONSTRUCT` statement activates the current form. This is the form most recently displayed or, if you are using more than one window, the form currently displayed in the current window. You can specify the current window by using the CURRENT WINDOW statement. When the `CONSTRUCT` statement completes execution, the form is cleared and deactivated.

Screen field tabbing order is defined by the order of the field names in the `FROM` clause; by default this is the list of column names in the `ON` clause when no `FROM` clause is specified.

To complete the search functionality of your program, you must implement the following steps after the `CONSTRUCT` instruction:

1. Concatenate the variable that contains the Boolean SQL expression with other strings, to create a string representation of an SQL statement to be executed. The Boolean SQL expression generated by the `CONSTRUCT` statement is typically used to create `SELECT` statements, but it can be used in `DELETE` and `UPDATE` statements.
2. Use the PREPARE statement to create an executable SQL statement from the character string that was generated in the previous step.
3. Execute the prepared statement in one of the following ways:
   - If the SQL statement produces a result set (like `SELECT`), use a database cursor with DECLARE and FOREACH instructions (or else OPEN and FETCH) to execute the prepared SQL statement.
   - If the SQL statement does not produce a result set (like `DELETE`), use the EXECUTE statement to execute an SQL statement.

If no criteria were entered, the string `' 1=1'` is assigned to *variable*. This is a Boolean SQL expression that always evaluates to TRUE so that all rows are returned.

After executing the `CONSTRUCT` instruction, the runtime system sets the INT_FLAG variable to TRUE if the input was canceled by the user.

When the CONSTRUCT statement completes execution, the form is cleared.

Instruction Configuration

The ATTRIBUTES clause specifications override all default attributes and temporarily override any display attributes that the OPTIONS or the OPEN WINDOW statement specified for these fields. While the CONSTRUCT statement is executing, the runtime system ignores the INVISIBLE attribute.

- HELP option
- FIELD ORDER FORM option
- ACCEPT option
- CANCEL option

HELP option

The HELP clause specifies the number of a help message to display if the user invokes the help while the focus is in any field used by the instruction. The predefined *help* action is automatically created by the runtime system. You can bind action views to the *help* action.

Warnings:

1. The HELP *option* overrides the HELP *attribute*!

FIELD ORDER FORM option

By default, the tabbing order is defined by the column list in the instruction description. You can control the tabbing order by using the FIELD ORDER FORM attribute: When this attribute is used, the tabbing order is defined by the TABINDEX attribute of the form fields.

ACCEPT option

The ACCEPT attribute can be set to FALSE to avoid the automatic creation of the *accept* default action. This option can be used for example when you want to write a specific validation procedure, by using ACCEPT CONSTRUCT.

CANCEL option

The CANCEL attribute can be set to FALSE to avoid the automatic creation of the *cancel* default action. This is useful for example when you only need a validation action (*accept*), or when you want to write a specific cancellation procedure, by using EXIT CONSTRUCT.

Note that if the `CANCEL=FALSE` option is set, no *close* action will be created, and you must write an `ON ACTION close` control block to create an explicit action.

Default Actions

When an `CONSTRUCT` instruction executes, the runtime system creates a set of default actions. See the control block execution order to understand what control blocks are executed when a specific action is fired.

The following table lists the default actions created for this dialog:

Default action	Description
`accept`	Validates the `CONSTRUCT` dialog (validates field criteria) *Creation can be avoided with `ACCEPT` attribute.*
`cancel`	Cancels the `CONSTRUCT` dialog (no validation, INT_FLAG is set) *Creation can be avoided with `CANCEL` attribute.*
`close`	By default, cancels the `CONSTRUCT` dialog (no validation, INT_FLAG is set) Default action view is hidden. See Windows closed by the user.
`help`	Shows the help topic defined by the `HELP` clause. *Only created when a `HELP` clause is defined.*

The `accept` and `cancel` default actions can be avoided with the `ACCEPT` and `CANCEL` dialog control attributes:

```
01  CONSTRUCT BY NAME cond ON field1 ATTRIBUTES ( CANCEL=FALSE )
02        ...
```

Control Blocks

- BEFORE CONSTRUCT block
- AFTER CONSTRUCT block
- BEFORE FIELD block
- AFTER FIELD block

BEFORE CONSTRUCT block

Use a `BEFORE CONSTRUCT` block to execute instructions before the runtime system gives control to the user for search criteria input.

AFTER CONSTRUCT block

Use an `AFTER CONSTRUCT` block to execute instructions after the user has finished search criteria input.

BEFORE FIELD block

Use a `BEFORE FIELD` *`field-name`* block to execute instructions before the *field-name* field is made current.

The `BEFORE FIELD` block is also executed when using NEXT FIELD.

AFTER FIELD block

Use an `AFTER FIELD` *`field-name`* block to execute instructions when the user moves to another field.

Interaction Blocks

- ON IDLE block
- ON ACTION block
- ON KEY block

ON IDLE block

The `ON IDLE` *`idle-seconds`* clause defines a set of instructions that must be executed after *idle-seconds* of inactivity. This can be used, for example, to quit the dialog after the user has not interacted with the program for a specified period of time. The parameter *idle-seconds* must be an integer literal or variable. If it evaluates to zero, the timeout is disabled.

```
01 ...
02    ON IDLE 10
03       IF ask_question("Do you want to leave the dialog?") THEN
04          EXIT CONSTRUCT
05       END IF
06 ...
```

ON ACTION block

You can use `ON ACTION` blocks to execute a sequence of instructions when the user raises a specific action. This is the preferred solution compared to `ON KEY` blocks, because `ON ACTION` blocks use abstract names to control user interaction.

```
01 ...
02    ON ACTION zoom
03       CALL zoom_customers() RETURNING st, cust_id, cust_name
```

```
04      ...
```

ON KEY block

For backward compatibility, you can use `ON KEY` blocks to execute a sequence of instructions when the user presses a specific key. The following key names are accepted by the compiler:

Key Name	Description
`ACCEPT`	The validation key.
`INTERRUPT`	The interruption key.
`ESC` or `ESCAPE`	The ESC key (not recommended, use `ACCEPT` instead).
`TAB`	The TAB key (not recommended).
`Control-`*`char`*	A control key where *char* can be any character except A, D, H, I, J, K, L, M, R, or X.
`F1` through `F255`	A function key.
`DELETE`	The key used to delete a new row in an array.
`INSERT`	The key used to insert a new row in an array.
`HELP`	The help key.
`LEFT`	The left arrow key.
`RIGHT`	The right arrow key.
`DOWN`	The down arrow key.
`UP`	The up arrow key.
`PREVIOUS` or `PREVPAGE`	The previous page key.
`NEXT` or `NEXTPAGE`	The next page key.

Control Block Execution Order

The following table shows the order in which the runtime system executes the control blocks in the `CONSTRUCT` instruction, according to the user action:

Context / User action	Control Block execution order
Entering the dialog	1. `BEFORE CONSTRUCT` 2. `BEFORE FIELD` (first field)
Moving from field A to field B	1. `AFTER FIELD` (for field A) 2. `BEFORE FIELD` (for field B)
Validating the dialog	1. `AFTER FIELD` 2. `AFTER CONSTRUCT`

Canceling the dialog	1. `AFTER CONSTRUCT`

Control Instructions

- CONTINUE CONSTRUCT
- EXIT CONSTRUCT
- ACCEPT CONSTRUCT
- NEXT FIELD

Continuing the dialog: CONTINUE CONSTRUCT

`CONTINUE CONTRUCT` skips all subsequent statements in the current control block and gives the control back to the dialog. This instruction is useful when program control is nested within multiple conditional statements, and you want to return the control to the dialog. Note that if this instruction is called in a control block that is not `AFTER CONSTRUCT`, further control blocks might be executed according to the context. Actually, `CONTINUE CONSTRUCT` just instructs the dialog to continue as if the code in the control block was terminated (i.e. it's a kind of `GOTO end_of_control_block`). However, when executed in `AFTER CONSTRUCT`, the focus returns to the most recently occupied field in the current form, giving the user another chance to enter data in that field. In this case the `BEFORE FIELD` of the current field will be fired.

Note that you can also use the `NEXT FIELD` control instruction to give the focus to a specific field and force the dialog to continue. However, unlike `CONTINUE CONSTRUCT`, the `NEXT FIELD` instruction will also skip the further control blocks that are normally executed.

Leaving the dialog: EXIT CONSTRUCT

`EXIT CONSTRUCT` terminates the `CONSTRUCT` instruction without executing any other statement.

Validating the dialog: ACCEPT CONSTRUCT

The `ACCEPT CONSTRUCT` instruction validates the `CONSTRUCT` instruction and exits the `CONSTRUCT` instruction if no error is raised. The `AFTER FIELD` and `AFTER CONSTRUCT` control blocks will be executed. Statements after the `ACCEPT CONSTRUCT` will not be executed.

Moving to a field: NEXT FIELD

The `NEXT FIELD` *`field-name`* instruction gives the focus to the specified field. You typically use this instruction to control field input dynamically, in `BEFORE FIELD` or `AFTER FIELD` blocks.

Abstract field identification is supported with the CURRENT, NEXT and PREVIOUS keywords. These keywords represent respectively the current, next and previous fields, corresponding to variables as defined in the input binding list (with the FROM or BY NAME clause).

Non-editable fields are fields defined with NOENTRY attribute or using a widget that does not allow input, such as a LABEL. If a NEXT FIELD instruction selects a non-editable field, the next editable field gets the focus (defined by the FIELD ORDER mode used by the dialog). However, the BEFORE FIELD and AFTER FIELD blocks of non-editable fields are executed when a NEXT FIELD instruction selects such a field.

Control Class

Inside the dialog instruction, the predefined keyword DIALOG represents the current dialog object. It can be used to execute methods provided in the dialog built-in class.

For example, you can enable or disable an action with the ui.Dialog.setActionActive() dialog method, or you can hide and show the default action view with ui.Dialog.setActionHidden():

```
01 ...
02    BEFORE CONSTRUCT
03       CALL DIALOG.setActionActive("refresh",FALSE)
04    AFTER FIELD field1
05       CALL DIALOG.setActionHidden("refresh",1)
```

The ui.Dialog.setFieldActive() method can be used to enable or disable a field during the dialog. This instruction takes an integer expression as argument.

```
01 ...
02    ON CHANGE custid
03       CALL DIALOG.setFieldActive( "custaddr", FALSE )
04 ...
```

Control Functions

The language provides several built-in functions and operators to use in an CONSTRUCT statement. You can access the field buffers and keystroke buffers with:

- FIELD_TOUCHED()
- GET_FLDBUF()
- INFIELD()
- FGL_DIALOG_GETFIELDNAME()
- FGL_DIALOG_GETBUFFER()
- FGL_DIALOG_SETBUFFER()

Examples

Example 1: Simple CONSTRUCT

Form definition in the *const.per* file:

```
01 DATABASE formonly
02
03 LAYOUT
04 GRID
05 {
06     FirstName  [f001     ]
07     LastName   [f002     ]
08     e-Mail     [f003     ]
09 }
10 END
11 END
12
13 ATTRIBUTES
14 f001 = formonly.field1 TYPE CHAR;
15 f002 = formonly.field2 TYPE CHAR;
16 f003 = formonly.field3 TYPE CHAR;
17 END
```

Program:

```
01 MAIN
02
03     DEFINE condition CHAR(100)
04
05     DATABASE stores
06
07     OPEN FORM f1 FROM "const"
08     DISPLAY FORM f1
09
10     CONSTRUCT condition
11         ON first_name, last_name, mail
12         FROM field1, field2, field3
13
14     DISPLAY condition
15
16 END MAIN
```

Example 2: CONSTRUCT followed by SQL Query

Form definition in the *const.per* file:

```
01 DATABASE stores
02
03 LAYOUT
04 GRID
```

```
{
    FirstName  [f001     ]
    LastName   [f002     ]
}
END
END

TABLES
customer
END

ATTRIBUTES
f001 = customer.first_name;
f002 = customer.last_name;
END
```

Program :

```
MAIN

    DEFINE condition VARCHAR(100)
    DEFINE statement VARCHAR(200)
    DEFINE cust RECORD
                first_name CHAR(30),
                last_name CHAR(30)
          END RECORD

    DATABASE stores

    OPEN FORM f1 FROM "const"
    DISPLAY FORM f1

    CONSTRUCT BY NAME condition ON first_name, last_name
       BEFORE CONSTRUCT
          DISPLAY "A*" TO first_name
          DISPLAY "B*" TO last_name
    END CONSTRUCT

    LET statement =
    "SELECT first_name, last_name FROM customer WHERE " || condition
    DISPLAY "SQL : " || statement

    PREPARE s1 FROM statement
    DECLARE c1 CURSOR FOR s1
    FOREACH c1 INTO cust.*
        DISPLAY cust.*
    END FOREACH

END MAIN
```

Multiple Dialogs

Summary:

- Basics
- Syntax
- Usage
  - Programming Steps
  - Sub-dialogs
  - Programming with DIALOG
  - DIALOG and sub-dialogs configuration clauses
  - Default Actions
  - Control Blocks
  - Control Blocks Execution Order
  - Interaction Blocks
  - Control Instructions
  - Control Class
  - Control Functions
- Examples
  - Example 1: Two Lists
  - Example 2: Query and List

See also: Arrays, Records, Result Sets, Programs, Windows, Forms, Display Array

---

Basics

The `DIALOG` instruction handles different parts of a form simultaneously, including simple display fields, simple input fields, read-only list of records, editable list of records, query by example fields, and action views. The `DIALOG` instruction acts as a collection of singular dialogs which work in parallel.

While the `DIALOG` instruction re-uses some of the semantics and behaviors of singular interactive instructions, there are some differences. By "singular interactive instructions", we mean the INPUT, CONSTRUCT, DISPLAY ARRAY and INPUT ARRAY instructions. The differences are discussed further within this topic.

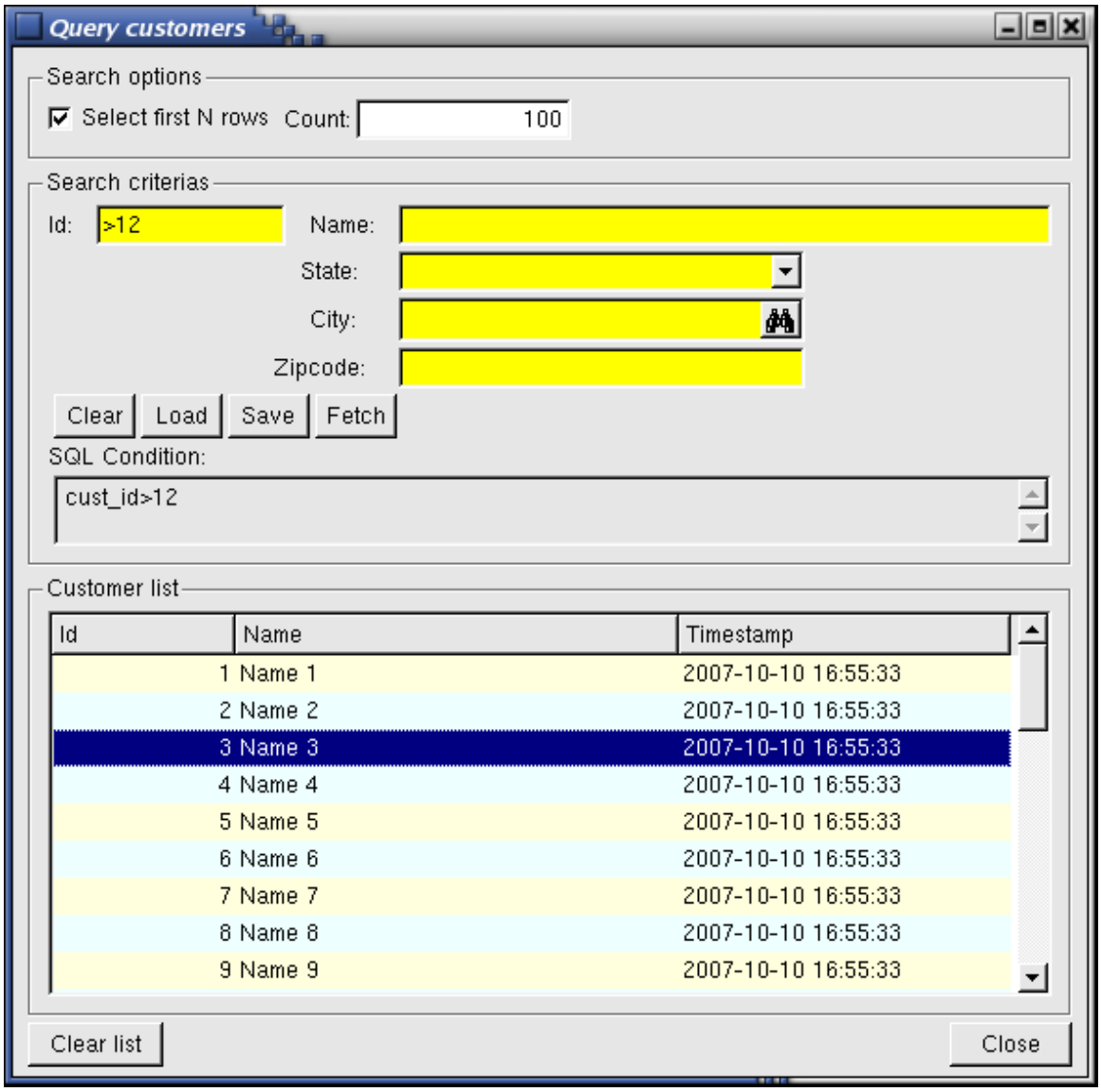
Like the singular interactive instructions, `DIALOG` is an interactive instruction. You can execute a `DIALOG` instruction from one of the singular dialogs, or execute a singular dialog from a `DIALOG` block. The parent dialog will be disabled until the child dialog returns.

The `DIALOG` instruction binds program variables such as simple records or arrays of records with a screen-record or screen-array defined in a form, allowing the user to view and update application data.

When a DIALOG block executes, it activates the current form (the form most recently displayed or the form in the current window). When the statement completes execution, the form is de-activated.

The DIALOG instruction was introduced in Genero BDL version **2.10**.

The following screen shot is from a demo program called **QueryCustomers** that you can find in FGLDIR/demo/MultipleDialogs. This demo involves a DIALOG block that contains a simple INPUT block, a CONSTRUCT block, and a DISPLAY ARRAY block:



DIALOG

Purpose:

The DIALOG block is an interactive instruction that executes multiple kinds of sub-controllers simultaneously to drive different parts of a form.

Syntax:

```
DIALOG
  [ ATTRIBUTE ( { dialog-control-attribute } [,...] ) ]

  { record-input-block
  | construct-block
  | display-array-block
  | input-array-block
  }
  [...]

[
  { dialog-control-block  }
  [...]
]

END DIALOG
```

where *record-input-block* is:

```
{
INPUT BY NAME { variable | record.* } [,...]
|
INPUT { variable | record.* } [,...]  FROM field-list
}
  [ ATTRIBUTE ( { input-control-attribute } [,...] ) ]
  { input-control-block  }
  [...]
END INPUT
```

where *input-control-block* is one of:

```
{ BEFORE INPUT
| BEFORE FIELD field-spec
| ON CHANGE field-spec
| AFTER FIELD field-spec
| AFTER INPUT
| ON ACTION action-name
| ON KEY ( key-name [,...] )
}
     dialog-statement
     [...]
```

where *construct-block* is:

```
{
CONSTRUCT BY NAME variable ON column-list
|
CONSTRUCT variable ON column-list FROM field-list
}
  [ ATTRIBUTE ( { construct-control-attribute } [,...] ) ]
  { construct-control-block  }
  [...]
END CONSTRUCT
```

where *construct-control-block* is one of:

```
{ BEFORE CONSTRUCT
| BEFORE FIELD field-spec
| ON CHANGE field-spec
| AFTER FIELD field-spec
| AFTER CONSTRUCT
| ON ACTION action-name
| ON KEY ( key-name [,...] )
}
     dialog-statement
     [...]
```

where *display-array-block* is:

```
DISPLAY ARRAY array TO screen-array.*
  [ ATTRIBUTE ( { display-array-control-attribute } [,...] ) ]
  { display-array-control-block  }
  [...]
END DISPLAY
```

where *display-array-control-block* is one of:

```
{ BEFORE DISPLAY
| BEFORE ROW
| AFTER ROW
| AFTER DISPLAY
| ON ACTION action-name
| ON KEY ( key-name [,...] )
}
     dialog-statement
     [...]
```

where *input-array-block* is:

```
INPUT ARRAY array FROM screen-array.*
  [ ATTRIBUTE ( { input-array-control-attribute } [,...] ) ]
  { input-array-control-block  }
  [...]
END INPUT
```

where *input-array-control-block* is one of:

```
{ BEFORE INPUT
| BEFORE ROW
| BEFORE FIELD field-spec
| ON CHANGE field-spec
| AFTER FIELD field-spec
| ON ROW CHANGE
| AFTER ROW
| BEFORE DELETE
| AFTER DELETE
| BEFORE INSERT
| AFTER INSERT
| AFTER INPUT
| ON ACTION action-name
| ON KEY ( key-name [,...] )
}
    dialog-statement
    [...]
```

where *dialog-control-block* is one of:

```
{ BEFORE DIALOG
| ON ACTION action-name
| ON KEY ( key-name [,...] )
| ON IDLE idle-seconds
| COMMAND option-name [ option-comment ] [ HELP help-number ]
| COMMAND KEY ( key-name [,...] ) option-name [ option-comment ] [ HELP
help-number ]
| AFTER DIALOG
}
    dialog-statement
    [...]
```

where *dialog-statement* is one of:

```
{ statement
| ACCEPT DIALOG
| CONTINUE DIALOG
| EXIT DIALOG
| NEXT FIELD { CURRENT | NEXT | PREVIOUS | field-name }
}
```

where *field-list* is:

```
{ field-name
| table-name.*
| table-name.field-name
| screen-array[line].*
| screen-array[line].field-name
| screen-record.*
| screen-record.field-name
} [,...]
```

where *field-spec* is:

```
{ field-name
| table-name.field-name
| screen-array.field-name
| screen-record.field-name
} [,...]
```

where *column-list* is :

```
{ column-name
| table-name.*
| table-name.column-name
} [,...]
```

Notes:

1. *array* is the array of records used by the `DIALOG` statement.
2. *help-number* is an integer that allows you to associate a help message number with the command.
3. *field-name* is the identifier of a field of the current form.
4. *option-name* is a string expression defining the label of the action and identifying the action that can be executed by the user.
5. *option-comment* is a string expression containing a description for the menu option, displayed when *option-name* is the current.
6. *column-name* is the identifier of a database column of the current form.
7. *table-name* is the identifier of a database table of the current form.
8. *variable* is a simple program variable (not a record).
9. *record* is a program record (structured variable).
10. *screen-array* is the screen array that will be used in the current form.
11. *line* is a screen array line in the form.
12. *screen-record* is the identifier of a screen record of the current form.
13. *action-name* identifies an action that can be executed by the user.
14. *idle-seconds* is an integer literal or variable that defines a number of seconds.
15. *key-name* is a hot-key identifier (like `F11` or `Control-z`).
16. *statement* is any instruction supported by the language.

The following table shows the *dialog-control-attributes* supported by the `DIALOG` statement:

Attribute	Description
`FIELD ORDER FORM`	Indicates that the tabbing order of fields is defined by the TABINDEX attribute of form fields. The default order in which the focus moves from field to field is determined by the order of the program variables bound to the `DIALOG` statement. The program options instruction can also change this behavior with `FIELD ORDER FORM` options. Action views can also get the focus when using this option.
`UNBUFFERED [ =bool]`	Indicates that the dialog must be sensitive to

	program variable changes. The *bool* parameter can be an integer literal or a program variable.

The following table shows the *input-control-attributes* supported by the `INPUT` sub-dialog of the `DIALOG` statement:

Attribute	Description
`HELP = int-expr`	Defines the help number of the help message to use when help is invoked by the user.
`NAME = string`	Identifies the sub-dialog by providing a unique name.
`WITHOUT DEFAULTS [=bool]`	Indicates whether or not fields are initially filled with the column default values defined in the form specification file or the database schema files. If set to FALSE, fields are initially populated with the column default values. If set to TRUE, default values are ignored. *bool* can be an integer literal or a program variable that evaluates to TRUE or FALSE.

The following table shows the *display-array-control-attributes* supported by the `DISPLAY ARRAY` sub-dialog of the `DIALOG` statement:

Attribute	Description
`COUNT = row-count`	Defines the number of data rows when using a static array or a dynamic array in paged mode. *row-count* can be an integer literal or a program variable.
`HELP = int-expr`	Defines the help number of the help message to use when help is invoked by the user.
`KEEP CURRENT ROW [=bool]`	Keeps current row highlighted after execution of the instruction. *bool* can be an integer literal or a program variable that evaluates to TRUE or FALSE.

The following table shows the *input-array-control-attributes* supported by the `INPUT ARRAY` sub-dialog of the `DIALOG` statement:

Attribute	Description
`HELP = int-expr`	Defines the help number of the help message to use when help is invoked by the user.
`WITHOUT DEFAULTS [=bool]`	Indicates whether or not fields are initially

	filled with the column default values defined in the form specification file or the database schema files. If set to FALSE, fields are initially populated with the column default values. If set to TRUE, default values are ignored. *bool* can be an integer literal or a program variable that evaluates to TRUE or FALSE.
`COUNT = row-count`	Defines the number of data rows in the static array. *row-count* can be an integer literal or a program variable.
`MAXCOUNT = row-count`	Defines the maximum number of data rows that can be entered in the program array. *row-count* can be an integer literal or a program variable.
`APPEND ROW [ =bool]`	Defines if the user can append new rows at the end of the list. *bool* can be an integer literal or a program variable that evaluates to TRUE or FALSE.
`INSERT ROW [ =bool]`	Defines if the user can insert new rows inside the list. *bool* can be an integer literal or a program variable that evaluates to TRUE or FALSE.
`DELETE ROW [ =bool]`	Defines if the user can delete rows. *bool* can be an integer literal or a program variable that evaluates to TRUE or FALSE.
`AUTO APPEND [ =bool]`	Defines if a temporary row will be created automatically when needed. *bool* can be an integer literal or a program variable that evaluates to TRUE or FALSE.
`KEEP CURRENT ROW [=bool]`	Keeps current row highlighted after execution of the instruction.

The following table shows the *construct-control-attributes* supported by the `CONSTRUCT` sub-dialog of the `DIALOG` statement:

Attribute	Description
`HELP = int-expr`	Defines the help number of the help message to use when help is invoked by the user.
`NAME = string`	Identifies the sub-dialog by providing a unique name.

Usage

Use the DIALOG instruction if you want to handle different parts of a form at the same time. The DIALOG instruction acts as a combination of classical / singular dialogs. The syntax of the DIALOG instruction is very close to singular dialogs, using common triggers such as BEFORE FIELD, ON ACTION, and so on. Despite the similarities, the behavior and semantics of DIALOG are a bit different from singular dialogs.

The following example is of a DIALOG instruction that includes both an INPUT and a DISPLAY ARRAY sub-dialog:

```
01 SCHEMA stores
02
03 DEFINE p_customer RECORD LIKE customer.*
04 DEFINE p_orders DYNAMIC ARRAY OF RECORD LIKE order.*
05
06 FUNCTION customer_dialog()
07
08   DIALOG ATTRIBUTES(UNBUFFERED, FIELD ORDER FORM)
09
10     INPUT BY NAME p_customer.*
11       AFTER FIELD cust_name
12         CALL setup_dialog(DIALOG)
13     END INPUT
14
15     DISPLAY ARRAY p_orders TO s_orders.*
16       BEFORE ROW
17         CALL setup_dialog(DIALOG)
18     END DISPLAY
19
20     ON ACTION close
21       EXIT DIALOG
22
23   END DIALOG
24
25 END FUNCTION
```

The main differences between multiple dialogs and singular dialogs are:

1. While DIALOG supports the compatible BUFFERED mode, it is strongly recommended that you use the UNBUFFERED mode, in order to synchronize views (form fields) with data models (program variables).
2. The DIALOG instruction does not use the INT_FLAG variable. You must implement ON ACTION accept or ON ACTION cancel to handle dialog validation / cancellation. These actions do not exist by default in DIALOG.
3. Unlike singular dialogs creating implicit accept and cancel actions, by default there is no way to quit the DIALOG instruction. You must implement your own action handler and execute EXIT DIALOG or ACCEPT DIALOG.
4. All elements of the dialog are active at the same time, so you must handle tabbing order properly. By default - as in singular dialogs - the tabbing order is driven by the binding list (order of program variables). It is strongly recommended

that you use the `FIELD ORDER FORM` option and the TABINDEX field attributes instead.
5. Like the singular INPUT ARRAY instruction, `DIALOG` creates implicit *insert*, *append* and *delete* actions. These actions are only active when the focus is in the list.

Tips:

1. Don't touch working programs: The purpose of the `DIALOG` instruction is not to replace singular dialogs, which are still supported and useful in most cases.
2. It is recommended that you use singular dialogs if no multiple dialog is required. For example, you would typically implement a master/detail form with `DIALOG`, and execute a singular CONSTRUCT instruction as a nested dialog called from the master/detail `DIALOG`.
3. Write a common *setup_dialog(ui.dialog)* function to centralize all field and action activations according to the context. You can then call that setup function at any place in the `DIALOG` code.
4. While static arrays are supported by the `DIALOG` instruction, it is strongly recommended that you use dynamic arrays instead. With a dynamic array, the actual number of rows is automatically defined by the array variable, while static arrays need an additional step to define the total number of rows.

Programming Steps

The following steps describe how to use the `DIALOG` statement:

1. Create a form specification file containing screen record(s) and/or screen array(s). The screen records and screen arrays identify the presentation elements to be used by the runtime system to display the data models (i.e. the content of program variables bound to the `DIALOG` instruction).
2. With the DEFINE instruction, declare program variables that will be used as data models. For record lists (`DISPLAY ARRAY` or `INPUT ARRAY`), the members of the program array must correspond to the elements of the screen array, by number and data types. To handle record lists, use dynamic arrays instead of static arrays.
3. Open and display the form, using a OPEN WINDOW with the `WITH FORM` clause or the OPEN FORM / DISPLAY FORM instructions.
4. Fill the program variables with data. For lists, you typically use a result set cursor.
5. Implement the `DIALOG` instruction block to handle interaction. Define each sub-dialog with program variables to be used as data models. The sub-dialogs will define how variables will be used (display or input).
6. Inside the `DIALOG` instruction, control the behavior of the instructions with control blocks,such as `BEFORE DIALOG`, `AFTER ROW`, `BEFORE FIELD`, or `ON ACTION`.
7. To end the DIALOG instruction, implement an `ON ACTION close` or `ON ACTION accept` / `ON ACTION cancel` to handle dialog validation and cancellation, with the `ACCEPT DIALOG` and `EXIT DIALOG` control instructions. Note that the INT_FLAG variable will **not** be set as it would in singular dialogs.

Sub-dialogs

A `DIALOG` instruction is made of one or several **sub-dialogs**, plus global control blocks and action handlers. The sub-dialogs bind program variables to form fields and define the type of interaction that will take place for the data model (simple input, list input or query). The sub-dialogs implement individual control blocks which let you control the behavior of the interactive instruction. Sub-dialogs can also hold action handlers, which will define local sub-dialog actions.

There are four types of `DIALOG` sub-dialogs:

1. Simple record input with the INPUT sub-dialog.
2. Read-only record list handling with the DISPLAY ARRAY sub-dialog.
3. Editable record list handling with the INPUT ARRAY sub-dialog.
4. Query By Example handling with the CONSTRUCT sub-dialog.

The program variables can be of any data type; the runtime system will adapt input and display rules to the variable type. For example, if you want to use a DATE variable, the `DIALOG` instruction will check for a valid date value when the user enters a value in the corresponding form field.

You typically define program variables using a LIKE clause, ensuring the form field matches the underlying database column.

If a variable is declared `LIKE` a SERIAL column, the runtime system does not allow the screen cursor to stop in the field, because values in SERIAL columns are automatically generated by the database server. When the user enters data for an `INPUT` or `INPUT ARRAY` instruction, the runtime system checks the entered value against the data type of the variable, not the data type of the form field. The field data types defined in the form, however, are used when doing a `CONSTRUCT`.

Some data validation rules can be defined at the form level, such as NOT NULL, REQUIRED and INCLUDE attributes. Data validation is discussed later in this documentation.

For more details about defining sub-dialogs:

- The INPUT sub-dialog
  - Identifying the INPUT sub-dialog
  - Control blocks in INPUT
- The DISPLAY ARRAY sub-dialog
  - Identifying the DISPLAY ARRAY sub-dialog
  - Control blocks in DISPLAY ARRAY
- The INPUT ARRAY sub-dialog
  - Identifying the INPUT ARRAY sub-dialog
  - Control blocks in INPUT ARRAY
- The CONSTRUCT sub-dialog

- Identifying the CONSTRUCT sub-dialog
- Control blocks in CONSTRUCT
- Query Operators

The INPUT sub-dialog

When using the `INPUT` sub-dialog, you bind each record member variable to the corresponding field of a screen record so the `DIALOG` instruction can manipulate the values that the user enters in the form fields.

The `INPUT` clause can be used in two forms:

1. `INPUT BY NAME variable-list`
2. `INPUT variable-list FROM field-list`

The `BY NAME` clause implicitly binds the fields to the variables that have the same identifiers as the field names. You must first declare variables with the same names as the fields from which they accept input. The runtime system ignores any record name prefix when making the match. The unqualified names of the variables and of the fields must be unique and unambiguous within their respective domains. If they are not, the runtime system generates an exception, and sets the STATUS variable to a negative value.

```
01 DEFINE p_cust RECORD
02        cust_num INTEGER,
03        cust_name VARCHAR(50),
04        cust_address VARCHAR(100)
05    END RECORD
06    ...
07 DIALOG
08    INPUT BY NAME p_cust.*
09       BEFORE FIELD cust_name
10        ...
11    END INPUT
12    ...
13 END DIALOG
```

The `FROM` clause explicitly binds the fields in the screen record to a list of program variables that can be simple variables or records. The number of variables or record members must equal the number of fields listed in the `FROM` clause. Each variable must be of the same (or a compatible) data type as the corresponding screen field. When the user enters data, the runtime system checks the entered value against the data type of the variable, not the data type of the screen field.

```
01 DEFINE custname VARCHAR(50)
02    ...
03 DIALOG
04    INPUT cust_name FROM customer.cust_name
05       BEFORE FIELD cust_name
```

```
06        ...
07    END INPUT
08    ...
09 END DIALOG
```

Identifying an INPUT sub-dialog

The name of an `INPUT` sub-dialog can be used to qualify sub-dialog actions with a prefix.

In order to identify the `INPUT` sub-dialog with a specific name, you can use the `ATTRIBUTES` clause to set the `NAME` attribute:

```
01    INPUT BY NAME p_cust.* ATTRIBUTES (NAME = "cust")
02      ...
```

For more details about the possible attributes, see INPUT ATTRIBUTE clause.

Control blocks in INPUT

Simple record input declared with the `INPUT` sub-dialog can raise the following triggers:

- BEFORE INPUT
- BEFORE FIELD *field-spec*
- ON CHANGE *field-spec*
- AFTER FIELD *field-spec*
- AFTER INPUT

Note that in the singular INPUT instruction, `BEFORE INPUT` and `AFTER INPUT` blocks are typically used as initialization and finalization blocks. In an `INPUT` sub-dialog of a `DIALOG` instruction, `BEFORE INPUT` and `AFTER INPUT` blocks will be executed each time the focus goes to (`BEFORE`) or leaves (`AFTER`) the group of fields defined by this sub-dialog.

The DISPLAY ARRAY sub-dialog

The `DISPLAY ARRAY` sub-dialog binds the members of the flat record (or the primitive member) of an array to the screen-array or screen-record fields specified with the `TO` keyword. The number of variables in each record of the program array must be the same as the number of fields in each screen record (that is, in a single row of the screen array).

You typically bind a program array to a screen-array in order to display a page of records. However, the `DIALOG` instruction can also bind the program array to a simple flat screen-record. In this case, only one record will be visible at a time.

In any case, implicit navigation actions (`firstrow`, `prevrow`, `nextrow` and `lastrow`) are created by the dialog. See also Default Actions for more details about implicit actions. Note that the default action views for these navigation actions are hidden by default. The

actions are automatically enabled and disabled according to the position of the current row in the list.

The next code example defines an array with a flat record and binds it to a screen array:

```
01 DEFINE p_items DYNAMIC ARRAY OF RECORD
02        item_num INTEGER,
03        item_name VARCHAR(50),
04        item_price DECIMAL(6,2)
05    END RECORD
06    ...
07 DIALOG
08    DISPLAY ARRAY p_items TO sa.*
09       BEFORE ROW
10       ...
11    END DISPLAY
12    ...
13 END DIALOG
```

If the screen array is defined with one field only, you can bind an array defined with a primitive type:

```
01 DEFINE p_names DYNAMIC ARRAY OF VARCHAR(50)
02    ...
03 DIALOG
04    DISPLAY ARRAY p_names TO sa.*
05       BEFORE DELETE
06       ...
07    END DISPLAY
08    ...
09 END DIALOG
```

Identifying a DISPLAY ARRAY sub-dialog

The name of the screen array specified with the `TO` clause identifes the list. The dialog class method such as takes the name of the screen array as the parameter, identifying the list. For example, you would use DIALOG.getCurrentRow("screen-array") to query for the current row in the list identified by 'screen-array'. The name of the screen-array is also used to qualify sub-dialog actions with a prefix.

Control blocks in DISPLAY ARRAY

Read-only record lists declared with the `DISPLAY ARRAY` sub-dialog can raise the following triggers:

- BEFORE DISPLAY
- BEFORE ROW
- AFTER ROW
- AFTER DISPLAY

Note that in the singular DISPLAY ARRAY instruction, `BEFORE DISPLAY` and `AFTER DISPLAY` blocks are typically used as initialization and finalization blocks. In a `DISPLAY`

ARRAY sub-dialog of a DIALOG instruction, BEFORE DISPLAY and AFTER DISPLAY blocks will be executed each time the focus goes to (BEFORE) or leaves (AFTER) the group of fields defined by this sub-dialog.

The INPUT ARRAY sub-dialog

The INPUT ARRAY sub-dialog binds the members of the flat record (or the primitive member) of an array to the screen-array or screen-record fields specified with the FROM keyword. The number of variables in each record of the program array must be the same as the number of fields in each screen record (that is, in a single row of the screen array).

Note that you typically bind a program array to a screen-array in order to display a page of records. However, the DIALOG instruction can also bind the program array to a simple flat screen-record. In this case, only one record will be visible at a time.

In any case, implicit navigation actions (firstrow, prevrow, nextrow and lastrow) are created by the dialog. See also Default Actions for more details about implicit actions. Note that the default action views for these navigation actions are hidden by default. The actions are automatically enabled and disabled according to the position of the current row in the list.

The next code example defines an array with a flat record and binds it to a screen array:

```
01 DEFINE p_items DYNAMIC ARRAY OF RECORD
02        item_num INTEGER,
03        item_name VARCHAR(50),
04        item_price DECIMAL(6,2)
05    END RECORD
06    ...
07 DIALOG
08    INPUT ARRAY p_items FROM sa.*
09       BEFORE INSERT
10       ...
11    END INPUT
12    ...
13 END DIALOG
```

If the screen array is defined with one field only, you can bind an array defined with a primitive type:

```
01 DEFINE p_names DYNAMIC ARRAY OF VARCHAR(50)
02    ...
03 DIALOG
04    INPUT ARRAY p_names FROM sa.*
05       BEFORE DELETE
06       ...
07    END INPUT
08    ...
09 END DIALOG
```

Identifying an INPUT ARRAY sub-dialog

The name of the screen array specified with the `FROM` clause will be used to identify the list. For example, the dialog class method such as DIALOG.getCurrentRow("screen-array") takes the name of the screen array as the parameter, to identify the list you want to query for the current row. The name of the screen-array is also used to qualify sub-dialog actions with a prefix.

Control blocks in INPUT ARRAY

Editable record lists declared with the `INPUT ARRAY` sub-dialog can raise the following triggers:

- BEFORE INPUT
- BEFORE ROW
- BEFORE FIELD *field-spec*
- ON CHANGE *field-spec*
- AFTER FIELD *field-spec*
- ON ROW CHANGE
- AFTER ROW
- BEFORE DELETE
- AFTER DELETE
- BEFORE INSERT
- AFTER INSERT
- AFTER INPUT

Note that in the singular INPUT ARRAY instruction, `BEFORE INPUT` and `AFTER INPUT` blocks are typically used as initialization and finalization blocks. In the `INPUT ARRAY` sub-dialog of a `DIALOG` instruction, `BEFORE INPUT` and `AFTER INPUT` blocks will be executed each time the focus goes to (`BEFORE`) or leaves (`AFTER`) the group of fields defined by this sub-dialog.

The CONSTRUCT sub-dialog

The `CONSTRUCT` sub-dialog binds a character string variable with screen fields, to implement Query By Example (QBE). When such a sub-dialog is used, the `DIALOG` instruction produces an SQL condition corresponding to search criteria that a user specifies in the fields. The instruction fills the character variable with the SQL condition, and you can use the content of this variable to create the `WHERE` clause of a `SELECT` statement to query the database.

```
01 DEFINE sql_condition STRING
02    ...
03 DIALOG
04    CONSTRUCT BY NAME sql_condition ON customer.cust_name,
customer.cust_address
05       BEFORE FIELD cust_name
06       ...
07    END CONSTRUCT
```

```
08    ...
09 END DIALOG
```

You must make sure the character string variable is large enough to store all possible SQL conditions. It is better to use a STRING data type to avoid any size problems.

Warnings:

1. `CONSTRUCT` uses the field data types defined in the current form file to produce the SQL conditions. This is different from other interactive instructions, where the data types of the program variables define the way to handle input/display. It is *strongly* recommended (but not mandatory) that the form field data types correspond to the data types of the program variables used for input. This is implicit if both form fields and program variables are based on the database schema file.

The `CONSTRUCT` clause can be used in two forms:

1. `CONSTRUCT BY NAME` *`string-variable`* `ON` *`column-list`*
2. `CONSTRUCT` *`string-variable`* `ON` *`column-list`* `FROM` *`field-list`*

The `BY NAME` clause implicitly binds the form fields to the columns, where the form field identifiers match the column names specified in the column-list after the `ON` keyword. You can specify the individual column names (separated by commas) or use the `tablename.*` shortcut to include all columns defined for a table in the database schema file.

The `FROM` clause explicitly binds the form fields listed after the `FROM` keyword with the column definitions listed after the `ON` keyword.

In both cases, the name of the columns in *column-list* will be used to produce the SQL condition in *string-variable*.

Identifying a CONSTRUCT sub-dialog

The name of a `CONSTRUCT` sub-dialog can be used to qualify sub-dialog actions with a prefix. In order to identify the `CONSTRUCT` sub-dialog with a specific name, use the `ATTRIBUTES` clause to set the `NAME` attribute:

```
01    CONSTRUCT BY NAME sql_condition ON customer.* ATTRIBUTES (NAME =
"q_cust")
02        ...
```

For more details about the possible attributes, see INPUT ATTRIBUTE clause.

Control blocks in CONSTRUCT

A Query By Example declared with the `CONSTRUCT` clause can raise the following triggers:

- BEFORE CONSTRUCT
- BEFORE FIELD *field-spec*
- AFTER FIELD *field-spec*
- AFTER CONSTRUCT

Note that in the singular CONSTRUCT instruction, `BEFORE CONSTRUCT` and `AFTER CONSTRUCT` blocks are typically used as initialization and finalization blocks. In `DIALOG` instruction, `BEFORE CONSTRUCT` and `AFTER CONSTRUCT` blocks will be executed each time the focus goes to (`BEFORE`) or leaves (`AFTER`) the group of fields defined by this sub-dialog.

Query Operators

The following table lists all relational operators that can be used during a Query By Example input:

Symbol	Meaning	Pattern
Any simple data type		
`=`	Is Null	=
`==`	Equal to	== *value*
`>`	Greater than	> *value*
`>=`	Greater than or equal to	>= *value*
`<`	Less than	< *value*
`<=`	Less than or equal to	<= *value*
`<> or !=`	Not equal to	!= *value*, <> *value*
`: or ..`	Range	*value1*:*value2*, *value1*..*value2*
`|`	List of values	*value1* | *value2*
Character data types only		
`*`	Wildcard for any string	*x, x*, *x*
`?`	Single-character wildcard	?x, x?, ?x?, x??
`[c]`	A set of characters	[a-z]*, [xy]?

Programming with `DIALOG`

The following sections describe the concepts you must understand in order to program a `DIALOG` instruction. The following topics are covered:

- Global configuration settings in FGLPROFILE
- Identifying sub-dialogs with a name
- Binding Action Views to Action Handlers in DIALOG
- The Buffered and Unbuffered modes
- The WITHOUT DEFAULTS option
  - WITHOUT DEFAULTS in INPUT
  - WITHOUT DEFAULTS in INPUT ARRAY

- Which form item has the focus?
- The TOUCHED flag of input fields
- Executing form-level validation rules
- Handling the Tabbing Order
- Detecting focus changes
- Detecting data modification immediately
- Defining the total number of rows in a list
- Handling the current row in a list
- The paged mode of read-only lists
- Inserting and deleting rows in a list
- Handling temporary rows
- Implementing the close action

Global settings in FGLPROFILE

By setting global parameters in FGLPROFILE, you can control the behavior of all dialogs of the program. These options are provided as global parameters to define a common pattern for all dialogs of your application. A complete description is available in the Runtime Configuration section.

List of FGLPROFILE entries affecting the behavior of dialogs:

1. Dialog.fieldOrder
2. Dialog.currentRowVisibleAfterSort

Identifying sub-dialogs by a name

The `DIALOG` instruction is a collection of sub-dialogs that act as controllers for different parts of a form. In order to program a DIALOG instruction, there must be a unique identifier for each sub-dialog. For example, to set the current row of a screen array with the setCurrentRow() method, you pass the name of the screen array to specify the sub-dialog to be affected. Sub-dialog identifiers are also used as a prefix to specify actions for the sub-dialog.

The following links describe how to specify the names of the different types of `DIALOG` sub-dialogs:

- Identifying the INPUT sub-dialog
- Identifying the DISPLAY ARRAY sub-dialog
- Identifying the INPUT ARRAY sub-dialog
- Identifying the CONSTRUCT sub-dialog

Binding Action View to Action Handlers in DIALOG

In forms, actions views like buttons are bound to ON ACTION handlers by name. Within the DIALOG instruction, we distinguish *dialog actions* from *sub-dialog actions.* When the action handler is defined as dialog action, the action name is a simple identifier as in singular interactive instructions. When the ON ACTION action handler is defined inside a sub-dialog or if the action is an implicit action such as *insert* in INPUT ARRAY, the action name gets a prefix ("*sub-dialog-name.action-name*") to identify the sub-dialog action with a unique name.

```
01 DIALOG
02    INPUT BY NAME ... ATTRIBUTES (NAME = "cust")
03       ON ACTION suspend -- this is the local sub-dialog action
"cust.suspend"
04       ...
05    END INPUT
06    BEFORE DIALOG
07       ...
08    ON ACTION close -- this is the dialog action "close"
09       ...
10 END DIALOG
```

By using this sub-dialog identifier, you can bind specific action views to sub-dialog actions. For INPUT and CONSTRUCT sub-dialogs, the sub-dialog identifier can be specified with the NAME attribute. The INPUT ARRAY and DISPLAY ARRAY sub-dialogs are implicitly identified with the screen-record name defined in the form.

Note that the sub-dialog prefix is optional in the action view so you can still bind action views with the action name only: any sub-dialog action with the matching name will be attached. This is especially useful for common actions such as the implicit *insert* / *append* / *delete* actions created by INPUT ARRAY, when the dialog handles multiple editable lists. You can bind toolbar buttons to these action without the sub-dialog prefix: The buttons will apply to the current list that has the focus.

Concerning action views bound to sub-dialog actions without the sub-dialog qualifier, the runtime system automatically enables or disables the actions when entering or leaving the group of fields controlled by the sub-dialog (i.e. typical navigation buttons in the toolbar will be disabled if the focus is not in a list). However, action views bound to sub-dialog actions with qualified sub-dialog action names will always be active.

If a sub-dialog action is fired when the focus is not in the sub-dialog of the action, the focus will automatically be given to the first field of the sub-dialog, before executing the user code defined in the ON ACTION clause. This will trigger the same validation rules and control blocks as if the user had selected the first field of the sub-dialog by hand (see Control Block Execution Order for more details).

When using ui.Dialog.setActionActive() (or any method that takes an action name as parameter), you can specify the action name with or without a sub-dialog identifier. If you qualify the action with the sub-dialog identifier, the sub-dialog action is clearly identified. If you don't specify a sub-dialog prefix, the action will be identified based on the focus context. When the focus is in the sub-dialog of the action, non-qualified action names

identify the local sub-dialog action; otherwise, they identify a dialog action if one exists with the same name.

Note that disabling an action by program with ui.Dialog.setActionActive(), will take precedence over the built-in activation rules (i.e. if the action is disabled by program, the action will not be activated when entering the sub-dialog).

For action views bound to sub-dialog actions with qualifiers, the Action Defaults defined with the corresponding action name will be used to set the attributes with the default values. In other words, the prefix will be ignored. For example, if an action view is defined with the name "*custlist.append*", it will get the action defaults defined for the "*append*" action.

See also The Interaction Model for information about binding Action Views and Action Handlers.

The Buffered and Unbuffered modes

The variables act as a data model to display data or to get user input through the `DIALOG` instruction. Always use the variables if you want to change some field values programmatically.

When you use the default "buffered" mode, program variable changes are not automatically displayed to form fields; you need to execute `DISPLAY TO` or `DISPLAY BY NAME`. Additionally, if an action is triggered, the value of the current field is not validated and is not copied into the corresponding program variable. The only way to get the text of a field is to use GET_FLDBUF() or DIALOG.getFieldBuffer(). Note that these functions return the current text, which might not be a valid representation of a value of the field datatype.

When you use the `UNBUFFERED` attribute, program variables and form fields are automatically synchronized, and the instruction is sensitive to program variable changes: You don't need to display values explicitly with `DISPLAY TO` or `DISPLAY BY NAME`. When an action is triggered, the value of the current field is validated and is copied into the corresponding program variable. If you need to display new data during the `DIALOG` execution, just assign the values to the program variables; the runtime system will automatically display the values to the screen:

```
01 DIALOG ATTRIBUTES(UNBUFFERED)
02  INPUT BY NAME p_items.*
03    ON CHANGE code
04       IF p_items.code = "A34" THEN
05          LET p_items.desc = "Item A34"
06       END IF
07    ...
08  END INPUT
09 END DIALOG
```

During data input, values entered by the user in form fields are automatically validated and copied into the program variables. Actually the value entered in form fields is first available in the form field buffer. This buffer can be queried with built-in functions or dialog class methods. When you use the `UNBUFFERED` mode, the field buffer is used to synchronize program variables each time control returns to the runtime system - for example, when the user clicks on a button to execute an action.

When you use the `UNBUFFERED` mode, you may want to prevent data validation for some actions like *cancel* or *close*. To avoid field validation for a given action, you can set the `validate` Action Default attribute to "no", in the **.4ad** file or in the ACTION DEFAULTS section of the form file:

```
01 ACTION DEFAULTS
02    ACTION undo (TEXT = "Undo", VALIDATE = NO)
03    ...
04 END
```

Note that some predefined actions are already configured with `validate=no` in the **default.4ad** file.

If field validation is disabled for an action, the code executed in the `ON ACTION` block acts as if the dialog was in `BUFFERED` mode: The program variable is not set; however, the input buffer of the current field is updated. When returning from the user code, the dialog will not synchronize the form fields with program variables, and the current field will display the input buffer content. Therefore, if you change the value of the program variable during an `ON ACTION` block where validation is disabled, you must explicitly DISPLAY the values to the fields.

To illustrate this case, imagine that you want to implement an *undo* action to allow the modifications done by the user to be reverted (before these have been saved to the database of course). You typically copy the current record into a clone variable when the dialog starts, and copy these old values back to the input record when the *undo* action is fired. An *undo* action is a good candidate to avoid field validation, since you want to ignore current values. If you don't re-display the values, the input buffer of the current field will remain when returning from the `ON ACTION` block:

```
01 DIALOG ATTRIBUTES(UNBUFFERED)
02  INPUT BY NAME p_cust.*
03    BEFORE INPUT
04      LET p_cust_copy.* = p_cust.*
05    ON ACTION undo -- Defined with VALIDATE=NO
06      LET p_cust.* = p_cust_copy.*
07      DISPLAY BY NAME p_cust.*
08  END INPUT
09 END DIALOG
```

The WITHOUT DEFAULTS option

The INPUT and INPUT ARRAY sub-dialogs support the WITHOUT DEFAULTS option in the binding clause or as an ATTRIBUTE. When used in the syntax of the binding clause, the option is defined statically at compile time as TRUE. When used as an ATTRIBUTE option, it can be specified with an integer expression that is evaluated when the DIALOG interactive instruction starts:

```
01   INPUT BY NAME p_cust.* ATTRIBUTE (WITHOUT DEFAULTS = NOT new)
03    ...
03   END INPUT
```

The WITHOUT DEFAULTS clause in INPUT

In the default mode, the INPUT sub-dialog clears the program variables and assigns the values defined by the DEFAULT attribute in the form file (or indirectly, the default value defined in the database schema files). This mode is typically used to input and INSERT a new record in the database. The REQUIRED field attributes are checked to make sure that the user has entered all data that is mandatory. Note that REQUIRED only forces the user to enter the field, and can leave the value NULL unless the NOT NULL attribute is used. Therefore, if you have an AFTER FIELD or ON CHANGE control block with validation rules, you can use the REQUIRED attribute to force the user to enter the field and trigger that block.

In contrast, the WITHOUT DEFAULTS option starts the dialog with the existing values of program variables. This mode is typically used in order to UPDATE an existing database row. Existing values are considered valid, thus the REQUIRED attributes are ignored when this option is used.

The WITHOUT DEFAULTS clause in INPUT ARRAY

With the INPUT ARRAY sub-dialog, the WITHOUT DEFAULT clause defines whether the program array is populated when the dialog begins. Once the dialog is started, existing rows are always handled as records to be updated in the database (i.e. WITHOUT DEFAULTS=TRUE), while newly created rows are handled as records to be inserted in the database (i.e. WITHOUT DEFAULTS=FALSE).

It is unusual to implement an INPUT ARRAY sub-dialog with no WITHOUT DEFAULTS option, because the data of the program variable would be cleared and the list empty. So, you typically use the WITHOUT DEFAULT clause in INPUT ARRAY. Note this is the default in INPUT ARRAY used inside DIALOG, but in singular INPUT ARRAY, the default is WITHOUT DEFAULTS=FALSE.

Which form item has the focus?

Since several parts of a form can now be active at the same time, you may need to know which form item is current. For example, if you have several lists driven by multiple

DISPLAY ARRAY sub-dialogs, and you want to implement a *clear* action for each list with a unique `ON ACTION clear` block, you need to query the dialog for the current list.

To get the name of the current form item, use the DIALOG.getCurrentItem() method. This method is the new version of the former fgl_dialog_getfieldname() built-in function. It has been extended to return identifiers for fields, lists or actions identifiers.

```
01 DIALOG ATTRIBUTES(UNBUFFERED)
03    DISPLAY ARRAY p_orders TO orders.*
03      ...
04    END DISPLAY
05    DISPLAY ARRAY p_items TO items.*
06      ...
07    END DISPLAY
08    ON ACTION clear
09       IF DIALOG.getCurrentItem() == "items" THEN
10          ...
11       END IF
12    ...
13 END DIALOG
```

Note that you can also detect when you enter or leave a field or a group of fields by using control blocks such as `BEFORE INPUT` or `AFTER DISPLAY`. See Detecting focus changes for more details.

The TOUCHED flag of input fields

Each input field has a *TOUCHED* flag: This flag is used to execute form-level validation rules and trigger `ON CHANGE` blocks. The flag can also be queried to detect if a field was touched/changed during the `DIALOG` instruction, for example with the FIELD_TOUCHED() operator or with ui.Dialog.getFieldTouched("*field-name*").

The touched flag is set to `TRUE` when the user enters data in a field, or when the program executes a DISPLAY TO / DISPLAY BY NAME instruction. The flag can be set to `TRUE` or reset to `FALSE` with the ui.Dialog.setFieldTouched("*field-name*", *value*) method.

The touched flags of all fields are automatically reset by the interactive instruction in the following cases:

- When a `DIALOG` instruction starts, all touched flags are set to `FALSE`.
- With an INPUT sub-dialog or a CONSTRUCT sub-dialog, the touched flags are reset to `FALSE` when entering the group of fields.
- With an INPUT ARRAY sub-dialog, the touched flags are reset to `FALSE` when moving to another row or when creating a new row.
- With a DISPLAY ARRAY sub-dialog, the touched flags are set to `TRUE` for all fields. Read the warning below:

Warning: When using a `DISPLAY ARRAY` sub-dialog, the touched flags are set to `TRUE` for all fields. This behavior exists because of backward compatibility (actually `DISPLAY ARRAY` acts like a `DISPLAY BY NAME` concerning the touched flag). You should not have to use the touched flags in `DISPLAY ARRAY`, since this sub-dialog does not allow data input. However, you must pay attention when implementing nested dialogs, because `DISPLAY ARRAY` will set the touched flags of the fields driven by the parent dialog, for example when executing a `DISPLAY ARRAY` from an `INPUT ARRAY`.

You can query the touched flags with the ui.Dialog.getFieldTouched("*field-name*") method. Note that this flag can be queried in the AFTER INPUT, AFTER CONSTRUCT, AFTER INSERT or AFTER ROW control blocks.

To emulate user input by program or to reset the touched flags after data was saved in the database, you might want to reset touched flags with a call to ui.Dialog.setFieldTouched("*field-name*", FALSE).

Note that when using a list driven by an INPUT ARRAY binding, a temporary row added at the end of the list will be automatically removed if none of the touched flags is set.

For typical EDIT fields, the touched flag is set when leaving the field. If you want to detect data modification earlier, you should use the dialogtouched predefined action. However, this event is only an indicator that the user started to modify a field, the value will not be available in the program variables.

Executing form-level validation rules

Form-level validation rules can be defined for each field with form specification attributes such as NOT NULL, REQUIRED and INCLUDE. These attributes are part of the business rules of the application and must be checked before saving data into the database.

Implicit validation rule checking

The `DIALOG` instruction automatically executes form-level validation rules in the following cases:

- The NOT NULL attribute is satisfied if a value is in the field.
  `NOT NULL` is checked:
  - when the user moves to a different row in a list controlled by an `INPUT ARRAY`;
    Note: If the row is temporary and none of the fields is touched, the attribute is ignored.
  - when focus leaves the sub-dialog controlling the field;
  - when `NEXT FIELD` gives the focus to a field in a different sub-dialog than the current sub-dialog.
  - when the `DIALOG` instruction ends with `ACCEPT DIALOG`.

- The REQUIRED attribute is satisfied if the field is touched, if a DEFAULT value is defined, or if the WITHOUT DEFAULTS option is used.
  `REQUIRED` is checked:
  - when the user moves to a different row in a list controlled by an `INPUT ARRAY`;
    Note: If the row is temporary and none of the fields is touched, the attribute is ignored.
  - when focus leaves the sub-dialog controlling the field;
  - when `NEXT FIELD` gives the focus to a field in a different sub-dialog than the current sub-dialog.
  - when the `DIALOG` instruction ends with `ACCEPT DIALOG`.
- The INCLUDE attribute is satisfied if the value is in the list defined by the attribute.
  `INCLUDE` is checked when the target program variable must be assigned. This happens:
  - when UNBUFFERED mode is used, focus is in the field, and an action is fired;
  - when the focus leaves the field;
  - when the user moves to a different row in a list controlled by an `INPUT ARRAY`;
    Note: If the row is temporary and none of the fields is touched, the attribute is ignored.
  - when focus leaves the sub-dialog controlling the field;
  - when `NEXT FIELD` gives the focus to a field in a different sub-dialog than the current sub-dialog.
  - when the `DIALOG` instruction ends with `ACCEPT DIALOG`.

Note that automatic validation occurs when the focus leaves a sub-dialog of the `DIALOG` instruction.

Explicit validation rule checking

The `DIALOG` instruction can be used as in singular interactive instructions, with the typical OK / Cancel buttons (i.e. accept / cancel actions) to finish the instruction. This lets the user input or modify one record at a time, and program flow must re-enter the `DIALOG` instruction to edit or create another record. To implement this, you can use the default behavior of the `DIALOG` instruction, and have it execute the form-level validation rules automatically when focus is lost for a sub-dialog or when leaving the dialog with ACCEPT DIALOG (raised by the OK button). However, you may want to stay in the `DIALOG` instruction and let the user input / modify multiple records. In this case, you need a way to execute the form-level validation rules defined for each field, before saving the data to the database. Form-level validation rules are defined by the NOT NULL, REQUIRED and INCLUDE attributes.

To validate a sub-set of fields controlled by the `DIALOG` instruction, use the ui.Dialog.validate("field-list") method, as shown in the following example:

```
01    ON ACTION save
02       IF DIALOG.validate("cust.*") < 0 THEN
03         CONTINUE DIALOG
```

```
04        END IF
05        CALL customer_save()
```

Note that this method automatically displays an error message and registers the next field in case of error. It is mandatory to execute a `CONTINUE DIALOG` instruction if the function returns an error.

Handling the Tabbing Order

The FIELD ORDER dialog attribute defines the way tabbing order works. Tabbing order can be based on the dialog binding list (`FIELD ORDER CONSTRAINED`, the default) or it can be based on the form tabbing order (`FIELD ORDER FORM`). It is recommended that you use the `FIELD ORDER FORM` option, to use the tabbing order specified in the form file.

The TABINDEX attribute allows tabbing order in the form to be defined for each form item. By default, the form compiler assigns a tabbing index for each form item according to the position of the item in the layout. Form items that can get the focus are form fields, read-only lists, editable list cells.

If a form item is hidden or disabled, it is removed from the tabbing list. If the user presses tab (or shift-tab), the focus will go to the next (or previous) element that is visible and activated.

Note that the tabbing position of a read-only list driven by a `DISPLAY ARRAY` binding is defined by the `TABINDEX` of the first field.

The NEXT FIELD instruction can also use the tabbing order, when executing `NEXT FIELD NEXT` and `NEXT FIELD PREVIOUS`.

Note that if the form uses a TABLE container, the front-end resets the tab indexes when the user moves columns around. This way, the visual column order always corresponds to the input tabbing order. If the order of the columns in an editable list shouldn't be changed, you can freeze the table columns with the UNMOVABLECOLUMNS attribute.

Note also that the `DIALOG` instruction creates two implicit actions to tab out of an INPUT ARRAY list with *control-tab* and *shit-control-tab* accelerators. See Predefined Actions for more details.

Detecting focus changes

We have seen that a `DIALOG` block can handle different parts of a form simultaneously. You may want to execute some code when a part of the form gets (or loses) the focus.

To detect when a group of fields, a row or a specific field gets the focus, use the following control blocks:

- BEFORE INPUT (a field of this `INPUT` or `INPUT ARRAY` sub-dialog gets the focus and none of its fields had focus before)
- BEFORE CONSTRUCT (a field of this `CONSTRUCT` sub-dialog gets the focus and none of its fields had focus before)
- BEFORE DISPLAY (this `DISPLAY ARRAY` sub-dialog gets the focus and none of its fields had focus before)
- BEFORE ROW (a new row gets the focus inside a `DISPLAY ARRAY` or `INPUT ARRAY` list)
- BEFORE FIELD (a specific field (or group of fields) gets the focus)

Note that these triggers are also executed by NEXT FIELD.

To detect when a specific field, a row or a group of fields loses the focus:

- AFTER FIELD (the field (or group of fields) loses focus)
- AFTER ROW (a row inside a `DISPLAY ARRAY` or `INPUT ARRAY` list loses focus)
- AFTER DISPLAY (this `DISPLAY ARRAY` sub-dialog loses the focus = focus goes to another sub-dialog)
- AFTER CONSTRUCT (this `CONSTRUCT` sub-dialog loses the focus = focus goes to another sub-dialog)
- AFTER INPUT (this `INPUT` or `INPUT ARRAY` sub-dialog loses focus = focus goes to another sub-dialog)

Detecting data modification immediately

Unlike singular interactive instruction such as INPUT that are quit frequently with a *cancel* or *accept* action, the `DIALOG` instruction can be used continuously for several data operations, such as navigation, creation, or modification. In this case the form is in navigation mode by default; as soon as the user starts to modify a field, it switches to modification mode. You typically want to get control and enable a *save* action when the user starts to modify the current record. To do this, you can use the "dialogtouched" predefined action.

If you want to use this feature, just create an `ON ACTION dialogtouched` block as in the next example:

```
01  DIALOG
02    ...
03    ON ACTION dialogtouched
04       LET changing = TRUE
05       CALL DIALOG.setActionActive("dialogtouched", FALSE)
06    ...
07  END DIALOG
```

When such an action is defined, the front-end knows that it must send the action as soon as the current field is modified (without leaving the field). Note that you must disable / enable this special action in accordance with the status of the dialog: If this action is enabled, the `ON ACTION` block will be fired each time the user modifies the value in the current field.

Defining the total number of rows in a list

Before dynamic arrays, the language only supported static arrays, and you had to specify the number of rows the `DISPLAY ARRAY` and `INPUT ARRAY` interactive instructions must handle. When using a static array, you must specify the actual number of rows with the SET_COUNT() built-in function or with the COUNT attribute. Both of them are only taken into account when the interactive instruction starts. Further, when using multiple lists in `DIALOG`, the `SET_COUNT()` built-in function is unusable, as it defines the total number of rows for all lists. Thus, the only way to define the number of rows when using a static array in multiple dialogs is to use the `COUNT` attribute. Remember the `COUNT` attribute is only taken into account when the dialog starts.

Even if `DIALOG` is able to handle static arrays, it is strongly recommended that you use dynamic arrays in `DISPLAY ARRAY` or `INPUT ARRAY` sub-dialogs. When using a dynamic array, the total number of rows is automatically defined by the array variable (i.e. by array.getLength()). Note that special consideration has to be taken when using the paged mode. In this mode the dynamic array only holds a page of the row set: You must specify the total number of rows with the ui.Dialog.setArrayLength() method.

See Example 1 for dynamic array usage with `DIALOG`.

The paged mode of read-only lists

When implementing a read-only list with a DISPLAY ARRAY sub-dialog, it is possible to use the paged mode by using the `ON FILL BUFFER` block. The *paged mode* allows the program to display a very large number of rows without copying all database rows into the program array. Lists are actually displayed by pages; the `DIALOG` instruction executes the instructions in the `ON FILL BUFFER` block when it needs a given page of rows. You must use a dynamic array to implement a paged list.

In paged mode, the dynamic array holds a page of rows, not all rows of the result set. Therefore, you must specify the total number of rows with the `COUNT` attribute in the `ATTRIBUTES` clause of `DISPLAY ARRAY`. The number of rows can be changed during dialog execution with the ui.Dialog.setArrayLength() method. Note that in singular `DISPLAY ARRAY` instructions, you define the total number of rows of a paged mode with the SET_COUNT() built-in function or the COUNT attribute. But these are only taken into account when the dialog starts. If the total number of rows changes during the execution of the dialog, the only way to specify the number of rows is `setArrayLength()`.

The ON FILL BUFFER clause is used to fill a page of rows in the dynamic array, according to an offset and a number of rows. The offset can be retrieved with the FGL_DIALOG_GETBUFFERSTART() built-in function, and the number of rows to provide is defined by the FGL_DIALOG_GETBUFFERLENGTH() built-in function.

A typical paged DISPLAY ARRAY implementation consists of a scroll cursor providing the list of records to be displayed. Scroll cursors use a static result set. If you want to display fresh data, you can implement an advanced paged mode by using a scroll cursor that provides the primary keys of the referenced result set, plus a prepared cursor to fetch rows on demand in the ON FILL BUFFER clause. In this case you may need to check whether a row still exists when fetching a record with the second cursor.

The following example shows a DISPLAY ARRAY implementation using a scroll cursor to fill pages of records in ON FILL BUFFER:

```
01 MAIN
02   DEFINE arr DYNAMIC ARRAY OF RECORD
03              id INTEGER,
04              fname CHAR(30),
05              lname CHAR(30)
06          END RECORD
07   DEFINE cnt, ofs, len, i INTEGER
08   DATABASE stores7
09   OPEN FORM f1 FROM "custlist"
10   DISPLAY FORM f1
11   SELECT COUNT(*) INTO cnt FROM customer
12   DECLARE c1 SCROLL CURSOR FOR
13           SELECT customer_num, fname, lname FROM customer
14   OPEN c1
15   DIALOG ATTRIBUTES(UNBUFFERED)
16     DISPLAY ARRAY arr TO sa.* ATTRIBUTES(COUNT=cnt)
17       ON FILL BUFFER
18         LET ofs = fgl_dialog_getBufferStart()
19         LET len = fgl_dialog_getBufferLength()
20         FOR i=1 TO len
21           FETCH ABSOLUTE ofs+i-1 c1 INTO arr[i].*
22         END FOR
23     END DISPLAY
24     ON ACTION ten_first_rows_only
25       CALL DIALOG.setArrayLength("sa", 10)
26     ON ACTION quit
27        EXIT DIALOG
28   END DIALOG
29 END MAIN
```

Handling the current row in a list

To set the current row in a list driven by a DISPLAY ARRAY or INPUT ARRAY sub-dialog, use the ui.Dialog.setCurrentRow("*screen-array*", *pos*) method:

```
01 DIALOG ATTRIBUTES(UNBUFFERED)
```

```
02    DISPLAY ARRAY p_items TO sa.*
03    ...
04    ON ACTION move_to_x
05       LET row = askForNewRow()
06       CALL DIALOG.setCurrentRow("sa", row)
07       NEXT FIELD item_num
08    ...
09 END DIALOG
```

Calling the `setCurrentRow()` method will not execute control blocks such as `BEFORE ROW` and `AFTER ROW`, and will not set the focus. If you want to set the focus to the list, you must use the `NEXT FIELD` instruction (this works with `DISPLAY ARRAY` as well as with `INPUT ARRAY`).

The method to query the current row of a list is ui.Dialog.getCurrentRow("*screen-array*").

The singular-dialog specific functions FGL_SET_ARR_CURR(), ARR_CURR() and ARR_COUNT() are also supported. These functions work in the context of the current list having the focus. Note however that `FGL_SET_ARR_CURR()` triggers control blocks such as `BEFORE ROW`, while `ui.Dialog.setCurrentRow()` does not trigger any control block.

Inserting and deleting rows in a list

You can implement an editable record list by using an INPUT ARRAY sub-dialog. This controller allows the user to directly edit existing rows and to create or remove rows with implicit actions.

New rows can be created with *append* or *insert* actions, and can be removed with the *delete* action. These three implicit actions are automatically created by the `DIALOG` instruction (you do not write ON ACTION blocks for these actions):

- The *insert* action will insert a new row before the current row. If there are no rows in the list, *insert* adds a new row.
- The *append* action creates a new row after the last row of the list.
- The *delete* action deletes the current row.

Each of the implicit actions can be prevented by setting the APPEND ROW, INSERT ROW and/or DELETE ROW control attributes to `FALSE`:

```
01 DIALOG ATTRIBUTES(UNBUFFERED)
02    INPUT ARRAY p_items FROM sa.* ATTRIBUTES(APPEND ROW=FALSE, INSERT
ROW=FALSE, DELETE ROW=FALSE)
03    ...
04    END INPUT
05 END DIALOG
```

Specific control blocks (predefined triggers) are available to take control when a row is deleted or created:

- BEFORE DELETE and AFTER DELETE control blocks can be used with row deletion.
  You can cancel row deletion with the CANCEL DELETE instruction in `BEFORE DELETE`.
- BEFORE INSERT and AFTER INSERT control blocks can be used with row creation.
  You can cancel a row creation with CANCEL INSERT in `BEFORE INSERT` or `AFTER INSERT` blocks.

The dynamic arrays and the ui.Dialog class provide methods such as `array.deleteElement()` or `ui.Dialog.appendRow()` to modify the list. When using these methods, the predefined triggers described above are not executed. While it is safe to use these methods within a `DISPLAY ARRAY`, you must take care when using an `INPUT ARRAY`. For example, you should not call such methods in triggers like BEFORE ROW, AFTER INSERT, BEFORE DELETE.

Users can append *temporary* rows by moving to the end of the list or by executing the *append* action. Appending temporary rows is a bit different from doing an *insert* action, because the row is considered temporary until the user modifies a field; inserted rows are not temporary, they are permanent. For more details, see Temporary rows below.

Note that by default (i.e. `AUTO APPEND` is not defined as `FALSE`), when the last row is removed by a *delete* action, the interactive instruction will automatically create a new temporary row at the same position. The visual effect of this behavior can be misinterpreted - if no data was entered in the last row, you can't see any difference. However, the last row is really deleted and a new row is created, and the `BEFORE DELETE` / `AFTER DELETE` / `AFTER ROW` / `BEFORE ROW` / `BEFORE INSERT` control block sequence is executed.

The *insert*, *append* or *delete* actions might be automatically disabled according to the context: If the `INPUT ARRAY` is using a static array that is full, or if the MAXCOUNT attribute is reached, both *insert* and *append* actions will be disabled. The *delete* action is automatically disabled if AUTO APPEND = FALSE and there are no more rows in the array.

Handling temporary rows

If a list is driven by an `INPUT ARRAY` sub-dialog, the user can create a new temporary row at the end of the list. The new row is called "*temporary*" because it will be removed if the user leaves the row without entering data. The row is kept if the TOUCHED flag of a field is set (from user input or by program) or when moving down to the next new temporary row. This is different from adding new rows with the ui.Dialog.appendRow() method; in that case, the row is considered permanent and remains in the list even if the user did not enter data in fields.

We distinguish *automatic* temporary row creation from *explicit* temporary row creation:

Automatic temporary row creation

By default, to follow the traditional behavior of singular INPUT ARRAY instructions, automatic temporary row creation takes place when:

- An `INPUT ARRAY` list has no rows and the list gets the focus. A new row is created to let the user enter data immediately.
- The user tries to move below the last row, with a DOWN keystroke or with the mouse.
- The user presses the TAB key on the last field in the row.
- The list has the focus and the last row of the list is deleted by an implicit *delete* action.
- The list has the focus and the last row of the list is deleted by a ui.Dialog.deleteRow() or ui.Dialog.deleteAllRows() call.

  An automatic temporary row is created even if `APPEND ROW` / `INSERT ROW` attributes are set to `FALSE`. This is the traditional behavior, to let the `INPUT ARRAY` continue. Otherwise, the singular interactive instruction would stop as there are no rows for the user to edit.

Explicit temporary row creation

Explicit temporary row creation takes place when the user decided to append a new row. The following user actions will trigger a temporary row creation (as long as the `APPEND ROW` attribute is `TRUE` - the default):

- When the implicit *append* action is fired, for example by pressing a button bound to the *append* action. Note that when no rows exist in the list, an *insert* action will have the same effect as an *append* action (i.e. a temporary row will be created at position 1).

Temporary row creation is useful because, in most cases, `INPUT ARRAY` is used to edit existing rows and append new rows at the end of the list. However, you might want to fully deny row addition or at least avoid the *automatic* temporary row creation when the last row is deleted or when an empty list gets the focus.

To fully deny the row addition, set the APPEND ROW attribute to `FALSE` attribute in the `ATTRIBUTE` clause of the `INPUT ARRAY` sub-dialog.

To avoid automatic temporary row creation, set the AUTO APPEND attribute to `FALSE`:

```
01 DIALOG ATTRIBUTES(UNBUFFERED)
02    INPUT ARRAY p_items FROM sa.* ATTRIBUTES(AUTO APPEND=FALSE)
03    ...
04    END INPUT
05 END DIALOG
```

In order to control row creation, the `DIALOG` instruction provides the BEFORE INSERT and AFTER INSERT control blocks. The `BEFORE INSERT` trigger is fired after a new row

was inserted or appended, just before the user gets control to enter data in fields. The `AFTER INSERT` block is fired if the user leaves the new row (i.e. when the focus moves to another row or leaves the list), or if the dialog is validated with ACCEPT DIALOG. Note that the `AFTER INSERT` block will also be fired if the user did not enter data, but then the temporary row is automatically deleted.

In the BEFORE INSERT control block, you can tell if a row is a temporary appended one by comparing the current row (DIALOG.getCurrentRow() or ARR_CURR()) with the total number of rows (DIALOG.getArrayLength() or ARR_COUNT()). If current row equals the row count, then you are in a temporary row.

See also BEFORE INSERT and AFTER INSERT for more details.

Implementing the close action

The *close* action is a predefined action used for the X cross button in the upper-right corner of graphical windows. Unlike singular interactive instructions, the `DIALOG` instruction does not create an implicit *close* action.

By default, the `DIALOG` instruction maps the *close* action to the `ON ACTION cancel` block, if such a block is defined. If an `ON ACTION close` block is defined, it is executed instead of the `ON ACTION cancel` block. This behavior is implemented to execute the cancel code automatically when the user closes the graphical window.

Note that the default action view of the *close* action is hidden.

For more details, read Windows closed by the user.

DIALOG and sub-dialog configuration clauses

This sections describes the ATTRIBUTES clause attributes that can be used to configure a `DIALOG` instruction and its sub-dialogs:

- DIALOG ATTRIBUTES clause
  - FIELD ORDER option
  - UNBUFFERED option
- INPUT ATTRIBUTES clause
  - NAME option
  - HELP option
  - WITHOUT DEFAULTS option
- DISPLAY ARRAY ATTRIBUTES clause
  - HELP option
  - COUNT option
- INPUT ARRAY ATTRIBUTES clause
  - INSERT ROW option

    - APPEND ROW option
    - DELETE ROW option
    - AUTO APPEND option
    - COUNT option
    - MAXCOUNT option
    - HELP option
    - KEEP CURRENT ROW option
    - WITHOUT DEFAULTS option
- CONSTRUCT ATTRIBUTES clause
    - NAME option
    - HELP option

DIALOG ATTRIBUTES clause

The `ATTRIBUTES` clause specifications override all default attributes and temporarily override any display attributes that the OPTIONS or the OPEN WINDOW statement specified for these fields.

FIELD ORDER FORM option

By default, the form tabbing order is defined by the variable list in the binding specification. You can control the tabbing order by using the `FIELD ORDER FORM` attribute; when this attribute is used, the tabbing order is defined by the TABINDEX attribute of the form items.

The field order mode can also be specified globally with the OPTIONS FIELD ORDER instruction.

With `FIELD ORDER FORM`, if the user changes the focus from field A to a distant field X with the mouse, the dialog does not execute the `BEFORE FIELD` / `AFTER FIELD` triggers of intermediate fields which appear in the binding specification between field A and field X. If the default `FIELD ORDER CONSTRAINT` mode is used, all intermediate triggers are executed unless you set the Dialog.fieldOrder FGLPROFILE entry to false (this entry is actually ignored when using `FIELD ORDER FORM`).

See also Handling the Tabbing Order.

UNBUFFERED option

The `UNBUFFERED` attribute indicates that the dialog must be sensitive to program variable changes. When using this option, you bypass the compatible "buffered" mode.

Note that the "unbuffered" mode can be set globally for all `DIALOG` instructions with a ui.Dialog class method:

```
01 CALL ui.Dialog.setDefaultUnbuffered(TRUE)
02 DIALOG -- Will work in UNBUFFERED mode
03    ...
```

```
04 END DIALOG
```

See also Buffered and Unbuffered mode.

INPUT ATTRIBUTES clause

NAME option

The `NAME` attribute can be used to identify the `INPUT` sub-dialog, especially useful to qualify sub-dialog actions.

HELP option

The `HELP` attribute defines the help number of the text to be displayed when invoked and focus is in one of the fields controlled by the `INPUT` sub-dialog.

WITHOUT DEFAULTS option

By default, sub-dialogs use the default values defined in the form files. If you want to use the values stored in the program variables bound to the dialog, you must use the `WITHOUT DEFAULTS` attribute. For more details see WITHOUT DEFAULTS option.

DISPLAY ARRAY ATTRIBUTES clause

HELP option

The `HELP` attribute defines the help number of the text to be displayed when invoked and focus is in the list controlled by the `DISPLAY ARRAY` sub-dialog.

COUNT option

The `COUNT` attribute defines the number of valid rows in the static array to be displayed as default rows. If you do not use the `COUNT` attribute, the runtime system cannot determine how much data to display, so the screen array remains empty. The `COUNT` option is ignored when using a dynamic array, unless page mode is used. In this case, the `COUNT` attribute must be used to define the total number of rows, because the dynamic array will only hold a page of the entire row set. If the value of `COUNT` is negative or zero, it defines an empty list.

INPUT ARRAY ATTRIBUTES clause

INPUT ARRAY specific attributes can be defined in the ATTRIBUTE clause of the sub-dialog header:

APPEND ROW option

The `APPEND ROW` attribute can be set to FALSE to avoid the automatic creation of the *append* default action, and prevent the user from adding rows at the end of the list. However, even if `APPEND ROW = FALSE`, the user can still insert rows in the middle of the list. Use the `INSERT ROW` attribute to prevent the user from inserting rows.

INSERT ROW option

The `INSERT ROW` attribute can be set to FALSE to avoid the automatic creation of the 'insert' default action, and prevent the user from inserting new rows in the middle of the list. However, even if `INSERT ROW = FALSE` , the user can still append rows at the end of the list. Use the `APPEND ROW` attribute to prevent the user from appending rows.

DELETE ROW option

When the `DELETE ROW` attribute is set to FALSE, the default *delete* action is not created, so the user cannot delete rows from the list. It is possible, however, to create new rows, unless the `INSERT ROW` and `APPEND ROW` attributes are set to `FALSE` as well.

AUTO APPEND option

By default, an `INPUT ARRAY` controller creates a temporary row when needed. For example, when the user deletes the last row of the list, an new row will be automatically created. You can prevent this default behavior by setting the `AUTO APPEND` attribute to FALSE. If this attribute is set to TRUE, the only way to create a new temporary row is to execute the *append* action. For more details, see Temporary Rows.

COUNT option

The `COUNT` attribute defines the number of valid rows in the static array to be displayed as default rows. If you do not use the `COUNT` attribute, the runtime system cannot determine how much data to display, so the screen array remains empty. The `COUNT` option is ignored when using a dynamic array. If you specify the `COUNT` attribute, the `WITHOUT DEFAULTS` option is not required because it is implicit. If the `COUNT` attribute is greater than `MAXCOUNT`, the runtime system will take `MAXCOUNT` as the actual number of rows. If the value of `COUNT` is negative or zero, it defines an empty list.

MAXCOUNT option

The `MAXCOUNT` attribute defines the maximum number of rows that can be inserted in the program array. This attribute allows you to give an upper limit. When using a dynamic array, the user can enter an infinite number of rows unless the `MAXCOUNT` attribute is used. When using a static array, and `MAXCOUNT` is greater than the size of the declared static array, the original static array size is used as the upper limit. If `MAXCOUNT` is negative or equal to zero, user cannot insert rows.

HELP option

The HELP attribute defines the help number of the text to be displayed when invoked and focus is in the list controlled by the INPUT ARRAY sub-dialog.

KEEP CURRENT ROW option

The current row of a list is highlighted during the execution of the dialog, and cleared when the DIALOG instruction ends. You can change this default behavior by using the KEEP CURRENT ROW attribute, to force the runtime system to keep the current row highlighted.

WITHOUT DEFAULTS option

You typically use the INPUT ARRAY sub-dialog with the WITHOUT DEFAULTS attribute. If this attribute is not set when using an INPUT ARRAY sub-dialog, the list is empty even if the array holds data. For more details see WITHOUT DEFAULTS option.

CONSTRUCT ATTRIBUTES clause

NAME option

The NAME attribute can be used to identify the CONSTRUCT sub-dialog; this is especially useful to qualify sub-dialog actions.

HELP option

The HELP attribute defines the help number of the text to be displayed when invoked and focus is in one of the fields controlled by the CONSTRUCT sub-dialog.

Default Actions

According to the sub-dialogs defined in the DIALOG instruction, the runtime system creates a set of default actions. These actions are provided to ease the implementation of the controller. For example, when using an INPUT ARRAY sub-dialog, the dialog instruction will automatically create the insert, append and delete default actions.

The following table lists the default actions created for the DIALOG interactive instruction, according to the sub-dialogs defined:

Default action	Control Block execution order
help	Shows the help topic defined by the HELP clause. *Only created when a HELP clause or option is defined for the sub-dialog.*
insert	Inserts a new row before current row. *Only created if INPUT ARRAY is used; action creation can be avoided with INSERT ROW = FALSE attribute.*

`append`	Appends a new row at the end of the list. *Only created if* `INPUT ARRAY` is *used; action creation can be avoided with* `APPEND ROW = FALSE` *attribute.*
`delete`	Deletes the current row. *Only created if* `INPUT ARRAY` is *used;* action *creation can be avoided with* `DELETE ROW = FALSE` *attribute.*
`firstrow`	Moves to the first row in the list.
`lastrow`	Moves to the last row in the list.
`nextrow`	Moves to the next row in the list.
`prevrow`	Moves to the previous row in the list.

The `insert`, `append` and `delete` default actions can be avoided with dialog control attributes:

```
01  INPUT ARRAY arr TO sr.* ATTRIBUTE( INSERT ROW=FALSE, APPEND
ROW=FALSE, ... )
02        ...
```

Control Blocks

Control blocks are predefined triggers where you can implement specific code to control the interactive instruction, by using ui.Dialog class methods or dialog specific instructions such as NEXT FIELD or CONTINUE DIALOG.

- BEFORE DIALOG block
- AFTER DIALOG block
- BEFORE FIELD block
- AFTER FIELD block
- ON CHANGE block
- BEFORE INPUT block
- AFTER INPUT block
- BEFORE CONSTRUCT block
- AFTER CONSTRUCT block
- BEFORE DISPLAY block
- AFTER DISPLAY block
- BEFORE ROW block
- ON ROW CHANGE block
- AFTER ROW block
- BEFORE INSERT block
- AFTER INSERT block
- BEFORE DELETE block
- AFTER DELETE block

BEFORE DIALOG block

The BEFORE DIALOG block is executed one time as the first trigger when the DIALOG instruction is starts, before the runtime system gives control to the user. You can implement variable initialization and dialog configuration in this block.

In the following example, the BEFORE DIALOG block performs some dialog setup and gives the focus to a specific field:

```
01    BEFORE DIALOG
02      CALL DIALOG.setActionActive("save",FALSE)
03      CALL DIALOG.setFieldActive("cust_status", is_admin())
04      IF cust_is_new() THEN
05         NEXT FIELD cust_name
06      END IF
```

A DIALOG instruction can include no more than one BEFORE DIALOG control block.

AFTER DIALOG block

The AFTER DIALOG block is executed one time as the last trigger when the DIALOG instruction terminates, typically after the user has validated or canceled the dialog, and before the runtime system executes the instruction that appears just after the END DIALOG keywords. You typically implement dialog finalization in this block.

The dialog terminates when an ACCEPT DIALOG or EXIT DIALOG control instruction is executed. However, the AFTER DIALOG block is not executed if an EXIT DIALOG is performed.

If you execute one of the following control instructions in an AFTER DIALOG block, the dialog will not terminate and it will give control back to the user:

1. NEXT FIELD
2. NEXT OPTION
3. CONTINUE DIALOG

In the next example, the AFTER DIALOG block checks whether a field value is correct and gives control back to the dialog if the value is wrong:

```
01    AFTER DIALOG
02      IF NOT cust_is_valid_status(p_cust.cust_status) THEN
03         ERROR "Customer state is not valid"
04         NEXT FIELD cust_status
05      END IF
```

BEFORE FIELD block

In parts of a dialog driven by a simple INPUT or by a CONSTRUCT sub-dialog, the `BEFORE FIELD` block is executed every time the cursor enters into the specified field. For editable lists driven by INPUT ARRAY, this block is executed when moving the focus from field to field in the same row, or when moving to another row in the same column.

The `BEFORE FIELD` keywords must be followed by a list of form field specification. The screen-record name can be omitted.

`BEFORE FIELD` is executed after `BEFORE INPUT`, `BEFORE CONSTRUCT`, `BEFORE ROW` and `BEFORE INSERT`. For more details, see Control Block Execution Order.

You typically do some field value initialization or message display in a `BEFORE FIELD` block:

```
01    BEFORE FIELD cust_status
02      LET p_cust.cust_comment = NULL
03      MESSAGE "Enter customer status"
```

Note that the `BEFORE FIELD` block is also executed when NEXT FIELD is executed programmatically. Note that the trigger is fired even if the field is declared as NOENTRY or disabled with ui.Dialog.setFieldActive("*field-name*",FALSE). If you execute `NEXT FIELD CURRENT` or `NEXT FIELD` *`current-field`* and *current-field* is the current field, `BEFORE FIELD` *`current-field`* is also executed; `AFTER FIELD` *`current-field`* is not executed.

Warning: When using the default `FIELD ORDER CONSTRAINT` mode, the dialog executes the `BEFORE FIELD` block of the field corresponding to the first variable of an `INPUT` or `INPUT ARRAY`, even if that field is not editable (NOENTRY or disabled). The block is executed when you enter the dialog and every time you create a new row in the case of `INPUT ARRAY`. This behavior is supported for backward compatibility. The block is not executed when using the `FIELD ORDER FORM`, as recommended for `DIALOG` instructions.

When form-level validation occurs and a field contains an invalid value, the dialog gives the focus to the field, but no `BEFORE FIELD` trigger will be executed.

AFTER FIELD block

In dialog parts driven by a simple INPUT or by a CONSTRUCT sub-dialog, the `AFTER FIELD` block is executed every time the user moves to another form item. For editable lists driven by INPUT ARRAY, this block is executed when moving the focus from field to field in the same row, or when moving to another row in the same column.

The `AFTER FIELD` keywords must be followed by a a list of form field specifications. The screen-record name can be omitted.

AFTER FIELD is executed before AFTER INSERT, ON ROW CHANGE, AFTER ROW, AFTER INPUT or AFTER CONSTRUCT. For more details, see Control Block Execution Order.

When a NEXT FIELD instruction is executed in an AFTER FIELD block, the cursor moves to the specified field, which can be the current field. This can be used to prevent the user from moving to another field / row during data input.

Note that the AFTER FIELD block of the current field is not executed when performing a NEXT FIELD; only BEFORE INPUT, BEFORE CONSTRUCT, BEFORE ROW, and BEFORE FIELD of the target item might be executed, based on the sub-dialog type.

You typically code some validation rules in an AFTER FIELD block:

```
01    AFTER FIELD item_quantity
02      IF p_item.item_quantity <= 0 THEN
03         ERROR "Item quantity cannot be negative or zero"
04         LET p_item.item_quantity = 0
05         NEXT FIELD item_quantity
06      END IF
```

When ACCEPT DIALOG is used, the AFTER FIELD trigger of the current field will be executed.

ON CHANGE block

The ON CHANGE block can be used to detect that a field changed by user input. The ON CHANGE block is executed if the value has changed since the field got the focus and if the TOUCHED flag is set. The ON CHANGE block can only be used in INPUT and INPUT ARRAY sub-dialogs, it is not available in CONSTRUCT.

For fields defined as RadioGroup, ComboBox, SpinEdit, Slider, and CheckBox views, the ON CHANGE block is fired immediately when the user changes the value. For other type of fields (like Edits), the ON CHANGE block is fired when leaving the field. You leave the field when you validate the dialog, when you move to another field, or when you move to another row in an INPUT ARRAY. Note that the dialogtouched predefined action can also be used to detect field changes immediately, but with this action you can't get the data in the target variables (should only be used to detect that the user has started to modify data).

If both an ON CHANGE block and AFTER FIELD block are defined for a field, the ON CHANGE block is executed before the AFTER FIELD block. For more details, see Control Block Execution Order.

When changing the value of the current field programmatically in an ON ACTION block, the ON CHANGE *current-field* block will be executed when leaving the field if the value is different from the reference value and if the TOUCHED flag is set.

When using the NEXT FIELD instruction, the comparison value is re-assigned as if the user had left and re-entered the field. Therefore, when using `NEXT FIELD` in an `ON CHANGE` block or in an `ON ACTION` block, the `ON CHANGE` block will only be fired again if the value is different from the reference value. For this reason, it is not recommended that you attempt field validation in `ON CHANGE` blocks;: it is better to perform validations in `AFTER FIELD` blocks.

BEFORE INPUT block

The `BEFORE INPUT` block is executed when the focus goes to a group of fields driven by an INPUT or INPUT ARRAY sub-dialog. This trigger is only fired if a field of the sub-dialog gets the focus, and none of the other fields had the focus. When the focus is in a list driven by an `INPUT ARRAY` sub-dialog, moving to a different row will not fire the `BEFORE INPUT` block.

Note that in singular INPUT and INPUT ARRAY instructions, the `BEFORE INPUT` is only executed once when the dialog is started, while the `BEFORE INPUT` block of the `DIALOG` instruction is executed each time the group of fields gets the focus. Thus, it is not designed to be used as an initialization block, but rather as a trigger to detect focus changes and activate the possible actions for the current group.

`BEFORE INPUT` is executed after the `BEFORE DIALOG` block and before the `BEFORE ROW`, `BEFORE FIELD` blocks. For more details, see Control Block Execution Order.

When using an INPUT ARRAY sub-dialog, the ui.Dialog.getCurrentRow("*screen-array*") function returns the index of the current row when it is called in the `BEFORE INPUT` block.

In the following example, the `BEFORE INPUT` block is used to set up a specific action and display a message:

```
01    INPUT BY NAME p_order.*
02      BEFORE INPUT
03       CALL DIALOG.setActionActive("validate_order", TRUE)
04       MESSAGE "Enter order information"
```

AFTER INPUT block

The `AFTER INPUT` block is executed when the focus is lost by a group of fields driven by an INPUT or INPUT ARRAY sub-dialog. This trigger is fired if a field of the sub-dialog loses the focus, and a field of a different sub-dialog gets the focus. If the focus leaves the current group of fields and goes to an action view, `AFTER INPUT` is not executed, because the focus did not go to another sub-dialog yet. When the focus is in a list driven by an `INPUT ARRAY` sub-dialog, moving to a different row will not fire the `AFTER INPUT` block.

Note that in singular INPUT and INPUT ARRAY instructions, the AFTER INPUT is only executed once when dialog ends, while the AFTER INPUT block of the DIALOG instruction is executed each time the group of fields loses the focus. Thus, it is not designed to be used as an finalization block, but rather as a trigger to detect focus changes and implement validation rules.

AFTER INPUT is executed after the AFTER FIELD, AFTER ROW blocks and before the AFTER DIALOG block. For more details, see Control Block Execution Order.

When using an INPUT ARRAY sub-dialog, the ui.Dialog.getCurrentRow("*screen-array*") function returns the index of the current row when it is called in the AFTER INPUT block.

Executing a NEXT FIELD in the AFTER INPUT control block will keep the focus in the group of fields. Within an INPUT ARRAY sub-dialog, NEXT FIELD will keep the focus in the list and stay in the current row. You typically use this behavior to control user input.

In the following example, the AFTER INPUT block is used to validate data and disable an action that can only be used in the current group:

```
01    INPUT BY NAME p_order.*
02      AFTER INPUT
03       IF NOT check_order_data(DIALOG) THEN
04         NEXT FIELD CURRENT
05       END IF
06       CALL DIALOG.setFieldActive("validate_order", FALSE)
```

BEFORE CONSTRUCT block

The BEFORE CONSTRUCT block is executed when the focus goes to a group of fields driven by a CONSTRUCT sub-dialog. This trigger is only fired if a field of the sub-dialog gets the focus, and none of the other fields had the focus.

Note that in the singular CONSTRUCT instruction, the BEFORE CONSTRUCT is only executed once when dialog is started, while the BEFORE CONSTRUCT block of the DIALOG instruction is executed each time the group of fields gets the focus. Thus, it is not designed to be used as an initialization block, but rather as a trigger to detect focus changes and activate the possible actions for the current group.

BEFORE CONSTRUCT is executed after the BEFORE DIALOG block and before the BEFORE FIELD blocks. For more details, see Control Block Execution Order.

In the following example, the BEFORE CONSTRUCT block is used to display a message:

```
01    CONSTRUCT BY NAME sql ON customer.*
02      BEFORE CONSTRUCT
03       MESSAGE "Enter customer search filter"
```

AFTER CONSTRUCT block

The AFTER CONSTRUCT block is executed when the focus is lost by a group of fields driven by a CONSTRUCT sub-dialog. This trigger is fired if a field of the sub-dialog loses the focus, and a field of a different sub-dialog gets the focus. If the focus leaves the current group of fields and goes to an action view, AFTER CONSTRUCT is not executed, because the focus did not yet go the another sub-dialog.

Note that in the singular CONSTRUCT instruction, the AFTER CONSTRUCT is only executed once when the dialog ends, while the AFTER CONSTRUCT block of the DIALOG instruction is executed each time the group of fields loses the focus. Thus, it is not designed to be used as an finalization block, but rather as a trigger to detect focus changes and implement validation rules.

AFTER CONSTRUCT is executed after the AFTER FIELD and before the AFTER DIALOG block. For more details, see Control Block Execution Order.

Executing a NEXT FIELD in the AFTER CONSTRUCT control block will keep the focus in the group of fields.

In the following example, the AFTER CONSTRUCT block is used to build the SELECT statement:

```
01    CONSTRUCT BY NAME sql ON customer.*
02      AFTER CONSTRUCT
03       LET sql = "SELECT * FROM customers WHERE " || sql
```

BEFORE DISPLAY block

The BEFORE DISPLAY block is executed when a DISPLAY ARRAY list gets the focus.

Note that in the singular DISPLAY ARRAY instruction, BEFORE DISPLAY is only executed once when the dialog is started, while the BEFORE DISPLAY block of the DIALOG instruction is executed each time the list gets the focus. Thus, it is not designed to be used as an initialization block, but rather as a trigger to detect focus changes and activate the possible actions for the current list.

BEFORE DISPLAY is executed before the BEFORE ROW block. For more details, see Control Block Execution Order.

When called in this block, the ui.Dialog.getCurrentRow("*screen-array*") function returns the index of the current row.

In the following example the BEFORE DISPLAY block enables an action and displays a message:

```
01    DISPLAY ARRAY p_items TO s_items.*
```

```
02      BEFORE DISPLAY
03       CALL DIALOG.setActionActive("clear_item_list", TRUE)
04       MESSAGE "You are now in the list of items"
```

AFTER DISPLAY block

The AFTER DISPLAY block is executed when a DISPLAY ARRAY list loses the focus and goes to another sub-dialog. If the focus leaves the current list and goes to an action view, AFTER DISPLAY is not executed, because the focus did not go to another sub-dialog yet.

Note that in the singular DISPLAY ARRAY instruction, the AFTER DISPLAY is only executed once when the dialog ends, while the AFTER DISPLAY block of the DIALOG instruction is executed each time the list loses the focus. Thus, it is not designed to be used as an finalization block, but rather as a trigger to detect focus lost and disable actions specific to the current list.

AFTER DISPLAY is executed after the AFTER ROW block. For more details, see Control Block Execution Order.

When called in this block, the ui.Dialog.getCurrentRow("*screen-array*") function returns the index of the row that you are leaving.

In the following example, the AFTER DISPLAY block disables an action that is specific to the current list:

```
01    DISPLAY ARRAY p_items TO s_items.*
02      AFTER DISPLAY
03       CALL DIALOG.setActionActive("clear_item_list", FALSE)
```

BEFORE ROW block

The BEFORE ROW block is executed when a DISPLAY ARRAY or INPUT ARRAY list gets the focus, or when the user moves to another row inside a list. This trigger can also be executed in other situations, for example when you delete a row, or when the user tries to insert a row but the maximum number of rows in the list is reached (see Control Blocks Execution Order for more details).

You typically do some dialog setup / message display in the BEFORE ROW block, because it indicates that the user selected a new row. Do not use this trigger to detect focus changes; You better use the BEFORE DISPLAY or BEFORE INPUT blocks instead.

BEFORE ROW is executed before the BEFORE INSERT and BEFORE FIELD blocks and after the BEFORE DISPLAY or BEFORE INPUT blocks. For more details, see Control Block Execution Order.

Note that when the dialog starts, BEFORE ROW will only be executed if the list has received the focus. If you have other elements in the form which can get the focus before the list, BEFORE ROW will not be triggered when the dialog starts. You must pay attention to this, because this behavior is new compared to singular DISPLAY ARRAY or INPUT ARRAY. In singular dialogs, the BEFORE ROW block is always executed when the dialog starts, since only the list can get the focus.

When called in this block, the ui.Dialog.getCurrentRow("*screen-array*") function returns the index of the current row.

In the following example the BEFORE ROW block gets the new row number and displays it in a message:

```
01    DISPLAY ARRAY p_items TO s_items.*
02      BEFORE ROW
03        MESSAGE "We are in items, on row #",
DIALOG.getCurrentRow("s_items")
```

ON ROW CHANGE block

The ON ROW CHANGE block is executed in a list driven by an INPUT ARRAY when you leave the current row and the row has been modified since it got the focus. This is typically used to detect whether the user has changed a value in the current row.

The ON ROW CHANGE block is only executed if at least one field value in the current row has changed since the row was entered, and the TOUCHED flags of the modified fields are set. The modified field(s) might not be the current field, and several field values can be changed. Values might have been changed by the user or by the program. Note that the TOUCHED flag is reset for all fields when entering another row, when going to another sub-dialog, or when leaving the dialog instruction.

ON ROW CHANGE is executed after the AFTER FIELD block and before the AFTER ROW block. For more details, see Control Block Execution Order.

When called in this block, the ui.Dialog.getCurrentRow("*screen-array*") function returns the index of the current row.

You can, for example, code database modifications (UPDATE) in the ON ROW CHANGE block:

```
01    INPUT ARRAY p_items FROM s_items.*
02     ...
03     ON ROW CHANGE
04       LET r = DIALOG.getCurrentRow("s_items")
05       UPDATE items SET
06              items.item_code        = p_items[r].item_code,
07              items.item_description = p_items[r].item_description,
08              items.item_price       = p_items[r].item_price,
```

```
09                items.item_updatedate  = TODAY
10              WHERE items.item_num = p_items[r].item_num
```

AFTER ROW block

The AFTER ROW block is executed when a DISPLAY ARRAY or INPUT ARRAY list loses the focus, or when the user moves to another row in a list. This trigger can also be executed in other situations, for example when you delete a row, or when the user inserts a new row (see Control Blocks Execution Order for more details).

AFTER ROW is executed after the AFTER FIELD, AFTER INSERT and before AFTER DISPLAY or AFTER INPUT blocks. For more details, see Control Block Execution Order.

When called in this block, the ui.Dialog.getCurrentRow("*screen-array*") function returns the index of the row that you are leaving.

Warning: When leaving a temporary row that will be removed because the user goes to a previous row in the list, AFTER ROW is executed for the temporary row, but ui.Dialog.getCurrentRow() / ARR_CURR() will be one row greater than ui.Dialog.getArrayLength() / ARR_COUNT(). You should not access a dynamic array with a row index that is greater than the total number of rows, otherwise the runtime system will adapt the total number of rows to the actual number of rows in the program array.

For both INPUT ARRAY and DISPLAY ARRAY sub-dialogs, a NEXT FIELD executed in the AFTER ROW control block will keep the focus in the list and stay in the current row. Thus you can use this to implement row input validation and prevent the user from leaving the list or moving to another row.

In the following example, the AFTER ROW block checks a variable value and forces the user to stay in the current row if the value is wrong:

```
01    INPUT ARRAY p_items FROM s_items.*
02     ...
03     AFTER ROW
04       LET r = DIALOG.getCurrentRow("s_items")
05       IF r <= DIALOG.getArrayLength("s_items") THEN
06         IF NOT item_is_valid_quantity(p_item[r].item_quantity) THEN
07            ERROR "Item quentity is not valid"
08            NEXT FIELD item_quantity
09          END IF
10       END IF
```

BEFORE INSERT block

The BEFORE INSERT block is executed when a new row ins inserted or when a temporary row is appended in an INPUT ARRAY. You typically use this trigger to set some default values in the new created row.

The BEFORE INSERT block is executed after the BEFORE ROW block and before the BEFORE FIELD block. For more details, see Control Block Execution Order.

When called in this block, the ui.Dialog.getCurrentRow("*screen-array*") function returns the index of the new created row.

To distinguish row insertion from an appended row, compare the current row (DIALOG.getCurrentRow("*screen-array*")) with the total number of rows (DIALOG.getArrayLength("*screen-array*")). If these correspond, you are in a temporary row.

You can cancel row creation by using the CANCEL APPEND instruction inside BEFORE INSERT.

In the following example, the BEFORE INSERT block checks if the user can create rows and denies new row creation if needed; otherwise, it sets some default values:

```
01    INPUT ARRAY p_items FROM s_items.*
02     ...
03     BEFORE INSERT
04       IF NOT user_can_append THEN
05          ERROR "You are not allowed to append rows"
06          CANCEL INSERT
07       END IF
08       LET r = DIALOG.getCurrentRow("s_items")
09       LET p_items[r].item_num = get_new_serial("items")
10       LET p_items[r].item_name = "undefined"
```

AFTER INSERT block

The AFTER INSERT block is executed when a new created row of an INPUT ARRAY list is validated. In this block, you typically implement SQL to insert a new row in the database table.

The AFTER INSERT block is executed after the AFTER FIELD block and before the AFTER ROW block. For more details, see Control Block Execution Order.

When called in this block, the ui.Dialog.getCurrentRow("*screen-array*") function returns the index of the last row.

Warning: When the the user appends a new row at the end of the list, then moves UP to another row or validates the dialog, the AFTER INSERT block is only

executed if at least one field was edited. If no data entry is detected, the dialog automatically removes the new appended row and thus does not trigger the `AFTER INSERT` block.

When executing a NEXT FIELD in the `AFTER INSERT` block, the dialog will keep the focus in the list and stay in the current row. You can use this to implement row input validation and prevent the user from leaving the list or moving to another row. However, this will not cancel the row insertion and will not fire the `BEFORE INSERT` / `AFTER INSERT` triggers again. The only way to keep the focus in the current row after this is to execute a `NEXT FIELD` in the AFTER ROW block.

In the following example, the `AFTER INSERT` block inserts a new row in the database and cancels the operation if the SQL command fails:

```
01    INPUT ARRAY p_items FROM s_items.*
02     ...
03     AFTER INSERT
04       WHENEVER ERROR CONTINUE
05       INSERT INTO items VALUES (
p_items[DIALOG.getCurrentRow("s_items")].* )
06       WHENEVER ERROR STOP
07       IF SQLCA.SQLCODE<>0 THEN
08          ERROR SQLERRMESSAGE
09          CANCEL INSERT
10       END IF
```

BEFORE DELETE block

The `BEFORE DELETE` block is executed each time the user deletes a row of an INPUT ARRAY list, before the row is removed from the list.

You typically code the database table synchronization in the `BEFORE DELETE` block, by executing a DELETE SQL statement using the primary key of the current row. In the `BEFORE DELETE` block, the row to be deleted still exists in the program array, so you can access its data to identify what record needs to be removed.

The `BEFORE DELETE` block is executed before the `AFTER DELETE` block. For more details, see Control Block Execution Order.

If needed, the deletion can be canceled with the CANCEL DELETE instruction.

When called in this block, ui.Dialog.getCurrentRow("*screen-array*") returns the index of the row that will be deleted.

The next example uses the `BEFORE DELETE` block to remove the row from the database table and cancels the deletion operation if an SQL error occurs:

```
01   INPUT ARRAY p_items FROM s_items.*
```

```
02    BEFORE DELETE
03       LET r = DIALOG.getCurrentRow("s_items")
04       WHENEVER ERROR CONTINUE
05       DELETE FROM items WHERE item_num = p_items[r].item_num
06       WHENEVER ERROR STOP
07       IF SQLCA.SQLCODE<>0 VALUES
08          ERROR SQLERRMESSAGE
09          CANCEL DELETE
10       END IF
11 ...
```

AFTER DELETE block

The AFTER DELETE block is executed each time the user deletes a row of an INPUT ARRAY list, after the row has been deleted from the list.

The AFTER DELETE block is executed after the BEFORE DELETE block and before the AFTER ROW block for the deleted row and the BEFORE ROW block of the new current row. For more details, see Control Block Execution Order.

When an AFTER DELETE block executes, the program array has already been modified; the deleted row no longer exists in the array. Note that the ARR_CURR() function or the ui.Dialog.getCurrentRow("*screen-array*") method returns the same index as in BEFORE ROW, but it is the index of the new current row. Note that the AFTER ROW block is also executed. Pay particular attention when deleting the last row in the list; in this case, the current row index returned by ARR_CURR() is one higher than the actual number of rows in the list (ARR_COUNT()).

Warning: When deleting the last row of the list, AFTER DELETE is executed for the delete row, and ui.Dialog.getCurrentRow() / ARR_CURR() will be one row greater than ui.Dialog.getArrayLength() / ARR_COUNT(). You should not access a dynamic array with a row index that is greater than the total number of rows; otherwise, the runtime system will adapt the total number of rows to the actual number of rows in the program array.

Here the AFTER DELETE block is used to re-number the rows with a new item line number (note that ui.Dialog.getArrayLength() may return zero) :

```
01  INPUT ARRAY p_items FROM s_items.*
02    AFTER DELETE
03       LET r = DIALOG.getCurrentRow("s_items")
04       FOR i=r TO DIALOG.getArrayLength("s_items")
05           LET p_items[i].item_lineno = i
06       END FOR
07 ...
```

It is not possible to use the CANCEL DELETE instruction in anAFTER DELETE block. At this time it is too late to cancel row deletion, as the data row no longer exists in the program array.

Control Block Execution Order

The following table shows the order in which the runtime system executes the control blocks in the `DIALOG` instruction, according to the context and user action:

Context / User action	Control Block execution order
Entering the dialog	1. `BEFORE DIALOG` 2. `BEFORE INPUT`, `BEFORE CONSTRUCT` or `BEFORE DISPLAY` (first sub-dialog getting focus) 3. `BEFORE ROW` (if focus goes to a list) 4. `BEFORE FIELD` (if focus goes to a field)
When the focus goes to an `INPUT` or to a `CONSTRUCT` from a different sub-dialog	1. *Triggers raised by the context of the sub-dialog you leave* 2. `BEFORE INPUT` or `BEFORE CONSTRUCT` (new sub-dialog getting focus) 3. `BEFORE FIELD`
When the focus goes to an `INPUT` or to a `CONSTRUCT` from a different sub-dialog	1. `ON CHANGE` (if INPUT and value of current field has changed) 2. `AFTER FIELD` (for the current field) 3. `AFTER INPUT` or `AFTER CONSTRUCT` (current sub-dialog losing focus) 4. *Triggers raised by the context of the sub-dialog you enter*
When the focus goes to a `DISPLAY ARRAY` list or to an `INPUT ARRAY` list from a different sub-dialog	1. *Triggers raised by the context of the sub-dialog you leave* 2. `BEFORE INPUT` or `BEFORE DISPLAY` (new sub-dialog getting focus) 3. `BEFORE ROW` (the row that was selected in the list) 4. `BEFORE FIELD` (if it's an INPUT ARRAY)
When the focus leaves a `DISPLAY ARRAY` or `INPUT ARRAY` list to a different sub-dialog	1. `ON CHANGE` (if INPUT ARRAY and value of current field has changed) 2. `AFTER FIELD` (if it's an INPUT ARRAY) 3. `AFTER INSERT` (if current row was just created) 4. `AFTER ROW` (the current row in the list you leave) 5. `AFTER INPUT` or `AFTER DISPLAY` (current sub-dialog losing focus) 6. *Triggers raised by the context of the sub-*

	dialog you enter
Moving from field A to field B in an `INPUT` or `CONSTRUCT` sub-dialog or in the same row of an `INPUT ARRAY` list	1. `ON CHANGE` (if value of current field has changed) 2. `AFTER FIELD A` 3. `BEFORE FIELD B`
Moving from field A of an `INPUT` or `CONSTRUCT` sub-dialog to field B in another `INPUT` or `CONSTRUCT` sub-dialog	1. `ON CHANGE` (if value of current field has changed) 2. `AFTER FIELD A` 3. `AFTER INPUT` or `AFTER CONSTRUCT` (for sub-dialog of field A) 4. `BEFORE INPUT` or `BEFORE CONSTRUCT` (for sub-dialog of field B) 5. `BEFORE FIELD B`
Moving to a different row in a `DISPLAY ARRAY` list	1. `AFTER ROW` (the row you leave) 2. `BEFORE ROW` (the new current row)
Moving to a different row in an `INPUT ARRAY` list when current row was newly created	1. `ON CHANGE` (if value of current field has changed) 2. `AFTER FIELD` (for field A in the row you leave) 3. `AFTER INSERT` (the newly created row) 4. `AFTER ROW` (the newly created row) 5. `BEFORE ROW` (the new current row) 6. `BEFORE FIELD` (field in the new current row)
Moving to a different row in an `INPUT ARRAY` list when current row was modified	1. `ON CHANGE` (if value of current field has changed) 2. `AFTER FIELD` (for field A in the row you leave) 3. `ON ROW CHANGE` (the values in current row have changed) 4. `AFTER ROW` (for the row that was modified) 5. `BEFORE ROW` (the new current row) 6. `BEFORE FIELD` (field in the new current row)
Inserting or appending a new row in an `INPUT ARRAY` list	1. *Triggers raised by the context of the sub-dialog you leave* 2. `BEFORE INSERT` (for the new current row) 3. `BEFORE ROW` (the new current row) 4. `BEFORE FIELD` (field in the new current row)
Deleting a row in an `INPUT ARRAY` list	1. `BEFORE DELETE` (for the current row to be deleted) 2. `AFTER DELETE` (now the deleted row is removed)

	3. `AFTER ROW` (for the deleted row) 4. `BEFORE ROW` (the new current row)
Validating the dialog with `ACCEPT DIALOG`	1. `ON CHANGE` (if focus is in input field and value has changed) 2. `AFTER FIELD` (if focus is in input field) 3. `ON ROW CHANGE` (if focus is in a list and if values have changed in the current row) 4. `AFTER ROW` (if focus is in a list) 5. `AFTER INPUT`, `AFTER CONSTRUCT` or `AFTER CONSTRUCT` (current sub-dialog) 6. `AFTER DIALOG`
Canceling the dialog with `EXIT DIALOG`	None of the control blocks will be executed; we just leave the dialog instruction.

Interaction Blocks

Interaction blocks are fired by user actions. The `DIALOG` block supports the following interaction blocks:

- ON IDLE block
- ON ACTION block
- ON KEY block
- COMMAND [KEY] block

ON IDLE block

The `ON IDLE` *`idle-seconds`* clause defines a set of instructions that must be executed after *idle-seconds* of inactivity. The parameter *idle-seconds* must be an integer literal or variable. If it evaluates to zero, the timeout is disabled. The timeout value is taken into account when the dialog initializes its internal data structures. If you use a program variable instead of an integer constant, any change of the variable will have no effect if the change is done after the dialog has initialized. The `BEFORE DIALOG` block is not part of the internal dialog initialization. Thus, if you what to change the value of the timeout variable, it must be done before the `DIALOG` block.

This trigger can, for example, be used to quit the dialog after the user has not interacted with the program for a specified period of time:

```
01 FUNCTION my_dialog()
02    DEFINE timeout SMALLINT
03    LET timeout = 10
04    DIALOG
```

```
05      ...
06      ON IDLE timeout
07         IF ask_question("No activity detected after 10 seconds, do
you want to leave the dialog?") THEN
08            EXIT DIALOG
09         END IF
10 ...
```

ON ACTION block

The ON ACTION blocks execute a sequence of instructions when the user triggers a specific action. This is the preferred solution compared to ON KEY blocks, because ON ACTION blocks use abstract names to control user interaction.

Action blocks will be bound by name to action views in the current form. Typical actions views are form buttons, toolbar buttons, topmenu options. When an action block is defined in a DIALOG instruction, all corresponding action views will be automatically enabled on the front-end side. When the user clicks on one of the action views or presses one of the accelerator keys defined for the action, the corresponding action block is executed on the application server side.

The name of the action must follow the ON ACTION keywords. Note that the fglcomp and fglform compilers convert action names to lowercase, but some resource files such as **.4ad** Action Defaults files are XML files where action names are case-sensitive. To avoid any confusion, always write your action names in lowercase.

The next example defines an action block to open a typical zoom window and let the user select a customer record:

```
01    ON ACTION zoom
02       CALL zoom_customers() RETURNING st, cust_id, cust_name
```

The DIALOG instruction supports *dialog actions* and *sub-dialog actions*. The *dialog actions* are global to the dialog and can be fired wherever the focus is. You defined a *dialog action* by writing the ON ACTION block outside any sub-dialog. The *sub-dialog actions* are specific to sub-dialogs. You create a *sub-dialog action* when the ON ACTION block is defined inside a sub-dialog. The *sub-dialog* actions get an implicit prefix identifying the sub-dialog:

```
01 DIALOG
02    INPUT BY NAME ... ATTRIBUTES (NAME = "cust")
03       ON ACTION suspend -- this is the local sub-dialog action
"cust.suspend"
04       ...
05    END INPUT
06    BEFORE DIALOG
07       ...
08    ON ACTION close -- this is the dialog action "close"
09       ...
```

```
10 END DIALOG
```

See also the "Binding Action Views to Actions Handlers in DIALOG" section for more details about binding action views to action handlers.

Actions can be individually enabled or disabled with a dialog class method called setActionActive():

```
01    BEFORE ROW s_items
02       CALL DIALOG.setActionActive("show_item_details", TRUE)
```

In `BUFFERED` mode, when the user triggers an action, `DIALOG` suspends input to the current field and preserves the input buffer that contains the characters typed by the user before the `ON ACTION` block is executed. After the block is executed, `DIALOG` restores the input buffer to the current field and resumes input on the same field, unless a control instruction such as `NEXT FIELD` or `EXIT DIALOG` was issued in the block.

In `UNBUFFERED` mode, before an `ON ACTION` block is executed, the value of the current field is validated and copied to the program variable. You can prevent field validation by using the validate action default attribute.

Some action names such as *close* and *interrupt* are predefined for a specific purpose. The *close* action will be automatically bound to the top-right button of the window, to trigger specific code when the user wants to close the window. The *interrupt* action is dedicated to sending an interruption event when the program is processing in a batch loop. The *interrupt* action does not require an `ON ACTION` block: Any action view with this name will automatically be enabled when the program is in processing mode. For more details about these specific actions, see Predefined Actions, Interruption Handling and Implementing the close action.

Warning: When a `ON ACTION` uses the same name as an implicit action such as insert, append or delete, either the implicit action or the user action will be executed, according to the context: The implicit action is always executed if the focus is in the sub-dialog of fields the action is designed for. Otherwise, the user code will be executed instead. For example, `ON ACTION` delete user action will only be executed if there is no `INPUT ARRAY` having the focus. Note that you can avoid list implicit actions with the `INSERT ROW` / `APPEND ROW` / `DELETE ROW` sub-dialog attributes.

ON KEY block

You can use `ON KEY` blocks to execute a sequence of instructions when the user presses a specific key. This clause allows you to define accelerator keys triggering actions. It is supported to simplify legacy code migration and is especially useful when writing TUI programs.

You must specify the key name between braces:

```
01    ON KEY (F5)
```

When you declare an `ON KEY` block, you actually define an `ON ACTION` block with an implicit accelerator key. The name of the action will be the key name in lowercase letters. Note that the Default Action View will be hidden (i.e. no automatic button will appear for this action on the front-end).

For backward compatibility, the `ON KEY` syntax allows multiple key names. If you specify multiple keys in an `ON KEY` clause, the `DIALOG` instruction will create an `ON ACTION` equivalent for each key:

```
01    ON KEY (F5, CONTROL-P, CONTROL-Z)
```

Concerning `BUFFERED/UNBUFFERED` modes driving the input buffer and variable synchronization, the same rules apply for `ON KEY` and `ON ACTION`. See ON ACTION block for more details.

The table below shows the key names that are accepted by the compiler:

Key Name	Description
`ACCEPT`	The validation key.
`INTERRUPT`	The interruption key.
`ESC` or `ESCAPE`	The ESC key (not recommended, use `ACCEPT` instead).
`TAB`	The TAB key (not recommended).
`Control-`*`char`*	A control key where *char* can be any character except A, D, H, I, J, K, L, M, R, or X.
`F1` through `F255`	A function key.
`DELETE`	The key used to delete a new row in an array.
`INSERT`	The key used to insert a new row in an array.
`HELP`	The help key.
`LEFT`	The left arrow key.
`RIGHT`	The right arrow key.
`DOWN`	The down arrow key.
`UP`	The up arrow key.
`PREVIOUS` or `PREVPAGE`	The previous page key.
`NEXT` or `NEXTPAGE`	The next page key.

COMMAND [KEY] block

You can use `COMMAND [KEY]` blocks to execute a sequence of instructions when the user clicks on a button or presses a specific key. This clause allows you to define *text* and *comment* action view decoration attributes as well as accelerator keys for a specific

action. It is supported to simplify legacy code migration and is especially useful when writing TUI programs.

When you declare a `COMMAND [KEY]` block, you actually define an `ON ACTION` block with an implicit *text* and *comment* decoration attribute. If you specify the optional `KEY` clause, you also define an implicit accelerator key. The name of the action will be the option text in lowercase letters. Unlike `ON KEY` actions, the Default Action View will be visible (i.e. the automatic button will appear for this action on the front-end).

```
01    COMMAND "Open" "Opens a new file"
```

When using the optional `KEY` clause, you must specify the key name between parentheses:

```
01    COMMAND KEY (F5) "Open" "Opens a new file"
```

For backward compatibility, the `COMMAND KEY` syntax allows multiple key names. If you specify multiple keys in an `COMMAND KEY` clause, the `DIALOG` instruction will create only one `ON ACTION` equivalent, and use the specified keys as accelerators:

```
01    COMMAND KEY (F5, CONTROL-P, CONTROL-Z) "Open" "Opens a new file"
```

The `COMMAND [KEY]` block specification can define a help number, to display the corresponding text of the current help file.

```
01    COMMAND "Open" "Opens a new file" HELP 34
```

Control Instructions

The control instructions are used to control the behavior of the interactive instruction. `DIALOG` supports the following control instructions:

- CLEAR instruction
- DISPLAY TO / BY NAME instruction
- NEXT FIELD instruction
- CONTINUE DIALOG instruction
- EXIT DIALOG instruction
- ACCEPT DIALOG instruction
- CANCEL DELETE instruction
- CANCEL INSERT instruction

NEXT FIELD instruction

The `NEXT FIELD` *`field-name`* instruction gives the focus to the specified field (or read-only list when using `DISPLAY ARRAY`).

You typically use the NEXT FIELD instruction to control field input dynamically, in BEFORE FIELD, ON CHANGE or AFTER FIELD blocks, or to control row validation in AFTER ROW. You can also give the focus to a specific field with the NEXT FIELD instruction. When using a read-only list driven by a DISPLAY ARRAY binding, it is possible to give the focus to the list by using NEXT FIELD; just specify the first field used by the DISPLAY ARRAY controller.

If the target field specified in the NEXT FIELD instruction is inside the current sub-dialog, neither AFTER FIELD nor AFTER ROW will be fired for the field or list you are leaving. However, the BEFORE FIELD control blocks of the destination field / list will be executed.

If the target field specified in the NEXT FIELD instruction is outside the current sub-dialog, the AFTER FIELD, AFTER INSERT, AFTER INPUT and AFTER ROW control blocks will be fired for the field or list you are leaving. Form-level validation rules will also be checked, as if the user had selected the new sub-dialog himself. This is required to guarantee that the current sub-dialog is left in a consistent state. Of course, the BEFORE INPUT/CONSTRUCT, BEFORE ROW and the BEFORE FIELD control blocks of the destination field / list will be executed after that.

Abstract field identification is supported with the CURRENT, NEXT and PREVIOUS keywords. These keywords represent the current, next and previous fields respectively, corresponding to variables as defined in the input binding list (with the FROM or BY NAME clause). The NEXT and PREVIOUS options follow the tabbing order defined by the form when using FIELD ORDER FORM. Otherwise, they follow the order defined by the binding list.

Non-editable fields are fields defined with the NOENTRY attribute, fields disabled with ui.Dialog.setFieldActive("*field-name*", *value*), or fields using a widget that does not allow input, such as a LABEL. If a NEXT FIELD instruction selects a non-editable field, the next editable field gets the focus (defined by the FIELD ORDER FORM mode used by the dialog). However, the BEFORE FIELD and AFTER FIELD blocks of non-editable fields are executed when a NEXT FIELD instruction selects such a field.

When using NEXT FIELD in AFTER ROW or in ON ROW CHANGE, the dialog will stay in the current row and give control back to the user. This behavior allows you to implement data input rules:

```
01    AFTER ROW
02       IF NOT int_flag AND
03        arr[arr_curr()].it_count * arr[arr_curr()].it_value > maxval
THEN
04          ERROR "Amount of line exceeds max value."
05          NEXT FIELD item_count
06       END IF
```

Note that you can also use the ui.Dialog.nextField("*field-name*") method to register a field when the name is not known at compile time. However, this method does only register the field: It does not stop code execution as the NEXT FIELD instruction. You must issue a CONTINUE DIALOG to get the same behavior.

CLEAR instruction

The `CLEAR field-list` instruction clears the value buffer of specified form fields. The CLEAR instruction changes the buffers directly in the current form, not the program variables bound to the dialog. It can be used outside any dialog instruction, such as the DISPLAY instruction.

As `DIALOG` is typically used with the UNBUFFERED mode, there is no reason to clear field buffers in a `DIALOG` block since any variable assignment will synchronize field buffers. Actually, changing the field buffers with DISPLAY or CLEAR instruction will have no visual effect if the fields are used by a dialog working in `UNBUFFERED` mode, because the variables bound to the dialog will be used to reset the field buffer just before giving control back to the user. So if you want to clear fields, just set the variables to `NULL` and the fields will be cleared. However, when using a CONSTRUCT binding, you may want to clear fields with this `CLEAR` instruction, as there are no program variables bound to fields (with `CONSTRUCT`, only one string variable is bound to hold the SQL condition).

You can specify a single field, a screen record or screen array (with or without a screen-line specification) as described in the following table:

CLEAR instruction	Result
`CLEAR field-name`	Clears the specified field in the current line of the screen array.
`CLEAR screen-array.*`	Clears all fields in the current line of the screen array.
`CLEAR screen-array[n].*`	Clears all fields in line *n* of the screen array.
`CLEAR screen-array[n].field-name`	Clears the specified field in line *n* of the screen array.

Note that a screen array with a screen-line specification doesn't make much sense in a GUI application using TABLE containers.

DISPLAY TO / BY NAME instruction

The `DISPLAY variable-list TO field-list` or `DISPLAY BY NAME variable-list` instruction fills the value buffers of specified form fields with the values contained in the specified program variables. The DISPLAY instruction changes the buffers directly in the current form, not the program variables bound to the dialog. `DISPLAY` can be used outside any dialog instruction, in the same way as the CLEAR instruction. `DISPLAY` also sets the TOUCHED flag.

As `DIALOG` is typically used with the UNBUFFERED mode, there is no reason to set field buffers in a `DIALOG` block since any variable assignment will synchronize field buffers.

Actually, changing the field buffers with the DISPLAY or CLEAR instruction will have no visual effect if the fields are used by a dialog working in UNBUFFERED mode, because the variables bound to the dialog will be used to reset the field buffer just before giving control back to the user. So if you want to set field values, just assign the variables and the fields will be synchronized. However, when using a CONSTRUCT binding, you may want to set field buffers with this DISPLAY instruction, as there are no program variables bound to fields (with CONSTRUCT, only one string variable is bound to hold the SQL condition).

Note that if you are used to performing a DISPLAY to set the TOUCHED flag of fields to simulate user input, you can now use the ui.Dialog.setFieldTouched("*field-name*", *value*) method instead.

CONTINUE DIALOG instruction

The CONTINUE DIALOG statement continues the execution of the DIALOG instruction, skipping all statements appearing after this instruction. Control returns to the dialog instruction, which executes remaining control blocks as if the program reached the end of the current control block. Then the control goes back to the user and the dialog waits for a new event.

The CONTINUE DIALOG statement is useful when program control is nested within multiple conditional statements, and you want to return control to the user by skipping the rest of the statements.

In the following code example, an ON ACTION block gives control back to the dialog, skipping all instructions below line 04:

```
01    ON ACTION zoom
02       IF p_cust.cust_id IS NULL OR p_cust.cust_name IS NULL THEN
03         ERROR "Zoom window cannot be opened if there is no info to
identify the customer"
04         CONTINUE DIALOG
05       END IF
06       IF p_cust.cust_address IS NULL THEN
07       ...
```

If this instruction is called in a control block that is not AFTER DIALOG, further control blocks might be executed according to the context. Actually, CONTINUE DIALOG just instructs the dialog to continue as if the code in the control block was terminated (i.e. it's a kind of GOTO end_of_control_block). However, when executed in AFTER DIALOG, the focus returns to the current field or read-only list. In this case the BEFORE ROW and BEFORE FIELD triggers will be fired.

A CONTINUE DIALOG in AFTER INPUT, AFTER DISPLAY or AFTER CONSTRUCT will only stop the program flow (instructions after CONTINUE DIALOG will not be executed). If the user has selected a field in a different sub-dialog, this new field will get the focus and all necessary AFTER / BEFORE control blocks will be executed.

EXIT DIALOG instruction

The `EXIT DIALOG` statement just terminates the `DIALOG` block without any further control block execution. Program flow resumes at the instruction following the `END DIALOG` keywords.

```
01    ON ACTION quit
02       EXIT DIALOG
```

When leaving the `DIALOG` instruction, all form items used by the dialog will be disabled until another interactive statement takes control.

ACCEPT DIALOG instruction

The `ACCEPT DIALOG` statement validates all input fields bound to the `DIALOG` instruction and leaves the block if no error is raised. Control blocks such as `ON CHANGE`, `AFTER FIELD`, `AFTER ROW`, `AFTER INPUT/DISPLAY/CONSTRUCT` will be executed according to the dialog structure. The statements appearing after the `ACCEPT DIALOG` will be skipped.

You typically code an `ACCEPT DIALOG` in an `ON ACTION accept` block:

```
01    ON ACTION accept
02       ACCEPT DIALOG
```

Input field validation is a process that does several successive validation tasks, as listed below:

1. The current field value is checked, according to the program variable data type (for example, the user must input a valid date in a DATE field).
2. NOT NULL field attributes are checked for all input fields. This attribute forces the field to have a value set by program or entered by the user. If the field contains no value, the constraint is not satisfied. Note that input values are right-trimmed, so if the user inputs only spaces, this corresponds to a NULL value which does not fulfill the NOT NULL constraint.
3. REQUIRED field attributes are checked for all input fields. This attribute forces the field to have a default value, or to be TOUCHED by the user or programmatically. If the field was not edited during the dialog, the constraint is not satisfied.
4. INCLUDE field attributes are checked for all input fields. This attribute forces the field to contain a value that is listed in the include list. If the field contains a value that is not in the list, the constraint is not satisfied.

If a field does not satisfy one of the above constraints, dialog termination is canceled, an error message is displayed, and the focus goes to the first field causing a problem.

After input field validation has succeeded, different types of control blocks will be executed based on the context and type of dialog bindings used. For more details, see Control Block Execution Order.

You may want to validate some parts of the dialog without leaving the block. To do so, you can use the the ui.Dialog.validate() method.

CANCEL DELETE instruction

In a list driven by an INPUT ARRAY sub-dialog, row deletion can be canceled by using the `CANCEL DELETE` instruction in the `BEFORE DELETE` block. Using this instruction in a different place will generate a compilation error.

When the `CANCEL DELETE` instruction is executed, the current `BEFORE DELETE` block is terminated without any other trigger execution (no `BEFORE ROW` or `BEFORE FIELD` is executed), and the program execution continues in the user event loop.

You can, for example, prevent row deletion based on some condition:

```
01    BEFORE DELETE
02       IF user_can_delete() == FALSE THEN
03          ERROR "You are not allowed to delete rows"
04          CANCEL DELETE
05       END IF
```

The instructions that appear after `CANCEL DELETE` will be skipped.

Note that you can also disable the *delete* action to prevent the user from performing a delete row action with:

```
01    CALL DIALOG.setActionActive("delete", FALSE)
```

On the other hand, you can prevent the user from deleting rows by using the `DELETE ROW = FALSE` option in the ATTRIBUTE clause.

See also BEFORE DELETE and AFTER DELETE control blocks.

CANCEL INSERT instruction

In a list driven by an INPUT ARRAY sub-dialog, row creation can be canceled by the program with the `CANCEL INSERT` instruction. This instruction can only be used in the BEFORE INSERT and AFTER INSERT control blocks. If it appears at a different place, the compiler will generate an error.

The instructions that appear after `CANCEL INSERT` will be skipped.

Note that you can also disable the *insert* and/or *append* actions to prevent the user from creating new rows with actions:

```
01    CALL DIALOG.setActionActive("insert", FALSE)
02    CALL DIALOG.setActionActive("append", FALSE)
```

However, this will not prevent the user from appending a new temporary row at the end of the list with a mouse click or the Down key. If you want to prevent row creation completely, you can use the `INSERT ROW = FALSE` and `APPEND ROW = FALSE` options in the ATTRIBUTE clause.

CANCEL INSERT in BEFORE INSERT

A `CANCEL INSERT` executed inside a `BEFORE INSERT` block prevents the new row creation. The following tasks are performed:

1. No new row will be created (the new row is not yet shown to the user).
2. The `BEFORE INSERT` block is terminated (further instructions are skipped).
3. The `BEFORE ROW` and `BEFORE FIELD` triggers are executed.
4. Control goes back to the user.

You can, for example, cancel a row creation if the user is not allowed to create rows:

```
01    BEFORE INSERT
02       IF NOT user_can_insert THEN
03          ERROR "You are not allowed to insert rows"
04          CANCEL INSERT
05       END IF
```

Warning: Executing `CANCEL INSERT` in `BEFORE INSERT` will also cancel a temporary row creation, except when there are no more rows in the list. In this case, `CANCEL INSERT` will just be ignored and leave the new row as is (otherwise, the instruction would loop without end). Note that you can prevent automatic temporary row creation with the AUTO APPEND attribute. If `AUTO APPEND=FALSE` and a `CANCEL INSERT` is executed in `BEFORE INSERT` (user has fired an explicit *append* action), the temporary row will be deleted and list will remain empty if it was the last row.

CANCEL INSERT in AFTER INSERT

A `CANCEL INSERT` executed inside an `AFTER INSERT` block removes the newly created row. The following tasks are performed:

1. The newly created row is removed from the list (the row exists now and user has entered data).
2. The `AFTER INSERT` block is terminated (further instructions are skipped).
3. The `BEFORE ROW` and `BEFORE FIELD` triggers are executed.
4. The control goes back to the user.

You can, for example, cancel a row insertion if a database error occurs when you try to insert the row into a database table:

```
01    AFTER INSERT
02       WHENEVER ERROR CONTINUE
03       LET r = DIALOG.getCurrentRow("s_items")
04       INSERT INTO items VALUES ( p_items[r].* )
05       WHENEVER ERROR STOP
06       IF SQLCA.SQLCODE<>0 THEN
07          ERROR SQLERRMESSAGE
08          CANCEL INSERT
09       END IF
```

Control Class

- Purpose of the ui.Dialog class
- The predefined DIALOG object reference
- Using DIALOG class methods

Purpose of the ui.Dialog class

When using a `DIALOG` instruction, you typically use the control instructions such as `EXIT DIALOG` to drive the dialog behavior. But some operations need parameters to be passed or values to be returned from the dialog. In this case, you must use the ui.Dialog built-class

The predefined DIALOG object reference

Inside the dialog instruction, the predefined keyword `DIALOG` represents the current dialog object. It can be used to execute methods provided in the dialog built-in class.

This keyword can be used as if it was defined as a ui.Dialog variable, but it cannot appear outside of the `DIALOG` block. If this is the case, you will get a compilation error. Note that the DIALOG object reference can be passed to functions, as in the following example:

```
01    BEFORE DIALOG
02       CALL setup_dialog(DIALOG)
```

Using DIALOG class methods

The DIALOG class can be used to manipulate dialog properties and control the behavior:

```
01    BEFORE DIALOG
02       CALL DIALOG.setActionActive("zoom",FALSE)
03    BEFORE FIELD field1
04       CALL DIALOG.setActionHidden("zoom",1)
05    AFTER FIELD field1
06       CALL DIALOG.setActionHidden("zoom",0)
07    ON CHANGE cust_name
```

```
08       CALL DIALOG.setFieldActive( "cust_addr", (rec.cust_name IS NOT
NULL) )
```

See ui.Dialog class for more details.

Control Functions

For backward compatibility, the language provides several built-in functions and operators to use in a `DIALOG` block. You can use the following built-in functions to keep track of the relative states of the current row, the program array, and the screen array, or to access the field buffers and keystroke buffers: ARR_CURR(), ARR_COUNT(), FGL_SET_ARR_CURR(), SET_COUNT(), FIELD_TOUCHED(), GET_FLDBUF(), INFIELD(), FGL_DIALOG_GETFIELDNAME(), FGL_DIALOG_GETBUFFER().

These functions and operators are provided for backward compatibility; you should use ui.Dialog methods instead.

The ARR_CURR(), ARR_COUNT() and FGL_SET_ARR_CURR() functions will work as in singular interactive instructions, when a `DISPLAY ARRAY` or `INPUT ARRAY` list has the focus.

Examples

Example 1: Two Lists

Form file:

```
01 LAYOUT
02 GRID
03 {
04 <t t1              >
05 [f11  |f12         ]
06 <                  >
07 <t t2              >
08 [f21  |f22         ]
09 <                  >
10 }
11 END
12 END
13 ATTRIBUTES
14 EDIT f11 = FORMONLY.column_11;
15 EDIT f12 = FORMONLY.column_12;
16 EDIT f21 = FORMONLY.column_21;
17 EDIT f22 = FORMONLY.column_22;
18 END
19 INSTRUCTIONS
20 SCREEN RECORD sr1(FORMONLY.column_11,FORMONLY.column_12);
```

```
21 SCREEN RECORD sr2(FORMONLY.column_21,FORMONLY.column_22);
22 END
```

Program file:

```
01 DEFINE
02    arr1 DYNAMIC ARRAY OF RECORD
03         column_11 INTEGER,
04         column_12 VARCHAR(10)
05    END RECORD,
06    arr2 DYNAMIC ARRAY OF RECORD
07         column_21 INTEGER,
08         column_22 VARCHAR(10)
09    END RECORD
10
11 MAIN
12  DEFINE i INTEGER
13  FOR i = 1 TO 20
14     LET arr1[i].column_11 = i
15     LET arr1[i].column_12 = "aaa "||i
16     LET arr2[i].column_21 = i
17     LET arr2[i].column_22 = "aaa "||i
18  END FOR
19  OPTIONS INPUT WRAP
20  OPEN FORM f FROM "lists"
21  DISPLAY FORM f
22  DIALOG ATTRIBUTES(UNBUFFERED)
23     DISPLAY ARRAY arr1 TO sr1.*
24       BEFORE DISPLAY
25         MESSAGE "We are in list one"
26     END DISPLAY
27     DISPLAY ARRAY arr2 TO sr2.*
28       BEFORE DISPLAY
29         MESSAGE "We are in list two"
30     END DISPLAY
31     ON ACTION close
32       EXIT DIALOG
33  END DIALOG
34 END MAIN
```

Example 2: Query and Lists

Form file:

```
01 LAYOUT ( TEXT = "Query customers", STYLE = "dialog3" )
02 GRID
03{
04 >g g1
<
05  Id: [f1        ] Name: [f2                                    ]
06                   State: [f3                      ]
07                    City: [f4                      ]
08                 Zipcode: [f5                      ]
09 [                                                 :cc     :sr
]
```

```
10 <
>
11 >g g2
<
12  >t t1                                                                <
13   Id        Name                          Timestamp
14  [c1       |c2                           |c3                         ]
15  [c1       |c2                           |c3                         ]
16  [c1       |c2                           |c3                         ]
17  [c1       |c2                           |c3                         ]
18  [c1       |c2                           |c3                         ]
19  [c1       |c2                           |c3                         ]
20  [c1       |c2                           |c3                         ]
21  [c1       |c2                           |c3                         ]
22  <                                                                    >
23 <
>
24[                                                          :cw
]
25}
26 END
27 END
28
29 ATTRIBUTES
30
31 GROUP g1 : TEXT = "Search criterias";
32 EDIT f1 = FORMONLY.cust_id TYPE INTEGER;
33 EDIT f2 = FORMONLY.cust_name TYPE VARCHAR;
34 COMBOBOX f3 = FORMONLY.cust_state TYPE VARCHAR,
35   QUERYEDITABLE, DEFAULT="CA", INITIALIZER=combo_fill_states;
36 BUTTONEDIT f4 = FORMONLY.cust_city TYPE CHAR, ACTION=zoom_city,
IMAGE="find";
37 EDIT f5 = FORMONLY.cust_zipcode TYPE VARCHAR;
38
39 GROUP g2 : TEXT = "Customer list";
40 EDIT c1 = FORMONLY.c_id TYPE INTEGER;
41 EDIT c2 = FORMONLY.c_name TYPE VARCHAR;
42 DATEEDIT c3 = FORMONLY.c_ts TYPE DATETIME YEAR TO SECOND;
43
44 BUTTON cc : clear, TEXT="Clear";
45 BUTTON sr : fetch, TEXT="Fetch";
46
47 BUTTON cw : close;
48
49 END
50
51 INSTRUCTIONS
52 SCREEN RECORD sr (FORMONLY.c_id THROUGH FORMONLY.c_ts);
53 END
```

Program file:

```
01 MAIN
02    DEFINE custarr DYNAMIC ARRAY OF RECORD
03              c_id INTEGER,
04              c_name VARCHAR(50),
```

```
05            c_ts DATETIME YEAR TO SECOND
06          END RECORD
07    DEFINE where_clause STRING
08
09    OPTIONS INPUT WRAP
10
11    OPEN FORM f1 FROM "QueryCustomers"
12    DISPLAY FORM f1
13
14    LET custarr[1].c_id = 123
15    LET custarr[1].c_name = "Parker"
16    LET custarr[1].c_ts = CURRENT YEAR TO SECOND
17    LET custarr[2].c_id = 124
18    LET custarr[2].c_name = "Duran"
19    LET custarr[2].c_ts = CURRENT YEAR TO SECOND
20
21    DIALOG ATTRIBUTES(FIELD ORDER FORM, UNBUFFERED)
22
23        CONSTRUCT BY NAME where_clause
24             ON cust_id, cust_name, cust_state, cust_city,
cust_zipcode
25           ON ACTION clear
26             CLEAR cust_id, cust_name, cust_state, cust_city,
cust_zipcode
27        END CONSTRUCT
28
29        DISPLAY ARRAY custarr TO sr.*
30          BEFORE ROW
31            MESSAGE SFMT("Row: %1/%2", DIALOG.getCurrentRow("sr"),
DIALOG.getArrayLength("sr"))
32        END DISPLAY
33
34        ON ACTION fetch
35           DISPLAY where_clause
36           -- Execute SQL query here to fill custarr ...
37
38        ON ACTION close
39           EXIT DIALOG
40
41    END DIALOG
42
43 END MAIN
```

Prompt for Values

Summary:

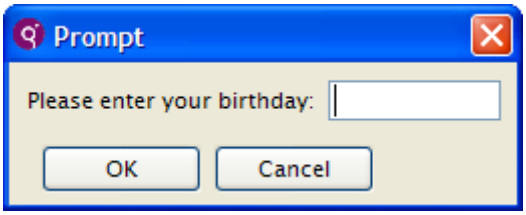
- Basics
- Syntax
- Usage
  - Programming Steps
  - Instruction Configuration
  - Default Actions
  - Interaction Blocks
- Examples
  - Example 1: Simple PROMPT
  - Example 2: PROMPT with interrupt checking
  - Example 3: PROMPT with ATTRIBUTE and ON ACTION handlers

See also: Record Input, Programs, Variables

Basics

The `PROMPT` instruction can be used to query for a single value from the user.

In GUI mode, the `PROMPT` instruction opens a modal window with an OK button and a Cancel button.



Syntax

Purpose:

The `PROMPT` statement assigns a user-supplied value to a variable.

Syntax:

```
PROMPT question
  [ ATTRIBUTES ( question-attribute [,...] ) ]
  FOR [CHAR[ACTER]] variable
```

```
  [ HELP number ]
  [ ATTRIBUTES ( input-attribute [,...] ) ]
[ dialog-control-block
  [...]
END PROMPT ]
```

where *dialog-control-block* is one of :

```
{ ON IDLE idle-seconds
| ON ACTION action-name
| ON KEY ( key-name [,...] )
}
  statement
  [...]
```

Notes:

1. *question* is a string expression displayed as a message for the input of the value.
2. *question-attribute* defines the display attributes for the *question.* See below for more details.
3. *variable* is the name of the variable that receives the data typed by the user.
4. The `FOR CHAR` clause exits the prompt statement when the first character has been typed.
5. *number* is the help message number to be displayed when the user presses the help key.
6. *input-attribute* is a display and control attribute for the input area. See below for more details.
7. *key-name* is an hot-key identifier (such as `F11` or `Control-z`).
8. *action-name* identifies an action that can be executed by the user.
9. *idle-seconds* is an integer literal or variable that defines a number of seconds.
10. *statement* is an instruction that is executed when the user presses the key defined by *key-name.*

The following display attributes can be used for both *question-attribute* and *input-attribute*:

Attribute	Description
`BLACK, BLUE, CYAN, GREEN, MAGENTA, RED, WHITE, YELLOW`	The color of the displayed text.
`BOLD, DIM, INVISIBLE, NORMAL`	The font attribute of the displayed text.
`REVERSE, BLINK, UNDERLINE`	The video attribute of the displayed text.

The following control attributes can be used for *input-attribute*:

Attribute	Description
`CENTURY = string`	Specify a year format indicator as defined in the

	CENTURY attribute of form files.
`FORMAT = string`	Specify a display format as defined in the FORMAT attribute of form files.
`PICTURE = string`	Specify an input picture as defined in the PICTURE attribute of form files.
`SHIFT = string`	Specify uppercase or lowercase shift. Values can be `'up'` or `'down'`.
`WITHOUT DEFAULTS [ =bool]`	Prompt must show by default the value of the program variable.
`CANCEL = bool`	Indicates if the default *cancel* action should be added to the dialog. If not specified, the action is added (`CANCEL=TRUE`).
`ACCEPT = bool`	Indicates if the default *accept* action should be added to the dialog. If not specified, the action is added (`ACCEPT=TRUE`).

Usage

You can use the `PROMPT` instruction to manage a single value input. You provide the text of the question to be displayed to the user and the variable that receives the value entered by the user. The runtime system displays the question in the prompt area (typically a popup window), waits for the user to enter a value, reads whatever value was entered until the user validates (for example with the RETURN key), and stores this value in a response variable. The prompt dialog remains visible until the user enters a response.

Warnings:

1. The `ON KEY` blocks are provided for backward compatibility; use `ON ACTION` instead.
2. The prompt finishes after `ON IDLE`, `ON ACTION`, or `ON KEY` block execution (to ensure backwards compatibility).

Programming Steps

To use the `PROMPT` statement, you must:

1. Declare a program variable with the DEFINE statement.
2. Describe the `PROMPT` statement, with *dialog-control-blocks* to control the instruction.
3. After executing the `PROMPT`, check the INT_FLAG variable to determine whether the input was validated or canceled by the user. See Example 2 below.

Instruction Configuration

HELP option

The HELP clause specifies the number of a help message to display if the user invokes the help while executing the instruction. The predefined *help* action is automatically created by the runtime system. You can bind action views to the *help* action.

ACCEPT option

The ACCEPT attribute can be set to FALSE to avoid the automatic creation of the *accept* default action.

CANCEL option

The CANCEL attribute can be set to FALSE to avoid the automatic creation of the *cancel* default action. If the CANCEL=FALSE option is set, no *close* action will be created, and you must write an ON ACTION close control block to create a close action.

Default Actions

When an PROMPT instruction executes, the runtime system creates a set of default actions.

The following table lists the default actions created for this dialog:

Default action	Description
accept	Validates the PROMPT dialog (validates field criterias) *Creation can be avoided with the ACCEPT attribute.*
cancel	Cancels the PROMPT dialog (no validation, int_flag is set) *Creation can be avoided with the CANCEL attribute.*
close	By default, cancels the PROMPT dialog (no validation, int_flag is set) Default action view is hidden. See Windows closed by the user.
help	Shows the help topic defined by the HELP clause. *Only created when a HELP clause is defined.*

Interaction Blocks

ON ACTION block

You can use `ON ACTION` blocks to execute a sequence of instructions when the user raises a specific action. This is the preferred solution compared to `ON KEY` blocks, because `ON ACTION` blocks use abstract names to control user interaction.

Warning: Because of backward compatibility, the prompt is finished after `ON IDLE`, `ON ACTION`, `ON KEY` block execution.

ON IDLE block

The `ON IDLE` *`idle-seconds`* clause defines a set of instructions that must be executed after *idle-seconds* of inactivity. The parameter *idle-seconds* must be an integer literal or variable. If it evaluates to zero, the timeout is disabled.

ON KEY block

For backward compatibility, you can use `ON KEY` blocks to execute a sequence of instructions when the user presses a specific key. The following key names are accepted by the compiler:

Key Name	Description
`ACCEPT`	The validation key.
`INTERRUPT`	The interruption key.
`ESC` or `ESCAPE`	The ESC key (not recommended, use `ACCEPT` instead).
`TAB`	The TAB key (not recommended).
`Control-`*`char`*	A control key where *char* can be any character except A, D, H, I, J, K, L, M, R, or X.
`F1` through `F255`	A function key.
`DELETE`	The key used to delete a new row in an array.
`INSERT`	The key used to insert a new row in an array.
`HELP`	The help key.
`LEFT`	The left arrow key.
`RIGHT`	The right arrow key.
`DOWN`	The down arrow key.
`UP`	The up arrow key.
`PREVIOUS` or `PREVPAGE`	The previous page key.
`NEXT` or `NEXTPAGE`	The next page key.

Examples

Example 1: Simple PROMPT

```
01 MAIN
02   DEFINE birth DATE
03   DEFINE chkey CHAR(1)
04   PROMPT "Please enter your birthday: " FOR birth
05   DISPLAY "Your birthday is : " || birth
06   PROMPT "Now press a key... " FOR CHAR chkey
07   DISPLAY "You pressed: " || chkey
08 END MAIN
```

Example 2: Simple PROMPT with Interrupt Checking

```
01 MAIN
02   DEFINE birth DATE
03   LET INT_FLAG = FALSE
04   PROMPT "Please enter your birthday: " FOR birth
05   IF INT_FLAG THEN
06     DISPLAY "Interrupt received."
07   ELSE
08     DISPLAY "Your birthday is : " || birth
09   END IF
10 END MAIN
```

Example 3: PROMPT with ATTRIBUTES and ON ACTION handlers

```
01 MAIN
02   DEFINE birth DATE
03   LET birth = TODAY
04   PROMPT "Please enter your birthday: " FOR birth
05      ATTRIBUTES(WITHOUT DEFAULTS)
06         ON ACTION action1
07            DISPLAY "Action 1"
08   END PROMPT
09   DISPLAY "Your birthday is " || birth
10 END MAIN
```

Displaying Messages

Summary:

- Displaying Text in Line Mode (DISPLAY)
- Displaying Error Messages (ERROR)
- Displaying Application Messages (MESSAGE)

DISPLAY

Purpose:

The DISPLAY instruction displays text in line mode to the standard output channel.

Syntax:

```
DISPLAY expression [,...]
```

Notes:

1. *expression* is any expression supported by the language.

Usage:

You can use this instruction to display information to the standard output channel.

The values contained in variables are formatted based on the data type and environment settings.

Example:

```
01 MAIN
02   DISPLAY "Today's date is: ", TODAY
03 END MAIN
```

ERROR

Purpose:

The ERROR instruction displays an error message to the user.

Syntax:

```
ERROR expression [,...] [ ATTRIBUTE ( display-attribute [,...] ) ]
```

Notes:

1. *expression* is any expression supported by the language.
2. *display-attribute* is an attribute to display the error text. See below.

Usage:

The `ERROR` instruction displays an error message to the user.

In TUI mode, the error text is displayed in the Error Line of the current window. In GUI mode, the text is displayed in a specific area, depending on the front end configuration.

Possible attributes that can be used as *display-attribute*:

Attribute	Description
`BLACK, BLUE, CYAN, GREEN, MAGENTA, RED, WHITE, YELLOW`	The color of the displayed text.
`BOLD, DIM, INVISIBLE, NORMAL`	The font attribute of the displayed text.
`REVERSE, BLINK` (**TUI Only!**), `UNDERLINE`	The video attribute of the displayed text.

Example:

```
01 MAIN
02    WHENEVER ERROR CONTINUE
03    DATABASE stock
04    WHENEVER ERROR STOP
05    IF sqlca.sqlcode THEN
06       ERROR "Connection failed (" || sqlca.sqlcode || ")"
07    END IF
08 END MAIN
```

MESSAGE

Purpose:

The `MESSAGE` instruction displays a message to the user.

Syntax:

```
MESSAGE message [,...] [ ATTRIBUTE ( display-attribute [,...] ) ]
```

Notes:

1. *expression* is any expression supported by the language.

2. *display-attribute* is an attribute to display the error text. See below.

Usage:

The `MESSAGE` instruction displays a message to the user.

In TUI mode, the text is displayed in the Comment Line of the current window. In GUI mode, the text is displayed in a specific area, depending on the front end configuration.

Possible attributes that can be used as *display-attribute*:

Attribute	Description
`BLACK, BLUE, CYAN, GREEN, MAGENTA, RED, WHITE, YELLOW`	The color of the displayed text.
`BOLD, DIM, INVISIBLE, NORMAL`	The font attribute of the displayed text.
`REVERSE, BLINK` (**TUI Only!**), `UNDERLINE`	The video attribute of the displayed text.

Example:

```
01 MAIN
02    WHENEVER ERROR CONTINUE
03    DATABASE stock
04    WHENEVER ERROR STOP
05    IF sqlca.sqlcode THEN
06       ERROR "Connection failed (" || sqlca.sqlcode || ")"
07    ELSE
08       MESSAGE "Connected to database."
09    END IF
10 END MAIN
```

Toolbars

Summary:

- Basics
- Syntax
- Usage
  - Defining ToolBars
  - Toolbar structure
  - Defining a Toolbar in form file
  - Loading a Toolbar from a file
  - Loading the default Toolbar from a file
  - Creating a Toolbar dynamically
- Examples
  - Simple Toolbar in XML format
  - Program creating a Toolbar dynamically
  - Toolbar definition in a PER file

See also: Action Defaults, Topmenus, Form Specification Files

---

Basics

A Toolbar is a view for actions presented as a set of buttons that can trigger events in an interactive instruction. This page describes how to use toolbars in programs; it is also possible to define toolbars in forms with the TOOLBAR section.

Syntax

```
<ToolBar [ toolbar-attribute="value" [...] ] >
  { <ToolBarSeparator separator-attribute="value" [...] />
  | <ToolBarItem item-attribute="value" [...] />
  } [...]
</ToolBar>
```

Notes:

1. *toolbar-attribute* defines a property of the toolbar.
2. *item-attribute* defines a property of the toolbar item.

Warnings:

1. The DOM tag names are case sensitive; `Toolbar` is different from `ToolBar`.
2. When binding to an action, make sure that you are using the right value in the `name` attribute. As `ON ACTION` and `COMMAND` generate lowercase identifiers, it is recommended to use **lowercase** `names`.

3. When binding to a key, make sure the `name` attribute value is in lowercase letters ("`f5`").
4. Make sure that the image file is available to the Front End.

Tips:

1. It is recommended that you define the decoration of a Toolbar item for common actions with Action Defaults.

Usage

Defining Toolbars

You can define a global/default toolbar at the program level, or you can define form-specific toolbars. The global toolbar is displayed by default in all windows, or in the global window container when using MDI. The form-specific toolbar is displayed in the form where it is defined. You can control the position and visibility of toolbars with a window style attribute. Typical "modal windows" do not display toolbars.

The Toolbar items (or buttons) are enabled according to the actions defined by the current interactive instruction, which can be MENU, INPUT, INPUT ARRAY, DISPLAY ARRAY, or CONSTRUCT. The action trigger bound to a Toolbar button is executed when the button is clicked.

A Toolbar item is bound to an *action node* of the current interactive instruction if its `name` attribute corresponds to an *action node* name (typically, the name of a ring menu option). A click on the Toolbar button has the same effect as raising the action. For example, if the current interactive instruction is a ring menu, this would have the same effect as selecting the ring menu option.

A Toolbar item is bound to a key trigger if the `name` attribute of the item corresponds to a valid hot key, in lowercase letters. In this case, a click on the Toolbar button has the same effect as pressing the hot key.

A Toolbar button is automatically disabled if the corresponding action is not available (for example, when a ring menu option is hidden).

Toolbar Structure

The following table shows the list of *toolbar-attributes* supported for the `ToolBar` node:

Attribute	Type	Description
`tag`	`STRING`	User-defined attribute to identify the node.
`name`	`STRING`	Identifies the Toolbar.

buttonTextHidden	INTEGER	Defines if the text of toolbar buttons must appear by default.

The following table shows the list of *item-attributes* supported for the ToolBarItem node:

Attribute	Type	Description
name	STRING	Identifies the action corresponding to the toolbar button. Can be prefixed with the sub-dialog identifier.
tag	STRING	User-defined attribute to identify the node.
text	STRING	The text to be displayed in the toolbar button.
comment	STRING	The message to be shown as tooltip when the user selects a toolbar button.
hidden	INTEGER	Indicates if the item is hidden.
image	STRING	The icon to be used in the toolbar button.

The following table shows the list of *separator-attributes* supported for the ToolBarSeparator node:

Attribute	Type	Description
tag	STRING	User-defined attribute to identify the node.
hidden	INTEGER	Indicates if the separator is hidden.

Defining toolbars in the form file

You can define a toolbar in the form specification file with the TOOLBAR section; see the example below.

Loading a Toolbar from an XML file

To load a Toolbar definition file, use the utility method provided by the Form built-in class:

```
01 CALL myform.loadToolbar("standard")
```

This method accepts a filename with or without the "4tb" extension. If you omit the file extension (recommended), the runtime system adds the extension automatically. If the file does not exist in the current directory, it is searched in the directories defined in the DBPATH environment variable.

If a form contains a specific toolbar loaded by the ui.Form.loadToolbar() method or defined in the Form Specification File, it will be replaced by the new toolbar loaded from this function.

Loading a default toolbar from an XML file

To load a default toolbar from an XML definition file, use the utility method provided by the Interface built-in class:

```
01 CALL ui.Interface.loadToolbar("standard")
```

This method accepts a filename with or without the "4tb" extension. If you omit the file extension (recommended), the runtime system adds the extension automatically. If the file does not exist in the current directory, it is searched in the directories defined in the DBPATH environment variable.

The default toolbar loaded by this method is also used for the MDI container.

Creating the toolbar manually with DOM

This example shows how to create a Toolbar in all forms by using the default initialization function and the DomNode class:

```
01 CALL ui.Form.setDefaultInitializer("myinit")
02 OPEN FORM f1 FROM "form1"
03 DISPLAY FORM f1
04 ...
05 FUNCTION myinit(form)
06   DEFINE form ui.Form
06   DEFINE f om.DomNode
08   LET f = form.getNode()
09   ...
10 END FUNCTION
```

After getting the DOM node of the form, create a node with the "ToolBar" tag name:

```
01 DEFINE tb om.DomNode
02 LET tb = f.createChild("ToolBar")
```

For each toolbar button, create a sub-node with the "ToolBarItem" tag name and set the attributes to define the button:

```
01 DEFINE tbi om.DomNode
02 LET tbi = tb.createChild("ToolBarItem")
03 CALL tbi.setAttribute("name","update")
04 CALL tbi.setAttribute("text","Modify")
05 CALL tbi.setAttribute("comment","Modify the current record")
```

```
06 CALL tbi.setAttribute("image","change")
```

If needed, you can create a "ToolBarSeparator" node to separate Toolbar buttons:

```
01 DEFINE tbs om.DomNode
02 LET tbs = tb.createChild("ToolBarSeparator")
```

Examples

Example 1: Simple Toolbar in XML format

```
01 <ToolBar>
02    <ToolBarItem name="f5" text="List" image="list" />
03    <ToolBarSeparator/>
04    <ToolBarItem name="query" text="Query" image="search" />
05    <ToolBarItem name="add" text="Append" image="add" />
06    <ToolBarItem name="delete" text="Delete" image="delete" />
07    <ToolBarItem name="modify" text="Modify" image="change" />
08    <ToolBarSeparator/>
09    <ToolBarItem name="f1" text="Help" image="list" />
10    <ToolBarSeparator/>
11    <ToolBarItem name="quit" text="Quit" image="quit" />
12 </ToolBar>
```

Example 2: Program creating the Toolbar dynamically

```
01 MAIN
02   DEFINE aui om.DomNode
03   DEFINE tb  om.DomNode
04   DEFINE tbi om.DomNode
05   DEFINE tbs om.DomNode
06
07   LET aui = ui.Interface.getRootNode()
08
09   LET tb = aui.createChild("ToolBar")
10
11   LET tbi = createToolBarItem(tb,"f1","Help","Show help","help")
12   LET tbs = createToolBarSeparator(tb)
13   LET tbi = createToolBarItem(tb,"upd","Modify","Modify current
record","change")
14   LET tbi = createToolBarItem(tb,"del","Remove","Remove current
record","delete")
15   LET tbi = createToolBarItem(tb,"add","Append","Add a new
record","add")
16   LET tbs = createToolBarSeparator(tb)
17   LET tbi = createToolBarItem(tb,"xxx","Exit","Quit
application","quit")
18
19   MENU "Example"
20     COMMAND KEY(F1)
21       DISPLAY "F1 action received"
22     COMMAND "upd"
```

```
23       DISPLAY "Update action received"
24     COMMAND "Del"
25       DISPLAY "Delete action received"
26     COMMAND "Add"
27       DISPLAY "Append action received"
28     COMMAND "xxx"
29       EXIT PROGRAM
30   END MENU
31
32 END MAIN
33
34 FUNCTION createToolBarSeparator(tb)
35   DEFINE tb om.DomNode
36   DEFINE tbs om.DomNode
37   LET tbs = tb.createChild("ToolBarSeparator")
38   RETURN tbs
39 END FUNCTION
40
41 FUNCTION createToolBarItem(tb,n,t,c,i)
42   DEFINE tb om.DomNode
43   DEFINE n,t,c,i VARCHAR(100)
44   DEFINE tbi om.DomNode
45   LET tbi = tb.createChild("ToolBarItem")
46   CALL tbi.setAttribute("name",n)
47   CALL tbi.setAttribute("text",t)
48   CALL tbi.setAttribute("comment",c)
49   CALL tbi.setAttribute("image",i)
50   RETURN tbi
51 END FUNCTION
```

Example 3: Toolbar definition in a PER file

```
01 TOOLBAR
02   ITEM accept ( TEXT="Ok", IMAGE="ok" )
03   ITEM cancel ( TEXT="cancel", IMAGE="cancel" )
04   SEPARATOR
05   ITEM editcut   -- Gets decoration from action defaults
06   ITEM editcopy  -- Gets decoration from action defaults
07   ITEM editpaste -- Gets decoration from action defaults
08   SEPARATOR
09   ITEM append ( TEXT="Append", IMAGE="add" )
10   ITEM update ( TEXT="Update", IMAGE="modify" )
11   ITEM delete ( TEXT="Delete", IMAGE="del" )
12   ITEM search ( TEXT="Search", IMAGE="find" )
13 END
```

Topmenus

Summary:

- Basics
- Syntax
- Usage
  - Defining TopMenus
  - TopMenu Structure
  - Defining a TopMenu in form file
  - Loading a TopMenu from a file
  - Loading a default TopMenu from a file
  - Creating a TopMenu dynamically
- Examples
  - Simple Topmenu in XML format
  - Topmenu definition in a PER file

See also: Action Defaults, Toolbars, Form Specification Files

---

Basics

A Topmenu is a view for actions presented as a typical pull-down menu, having options that can trigger events in an interactive instruction. This page describes how to use Topmenus in programs; it is also possible to define Topmenus in forms with the TOPMENU section.

Syntax

```
<TopMenu [ topmenu-attribute="value" [...] ] >
  group
  [...]
</TopMenu>
```

where *group* is:

```
<TopMenuGroup group-attribute="value" [...]>
  { <TopMenuSeparator separator-attribute="value" [...] />
  | <TopMenuCommand command-attribute="value" [...] />
  | group
  } [...]
</TopMenuGroup>
```

Notes:

1. The `TopMenu` node can hold `TopMenuSeparator`, `TopMenuGroup` or `TopMenuCommand` children.
2. *topmenu-attribute* defines a property of the `TopMenu`.
3. A `TopMenuGroup` node can hold `TopMenuSeparator`, `TopMenuGroup` or `TopMenuCommand` children.
4. *group-attribute* defines a property of a `TopMenuGroup`.
5. A `TopMenuCommand` node defines a leaf of the TopMenu tree and can be selected to execute an action.
6. *command-attribute* defines a property of a `TopMenuCommand`.
7. A `TopMenuSeparator` node defines an horizontal line in a TopMenu group.
8. *separator-attribute* defines a property of a `TopMenuSeparator`.

Warnings:

1. The DOM tag names are case sensitive; `Topmenu` is different from `TopMenu`.
2. When binding to an action, make sure that you are using the right value in the `name` attribute. As `ON ACTION` and `COMMAND` generate lowercase identifiers, it is recommended to use **lowercase** `names`.
3. When binding to a key, make sure the `name` attribute value is in lowercase letters ("`f5`").
4. Make sure that the image file is available to the Front End.

Tips:

1. For common actions, it is recommended that you define the decoration of a Topmenu command with Action Defaults.

Usage

Defining TopMenus

A Topmenu defines a graphical pull-down menu that holds views for actions controlled in BDL programs with `ON ACTION` clauses. See Interaction Model for more details about action management.

You can define a Topmenu in form files with the TOPMENU section, or you can load a Topmenu at runtime into the current form by using the following methods: ui.Form.loadTopMenu()

The Topmenu commands (pull-down menu options) are enabled according to the actions defined by the current interactive instruction, which can be MENU, INPUT, INPUT ARRAY, DISPLAY ARRAY or CONSTRUCT. When a Topmenu option is selected, the program executes the action trigger the Topmenu command is bound to.

A Topmenu command is bound to an *action node* of the current interactive instruction if its `name` attribute corresponds to an *action node* name (typically, the name of a ring menu option). In this case, a Topmenu command selection has the same effect as raising the action. For example, if the current interactive instruction is a ring menu, this would have the same effect as selecting the ring menu option.

A Topmenu command is bound to a key trigger if the `name` attribute of the command corresponds to a valid hot-key, in lowercase letters. In this case, selecting a Topmenu command has the same effect as pressing the hot-key.

A Topmenu command is automatically disabled if the corresponding action is not available (for example, when a ring menu option is hidden).

TopMenu structure

A Topmenu is part of a form definition; the `TopMenu` node must be created under the `Form` node.

The following table shows the list of *topmenu-attributes* supported for the `TopMenu` node:

Attribute	Type	Description
`tag`	`STRING`	User-defined attribute to identify the node.

The following table shows the list of *command-attributes* supported for the `TopMenuCommand` node:

Attribute	Type	Description
`name`	`STRING`	Identifies the action corresponding to the Topmenu command. Can be prefixed with the sub-dialog identifier.
`tag`	`STRING`	User-defined attribute to identify the node.
`text`	`STRING`	The text to be displayed in the pull-down menu option.
`comment`	`STRING`	The message to be shown for this element.
`hidden`	`INTEGER`	Indicates if the command is hidden.
`image`	`STRING`	The icon to be used in the pull-down menu option.
`acceleratorName`	`STRING`	Defines the accelerator name to be display on the left of the menu option text. Note this attribute is only used for decoration (you must also define an action default accelerator).

In order to define the tree structure of the pull-down menu, the `TopMenuGroup` node is provided to hold Topmenu commands and Topmenu groups:

```
TopMenu
  +- TopMenuGroup
     +- TopMenuCommand
     +- TopMenuCommand
     +- TopMenuCommand
  +- TopMenuGroup
     +- TopMenuGroup
        +- TopMenuCommand
        +- TopMenuCommand
     +- TopMenuGroup
        +- TopMenuCommand
        +- TopMenuCommand
        +- TopMenuCommand
```

The following table shows the list of *group-attributes* supported for the `TopMenuGroup` node:

Attribute	Type	Description
`tag`	`STRING`	User-defined attribute to identify the node.
`text`	`STRING`	The text to be displayed in the pull-down menu group.
`comment`	`STRING`	The message to be shown for this element.
`hidden`	`INTEGER`	Indicates if the group is hidden.
`image`	`STRING`	The icon to be used in the pull-down menu group. **Warning: Images cannot be displayed for the first level of TopMenuGroup elements.**

The following table shows the list of *separator-attributes* supported for the `TopMenuSeparator` node:

Attribute	Type	Description
`tag`	`STRING`	User-defined attribute to identify the node
`hidden`	`INTEGER`	Indicates if the separator is hidden.

Defining the Topmenu in a form file

You typically define a Topmenu in the form specification file, with the TOPMENU section; see below for an example.

Loading a Topmenu from an XML file

To load a Topmenu definition file, use the utility method provided by the Form built-in class:

```
01 CALL myform.loadTopMenu("standard")
```

This method accepts a filename with or without the "4tm" extension. If you omit the file extension (recommended), the runtime system adds the extension automatically. If the file does not exist in the current directory, it is searched in the directories defined in the DBPATH environment variable.

If a form contains a specific topmenu loaded by the ui.Form.loadTopmenu() method or defined in the Form Specification File, it will be replaced by the new topmenu loaded from this function.

Loading a default topmenu from an XML file

To load a default topmenu from an XML definition file, use the utility method provided by the Interface built-in class:

```
01 CALL ui.Interface.loadTopMenu("standard")
```

This method accepts a filename with or without the "4tm" extension. If you omit the file extension (recommended), the runtime system adds the extension automatically. If the file does not exist in the current directory, it is searched in the directories defined in the DBPATH environment variable.

The default topmenu loaded by this method is also used for the MDI container.

Creating the Topmenu dynamically

This example shows how to create a Topmenu in all forms by using the default initialization function and the DomNode class:

```
01 CALL ui.Form.setDefaultInitializer("myinit")
02 OPEN FORM f1 FROM "form1"
03 DISPLAY FORM f1
04 ...
05 FUNCTION myinit(form)
06   DEFINE form ui.Form
06   DEFINE f om.DomNode
08   LET f = form.getNode()
09   ...
10 END FUNCTION
```

After getting the DOM node of the form, create a node with the "TopMenu" tag name:

```
01 DEFINE tm om.DomNode
02 LET tm = f.createChild("TopMenu")
```

For each Topmenu group, create a sub-node with the "TopMenuGroup" tag name and set the attributes to define the group:

```
01 DEFINE tmg om.DomNode
02 LET tmg = tm.createChild("TopMenuGroup")
03 CALL tmg.setAttribute("text","Reports")
```

For each Topmenu option, create a sub-node in a group node with the "TopMenuCommand" tag name and set the attributes to define the option:

```
01 DEFINE tmi om.DomNode
02 LET tmi = tmg.createChild("TopMenuCommand")
03 CALL tmi.setAttribute("name","report")
04 CALL tmi.setAttribute("text","Order report")
05 CALL tmi.setAttribute("comment","Orders entered today")
06 CALL tmi.setAttribute("image","smiley")
```

If needed, you can create a "TopMenuSeparator" node inside a group, to separate menu options:

```
01 DEFINE tms om.DomNode
02 LET tms = tmg.createChild("TopMenuSeparator")
```

Examples

Example 1: Simple Topmenu in XML format

```
01 <TopMenu>
02    <TopMenuGroup text="Form" >
03       <TopMenuCommand name="help" text="Help" image="quest" />
04       <TopMenuCommand name="quit" text="Quit" acceleratorName="alt-
F4"/>
05    </TopMenuGroup>
06    <TopMenuGroup text="Edit" >
07       <TopMenuCommand name="accept" text="Validate" image="ok" />
08       <TopMenuCommand name="cancel" text="Cancel" image="cancel" />
09       <TopMenuSeparator/>
10       <TopMenuCommand name="editcut" text="Cut" />
11       <TopMenuCommand name="editcopy" text="Copy" />
12       <TopMenuCommand name="editpaste" text="Paste" />
13    </TopMenuGroup>
14    <TopMenuGroup text="Records" >
15       <TopMenuCommand name="append" text="Add" image="add" />
16       <TopMenuCommand name="delete" text="Remove" image="delete" />
17       <TopMenuCommand name="update" text="Modify" image="change" />
```

```
18       <TopMenuSeparator/>
19       <TopMenuCommand name="search" text="Query" image="find" />
20    </TopMenuGroup>
21 </TopMenu>
```

Example 2: Topmenu definition in a PER file

```
01 TOPMENU
02   GROUP form (TEXT="Form")
03      COMMAND help (TEXT="Help", IMAGE="quest")
04      COMMAND quit (TEXT="Quit", ACCELERATOR=ALT-F4)
05   END
06   GROUP edit (TEXT="Edit")
07      COMMAND accept (TEXT="Validate", IMAGE="ok")
08      COMMAND cancel (TEXT="Cancel", IMAGE="cancel")
09      SEPARATOR
10      COMMAND editcut   -- Gets decoration from action defaults
11      COMMAND editcopy  -- Gets decoration from action defaults
12      COMMAND editpaste -- Gets decoration from action defaults
13   END
14   GROUP records (TEXT="Records")
15      COMMAND append (TEXT="Add", IMAGE="add")
16      COMMAND delete (TEXT="Remove", IMAGE="del")
17      COMMAND update (TEXT="Modify", IMAGE="change")
18      SEPARATOR
19      COMMAND search (TEXT="Search", IMAGE="find")
20   END
21 END
```

StartMenus

Summary:

- Basics
- Syntax
- Usage
  - Defining StartMenus
  - StartMenu Structure
  - Loading a StartMenu from an XML file
  - Creating a StartMenu dynamically
- Examples
  - Simple StartMenu in XML format
  - Program creating a StartMenu dynamically

See also: Toolbars, Topmenus

Basics

The StartMenu defines a tree of commands that start programs on the application server where the runtime system executes.

Syntax

```
<StartMenu [ startmenu-attribute="value" [...] ] >
  group
  [...]
</StartMenu>
```

where *group* is:

```
<StartMenuGroup group-attribute="value" [...]>
  { <StartMenuSeparator/>
  | <StartMenuCommand command-attribute="value" [...] />
  | group
  } [...]
</StartMenuGroup>
```

Notes:

1. The `StartMenu` node can only hold `StartMenuGroup` children.
2. *startmenu-attribute* defines a property of the `StartMenu`. See below for more details.
3. A `StartMenuGroup` node can hold `StartMenuSeparator`, `StartMenuGroup`, or `StartMenuCommand` children.

4. *command-attribute* defines a property of a `StartMenuCommand`. See below for more details.
5. A `StartMenuCommand` node defines a leaf of the `StartMenu` tree that can be selected to start a program.
6. *group-attribute* defines a property of a `StartMenuGroup`. See below for more details.

Warnings:

1. The DOM tag names are case sensitive; `Startmenu` is different from `StartMenu`.

Usage

Defining StartMenus

It is recommended that you create a specific BDL program dedicated to running the Start Menu. This program must create (or load) a Start Menu, and then perform an interactive instruction to enter the interaction loop.

The StartMenu must be defined in the Abstract User Interface tree using the DOM API, under the "`UserInterface`" root node.

The StartMenu is unique for a program and cannot be redefined.

When a StartMenu command is selected by the user, the runtime system automatically starts a child process with the command specified in the command attribute.

StartMenu Structure

The following table shows the attributes of the `StartMenu` node:

Attribute	Type	Description
`name`	`STRING`	Identifies the StartMenu, not required.
`text`	`STRING`	Defines the text to be displayed as title.

The following table shows the attributes of the `StartMenuGroup` node:

Attribute	Type	Description
`disabled`	`INTEGER`	Indicates if the group must be disabled (grayed, cannot be selected).
`image`	`STRING`	Defines the icon to be used for this group.
`name`	`STRING`	Identifies the start menu group, not required.

text	STRING	Defines the text to be displayed for this group.

The following table shows the attributes of the StartMenuCommand node:

Attribute	Type	Description
disabled	INTEGER	Indicates if the item must be disabled (grayed, cannot be selected).
exec	STRING	Defines the command to be executed when the user selects this command.
image	STRING	Defines the icon to be used for this command.
name	STRING	Identifies the StartMenu item, not required.
text	STRING	Defines the text to be displayed for this command.
waiting	INTEGER	Defines if the command must be started without waiting (0, default) or waiting (1).

The following table shows the attributes of the StartMenuSeparator node:

Attribute	Type	Description
name	STRING	Identifies the StartMenu separator, not required.

Loading a StartMenu from an XML file

To load a StartMenu definition file, use the utility method provided by the Interface built-in class:

```
01 CALL ui.Interface.loadStartMenu("standard")
```

This method accepts a filename with or without the "4sm" extension. If you omit the file extension (recommended), the runtime system adds the extension automatically. If the file does not exist in the current directory, it is searched in the directories defined in the DBPATH environment variable.

Creating the StartMenu dynamically

You can create a StartMenu dynamically with the DomNode class:

First, get the AUI root node:

```
01 DEFINE aui om.DomNode
02 LET aui = ui.Interface.getRootNode()
```

Next, create a node with the "StartMenu" tag name:

```
01 DEFINE sm om.DomNode
02 LET sm = aui.createChild("StartMenu")
```

Next, create a "StartMenuGroup" node to group a couple of command nodes:

```
01 DEFINE smg om.DomNode
02 LET smg = sm.createChild("StartMenuGroup")
03 CALL smg.setAttribute("text","Programs")
```

Then, create "StartMenuCommand" nodes for each program and, if needed, add "StartMenuSeparator" nodes to separate entries:

```
01 DEFINE smc, sms om.DomNode
02 LET smc = smg.createChild("StartMenuCommand")
03 CALL smc.setAttribute("text","Orders")
04 CALL smc.setAttribute("exec","fglrun orders.42r")
05 LET smc = smg.createChild("StartMenuCommand")
06 CALL smc.setAttribute("text","Customers")
07 CALL smc.setAttribute("exec","fglrun customers.42r")
08 LET sms = smg.createChild("StartMenuSeparator")
09 LET smc = smg.createChild("StartMenuCommand")
10 CALL smc.setAttribute("text","Items")
11 CALL smc.setAttribute("exec","fglrun items.42r")
```

Examples

Example 1: Simple StartMenu in XML format

```
01 <StartMenu>
02    <StartMenuGroup text="Ordering" >
03       <StartMenuCommand text="Orders" exec="fglrun orders.42r" />
04       <StartMenuCommand text="Customers" exec="fglrun custs.42r" />
05       <StartMenuCommand text="Items" exec="fglrun items.42r" />
06    <StartMenuCommand text="Reports" exec="fglrun reports.42r" />
07    </StartMenuGroup>
08    <StartMenuGroup text="Configuration" >
09       <StartMenuCommand text="Database" exec="fglrun dbseconf.42r"
/>
10       <StartMenuCommand text="Users" exec="fglrun userconf.42r" />
11       <StartMenuCommand text="Printers" exec="fglrun prntconf.42r"
/>
12    </StartMenuGroup>
13 </StartMenu>
```

Example 2: Program creating the StartMenu dynamically

```
01 MAIN
02   DEFINE aui om.DomNode
03   DEFINE sm  om.DomNode
```

```
04   DEFINE smg om.DomNode
05   DEFINE smc om.DomNode
06
07   LET aui = ui.Interface.getRootNode()
08
09   LET sm = aui.createChild("StartMenu")
10
11   LET smg = createStartMenuGroup(sm,"Ordering")
13   LET smc = createStartMenuCommand(smg,"Orders","fglrun
orders.42r",NULL)
14   LET smc = createStartMenuCommand(smg,"Customers","fglrun
custs.42r",NULL)
15   LET smc = createStartMenuCommand(smg,"Items","fglrun
items.42r",NULL)
16   LET smc = createStartMenuCommand(smg,"Reports","fglrun
reports.42r",NULL)
17   LET smg = createStartMenuGroup(sm,"Configuration")
18   LET smc = createStartMenuCommand(smg,"Database","fglrun
dbseconf.42r",NULL)
19   LET smc = createStartMenuCommand(smg,"Users","fglrun
userconf.42r",NULL)
20   LET smc = createStartMenuCommand(smg,"Printers","fglrun
prntconf.42r",NULL)
21
22   MENU "Example"
23     COMMAND "Quit"
24       EXIT PROGRAM
25   END MENU
26
27 END MAIN
28
29 FUNCTION createStartMenuGroup(p,t)
30   DEFINE p om.DomNode
31   DEFINE t STRING
32   DEFINE s om.DomNode
33   LET s = p.createChild("StartMenuGroup")
34   CALL s.setAttribute("text",t)
35   RETURN s
36 END FUNCTION
37
38 FUNCTION createStartMenuCommand(p,t,c,i)
39   DEFINE p om.DomNode
40   DEFINE t,c,i STRING
41   DEFINE s om.DomNode
42   LET s = p.createChild("StartMenuCommand")
43   CALL s.setAttribute("text",t)
44   CALL s.setAttribute("exec",c)
45   CALL s.setAttribute("image",i)
46   RETURN s
47 END FUNCTION
```

Canvas

This page describes the usage of Canvas.

- What is Canvas?
- Basics
- Functions

See also: Forms, Windows

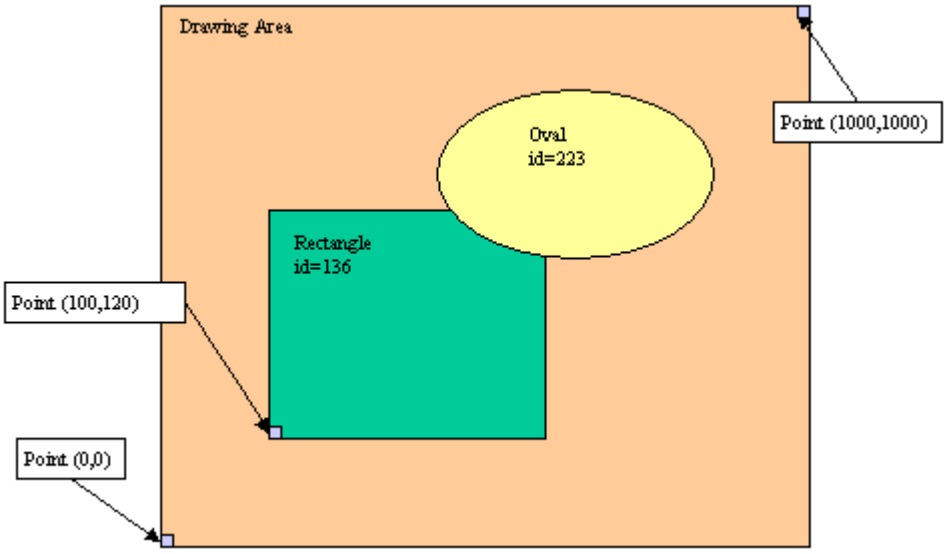
What is Canvas?

By using Canvas, you can draw simple shapes in a specific area of a form. Canvas can draw lines, rectangles, ovals, circles, texts, arcs, and polygons. Keys can be bound to graphical elements for selection with a right or left mouse click.

In programs, you select a given Canvas area by name and you create the shapes in the Abstract User Interface tree by using the built-in DOM API.

The painted canvas is automatically displayed on the front end when an interactive instruction is executed (like MENU or INPUT).

The canvas area represents an abstract drawing page where you define size and location of shapes with coordinates from (0,0) to (1000,1000). The origin point (0,0), is on the left-bottom of the drawing area.



Each canvas element is identified by a unique number (id). You can use this identifier to bind mouse clicks to canvas elements.

Basics

Purpose:

Use **Canvas** to draw simple shapes in a specific area of a form.

Syntax:

```
<Canvas colName="name" >
  { <CanvasArc canvasitem-attribute="value" [...] />
  | <CanvasCircle canvasitem-attribute="value" [...] />
  | <CanvasLine canvasitem-attribute="value" [...] />
  | <CanvasOval canvasitem-attribute="value" [...] />
  | <CanvasPolygon canvasitem-attribute="value" [...] />
  | <CanvasRectangle canvasitem-attribute="value" [...] />
  | <CanvasText canvasitem-attribute="value" [...] />
  } [...]
</Canvas>
[...]
```

Notes:

1. You define the content of canvas areas in the Abstract User Interface tree.
2. If the form defines canvas areas, the Abstract User Interface tree contains empty `<Canvas>` nodes that you can fill with canvas items.
3. A canvas node is identified in the program by the `colName` attribute.
4. You can get the canvas DomNode by name with the `Window.getElement(name)` method.
5. You cannot drop canvas nodes, as they are read-only in a form definition.

The following table describes all the types of canvas element that are supported:

Name	Description
`CanvasArc`	Arc defined by a center point, a diameter, a start angle, a end angle, and a fill color.
`CanvasCircle`	Circle defined by the bounding square top left point, a diameter, and a fill color.
`CanvasLine`	Line defined by a start point, an end point, a width, and a fill color.
`CanvasOval`	Oval defined by rectangle (with start point and end point), and a fill color.
`CanvasPolygon`	Polygon defined by a list of points, and a fill color.
`CanvasRectangle`	Rectangle defined by a start point, an end point, and a fill color.
`CanvasText`	Text defined by a start point, an anchor hint, the text, and a fill color.

The following table describes the attributes of canvas elements:

Name	Values	Description
startX	INTEGER (0->1000)	X position of starting point.
startY	INTEGER (0->1000)	Y position of starting point.
endX	INTEGER (0->1000)	X position of ending point.
endY	INTEGER (0->1000)	Y position of ending point.
xyList	STRING	Space-separated list of X Y coordinates. For example: "23 45 56 78".
width	INTEGER	Width of the shape.
height	INTEGER	Height of the shape.
diameter	INTEGER	Diameter for circles and arcs.
startDegrees	INTEGER	Beginning of the angular range occupied by an arc.
extentDegrees	INTEGER	Size of the angular range occupied by an arc.
text	STRING	The text to draw.
anchor	"n","e","w","s"	Anchor hint to give the draw direction for texts.
fillColor	STRING	Name of the color to be used for the element.
acceleratorKey1	STRING	Name of the key associated to a left button click.
acceleratorKey3	STRING	Name of the key associated to a right button click.

Usage:

First, you must define a drawing area in the form file. The drawing area is defined by a form field declared with the attribute WIDGET="CANVAS". In the following example, the name of the canvas field is '`canvas01`'. This field name identifies the drawing area:

```
01 DATABASE FORMONLY
02 LAYOUT
03 GRID
04 {
05   Canvas example:
06   [ca01                          ]
07   [                              ]
08   [                              ]
09   [                              ]
10   [                              ]
```

```
11   [                              ]
12 }
13 END
14 END
15 ATTRIBUTES
16 CANVAS ca01 : canvas01;
17 END
```

In programs, you draw canvas shapes by creating Canvas nodes in the Abstract User Interface tree with the DOM API utilities.

Define a variable to hold the DOM node of the canvas and a second to handle children created for shapes:

```
01 DEFINE c, s om.DomNode
```

Define a window object variable; open a window with the form containing the canvas area; get the current window object, and then get the canvas DOM node:

```
02 DEFINE w ui.Window
03 OPEN WINDOW w1 WITH FORM "form1"
04 LET w = ui.Window.getCurrent()
05 LET c = w.findNode("Canvas","canvas01")
```

Create a child node with a specific type defining the shape:

```
06 LET s = c.createChild("CanvasLine")
```

Set attributes to complete the shape definition:

```
07 CALL s.setAttribute( "fillColor", "red" )
08 CALL s.setAttribute( "startX", 10 )
09 CALL s.setAttribute( "startY", 20 )
10 CALL s.setAttribute( "endX", 100 )
11 CALL s.setAttribute( "endY", 150 )
12 CALL s.setAttribute( "width", 2 )
```

To clear a given shape in the canvas, remove the element in the canvas node:

```
13 CALL c.removeChild(s)
```

To clear the drawing area completely, remove all children of the canvas node:

```
14 LET s=c.getFirstChild()
15 WHILE s IS NOT NULL
16   CALL c.removeChild(s)
17   LET s=c.getFirstChild()
18 END WHILE
```

Functions

The following table describes the built-in functions provided for backward compatibility with version 3. This list is provided to let you search for existing code using these functions. You should review that code and use the technique described in the sections above.

Name	Description
`drawInit()`	Initializes the drawing API. It is mandatory to call this function at the beginning of your program, before the first display instruction.
`drawSelect()`	Selects a canvas area for drawing.
`drawDisableColorLines()`	By default shapes are paint with borders. This function enables/disables border drawing.
`drawLineWidth()`	Defines the width of lines.
`drawAnchor()`	Defines the anchor hint for texts.
`drawLine()`	Draws a line in the selected canvas.
`drawCircle()`	Draws a circle in the selected canvas.
`drawArc()`	Draws an arc in the selected canvas.
`drawRectangle()`	Draws a rectangle in the selected canvas.
`drawOval()`	Draws an oval in the selected canvas.
`drawText()`	Draws a text in the selected canvas.
`drawPolygon()`	Draws a polygon in the selected canvas.
`drawClear()`	Clears the selected canvas.
`drawButtonLeft()`	Enables left mouse click on a canvas element.
`drawButtonRight()`	Enables right mouse click on a canvas element.
`drawClearButton()`	Disables all mouse clicks on a canvas element.

Message Files

Summary:

- Basics
- Syntax
- Compiling Message Files
- Using Messages Files
- Example

See also: OPTIONS, SHOWHELP(), fglmkmsg, Localized Strings.

Basics

Message Files define text messages with a unique integer identifier. You can create as many message files as needed. Message files are typically used to implement application help system, especially when using the Text User Interface mode.

In order to use a message file, you need to do the following:

1. Create the source message file with a text editor.
2. Compiler the source message file to a binary format.
3. Copy the binary file to a distribution directory.
4. In programs, specify the message file with OPTIONS HELP FILE.

Message files are supported for backward compatibility. You should also have a look at the Localized Strings feature.

Syntax

```
filename.msg
```

Notes:

1. *filename* is the name of the message source file.

Syntax of a message file:

```
{
  message-definition
| include-directive
}
[...]
```

where *message-definition* is:

```
.message-number
message-line
[...]
```

where *include-directive* is:

```
.include file-name
```

Warning: Multi-line messages will include the new-line (ASCII 10) characters.

Compiling Message Files

In order to use message files in a program, the message source files (`.msg`) must be compiled with the fglmkmsg utility to produce compiled message files (`.iem`).

The following command line compiles the message source file mess01.msg:

```
fglmkmsg mess01.msg
```

This creates the compiled message file mess01.iem.

For backward compatibility, you can specify the output file as second argument:

```
fglmkmsg mess01.msg mess01.iem
```

Warning: The `.iem` compiled version of the message file must be distributed on the machine where the programs are executed.

Using Message Files

In order to use compiled message files (`.iem`) in programs, you must first specify the message file with the OPTIONS HELP FILE command:

```
01  OPTIONS HELP FILE "mymessages.iem"
```

The message file will first be searched with the string passed to the `OPTIONS HELP FILE` command (i.e. the current directory if the file is specified without a path), and if not found, the DBPATH environment variable with be used.

After the message file is defined, you can use a specific message with the either with the `HELP` keyword in a dialog instruction like INPUT:

```
01  INPUT BY NAME ... HELP 455
```

... or with the `SHOWHELP()` function:

```
01  CALL showhelp(1242)
```

Example

Message source file example:

```
01 .101
02 This is help about option 1
03 .102
04 This is help about help
05 .103
06 This is help about My Menu
```

Application using this help message:

```
01 MAIN
02     OPTIONS
03         HELP FILE "help.iem"
04     OPEN WINDOW w1 AT 5,5 WITH FORM "const"
05     MENU "My Menu"
06         COMMAND "Option 1" HELP 101
07             DISPLAY "Option 1 chosen"
08         COMMAND "Help"
09             CALL SHOWHELP(103)
10     END MENU
11     CLOSE WINDOW w1
12 END MAIN
```

MDI Windows

Summary:

- Purpose
- Usage
  - Configuring the parent container
  - Configuring child programs

See also Interface built-in class.

Purpose

By default, BDL program windows are displayed independently in separate windows on the front-end window manager. This mode is well known as SDI, Single Document Interface. The user interface can be configured to group program windows in a parent container (also known as MDI, Multiple Document Interface).

In BDL, Multiple Document Interface is called **WCI**: *Window Container Interface*.

The Window Container Interface (WCI) can be used to group several programs together in a parent window. The parent program is the container for the other programs, defined as children of the container. The container program can have its own windows, but this makes sense only for temporary modal windows (with style="dialog").

Usage

WCI configuration is done dynamically at the beginning of programs, by using the ui.Interface built-in class.

Configuring the parent container

The WCI container program is a separate BDL program of a special type, dedicated to contain other program windows. On the front-end, container programs automatically display a parent window that will hold all child program windows that will attach to the container.

The WCI container program must indicate that its type is special (setType method), and must identify itself (setName method):

```
01 MAIN
02   CALL ui.Interface.setName("parent1")
03   CALL ui.Interface.setType("container")
04   CALL ui.Interface.setText("SoftStore Manager")
```

```
05   CALL ui.Interface.loadStartMenu("mystartmenu")
06   MENU "Main"
07     COMMAND "Help" CALL help()
08     COMMAND "About" CALL aboutbox()
09     COMMAND "Exit" EXIT MENU
10   END MENU
11 END MAIN
```

Configuring child programs

WCI children programs must attach to a parent container by giving the name of the container program:

```
01 MAIN
02  CALL ui.Interface.setName("custapp")
03  CALL ui.Interface.setType("child")
04  CALL ui.Interface.setText("Customers")
05  CALL ui.Interface.setContainer("parent1")
06     ...
07 END MAIN
```

Multiple container programs can be used to group programs by application modules.

When the program is identified as a container (`type="container"`), a global window is automatically displayed as an container window. The default Toolbar and the default Topmenu are displayed and a Startmenu can be used. Other windows created by this kind of program can be displayed, inside the container (`windowType="normal"`) or as dialog windows (`windowType="modal"`). Window styles can be applied to the parent window by using the default style specification (`name="Window.main"`).

The client shows a system error and the programs stops when:

- A child program is started, but the parent container is not
- A container program is started twice

When the parent container program is stopped, other applications are automatically stopped.

The WCI container program can query for the existence of children with the `getChildCount` and `getChildInstances` methods:

```
01 MAIN
02   CALL ui.Interface.setName("parent1")
03   CALL ui.Interface.setType("container")
04   CALL ui.Interface.setText("SoftStore Manager")
05   CALL ui.Interface.loadStartMenu("mystartmenu")
06   MENU "Main"
07     COMMAND "Help" CALL help()
08     COMMAND "About" CALL aboutbox()
09     COMMAND "Exit"
10       IF ui.Interface.getChildCount()>0 THEN
11          ERROR "You must first exit the child programs."
```

```
12       ELSE
13          EXIT MENU
14       END IF
15   END MENU
16 END MAIN
```

Front End Functions

Summary:

- Basics
- The frontCall method
- Standard built-in front end functions

See also: Interface built-in class

Basics

The BDL language provides a specific method to call functions defined in the front end that will be executed locally on the workstation where the front end resides. When you call a user function from BDL, you specify a module name and a function name. Input and output parameters can be passed/returned in order to transmit/receive values to/from the front end. A typical example is an "open file" dialog window that allows you to select a file from the front end workstation file system.

A set of standard front end functions is built-in by default in the front end. It is possible to write your own functions in order to extend the front end possibilities. For example, you can write a set of functions to wrap an existing API, such as Window DDE or OLE. A set of user functions is defined in a module, implemented as a Windows DLL or UNIX shared library. These modules are loaded automatically according to the module name. See the front end documentation for more details about creating user function modules in the front end.

The frontCall method

Purpose:

The Interface built-in class provides a special method to perform front end calls.

Syntax:

```
ui.Interface.frontCall( module, function, parameter-list, returning-list )
```

Notes:

1. *module* is a string defining the name of the module (DLL or shared lib) where the function is defined. When you specify "standard", it references built-in functions provided by default by the front end.
2. *function* is a string defining the name of the function to be called.

3. *parameter-list* is a list of input parameters.
4. *returning-list* is a list of output parameters.

Usage:

Module and function names are case-sensitive.

Input and output parameters are provided as a variable list of parameters, with the square braces notation (`[param1,param2,...]`). Input parameters can be an expression supported by the language, while all output parameters must be variables only, to receive the returning values. An empty list is specified with `[]`.

Errors:

If the function could not be executed properly, one of the following exceptions might occur:

Error number	Description
-6331	The module specified could not be found.
-6332	The function specified could not be found.
-6333	The function call failed (fatal error in function).
-6334	A function call stack problem occurred (usually, wrong number of parameters provided).

Example:

```
01 MAIN
02   DEFINE data STRING
03   CALL ui.Interface.frontCall( "mymodule", "connect",
["client1",128], [] )
04   LET data = "Hello!"
05   CALL ui.Interface.frontCall( "mymodule", "send", [data,
data.getLength()], [] )
06   CALL ui.Interface.frontCall( "mymodule", "receive", [], [data] )
07   DISPLAY data
08   CALL ui.Interface.frontCall( "mymodule", "disconnect",
["client1"], [] )
09 END MAIN
```

Standard built-in front end functions

The following table shows the built-in functions implemented by the front ends in the "standard" module. Note that most of these functions are supported by desktop front-ends (GDC/GJC) only.

Warning: Additional modules and functions are available for front-ends, but are specific to the front-end type. Please refer to the front-end documentation for more details.

<table>
<tr><th>Function Name</th><th>Front-ends</th><th>Description</th></tr>
<tr><td>execute</td><td>GDC, GJC</td><td>Executes a command on the workstation with or without waiting.
Parameters:
- The command to be executed.
- The wait option (1=wait, 0=do not wait).
Returns:
- The execution result (TRUE=success, FALSE=error).</td></tr>
<tr><td>feinfo</td><td>GDC, GJC, GWC</td><td>Returns front end properties like the front end type, the workstation operating system type.
Parameters:
- The name of the property.
Returns:
- The value of the property.

<table><tr><th>Property name</th><th>Description</th></tr><tr><td>fename</td><td>The name of the front end.</td></tr><tr><td>ostype</td><td>The operating system type (UNIX, WINDOWS, MACOSX).</td></tr></table></td></tr>
<tr><td>shellexec</td><td>GDC, GJC</td><td>Opens a file on the workstation with the program associated to the file extension.
Parameters:
- The document file to be opened.
Windows Only!: - the action to perform, related to the way the file type is registred in Windows Registry (Optionnal).
Returns:
- The execution result (TRUE=success, FALSE=error)
Warning!: Under X11 Systems, this uses kde's kfmclient tool, which needs to be installed on your system.</td></tr>
<tr><td>getenv</td><td>GDC, GJC</td><td>Returns an environment variable set in the user session on the front end workstation.
Parameters:
- The name of the environment variable.
Returns:
- The value of the environment variable.</td></tr>
<tr><td>opendir</td><td>GDC, GJC</td><td>Displays a file dialog window to get a directory path on the local file system.</td></tr>
</table>

		Parameters: - The default path. - The caption to be displayed. Returns: - The name of the selected directory (or NULL if canceled).
openfile	GDC, GJC	Displays a file dialog window to get a path to open a file on the local file system. Parameters: - The default path. - The name to be displayed for the file type. - The file types (as a blank separated list of extensions). - The caption to be displayed. Returns: - The name of the selected file (or NULL if canceled).
savefile	GDC, GJC	Displays a file dialog window to get a path to save a file on the local file system. Parameters: - The default path. - The name to be displayed for the file type. - The file types (as a blank separated list of extensions). - The caption to be displayed. Returns: - The name of the selected file (or NULL if canceled).
cbclear	GDC, GJC	Clears the content of the clipboard. Parameters: none. Returns: - The execution result (TRUE=success, FALSE=error).
cbset	GDC, GJC	Set the content of the clipboard. Parameters: - The text to be set. Returns: - The execution result (TRUE=success, FALSE=error).
cbget	GDC, GJC	Gets the content of the clipboard. Parameters: none. Returns: - The text in the clipboard.
cbadd	GDC, GJC	Adds to the content of the clipboard. Parameters: - The text to be added. Returns: - The execution result (TRUE=success, FALSE=error).

cbpaste	GDC, GJC	Pastes the content of the clipboard to the current field. Parameters: none. Returns: - The execution result (TRUE=success, FALSE=error).
mdclose	GDC, GJC	Unloads a DLL or shared library module. Parameters: - The name of the module. Returns: - 0 = success, -1 = module not found, -2 = cannot unload (busy).

Front End Protocol

This page describes the **Front End Protocol**: the communication between the Runtime System and the Front End.

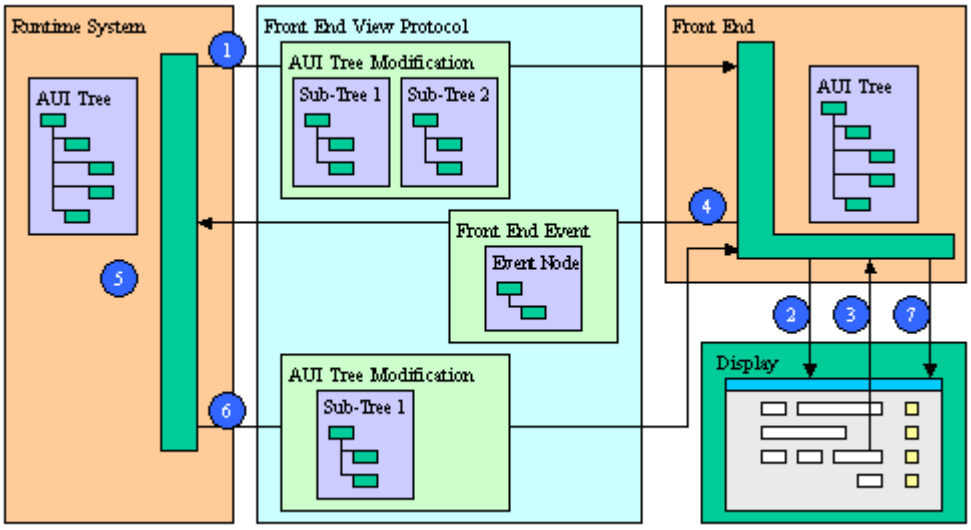
- Definition of the Front End Protocol
- Runtime System Commands
- Front End Events
- Communication Initialization
- Protocol compression and encapsulation

Definition of Front End Protocol

The purpose of the **Front End Protocol** is to synchronize the **Abstract User Interface (AUI) Tree** maintained by the Runtime System and the corresponding copy held by the Front End. For more details about these concepts, see the Dynamic User Interface.

The **AUI Tree** is used by the Front End to create graphical objects. The Front End and the Runtime System have the same version of the AUI Tree. This way, communications correspond to AUI Tree synchronization operations: on one hand the Front End sends **modification requests** to the Runtime System (also called Front End Events); on the other hand, the Runtime System analyses and validates Front End requests, performs some codes if required, and sends back **modification orders**.

The following schema describes typical communication between the Runtime System and the Front End:



1. Initialization phase: The Runtime System sends the initial AUI Tree.
2. The Front End builds the Graphical User Interface according to the AUI Tree.

3. The Front End waits for a user interaction (mouse click, keyboard typing).
4. When the user performs some interaction, the Front End sends Front End Events corresponding to the modifications made by the user.
5. Front End Events are analyzed and validated by the runtime system.
6. The Runtime System sends back the result of the Front End requests, by the way of AUI Tree Modifications Commands.
7. When receiving these commands, the Front End modifies its version of the AUI Tree and updates the Graphical User Interface. It then waits for new user interactions (step 3).

String Literals

In the Front End Protocol, the type of all attributes is string. The value of the attributes follows the same rules as 4gl string literals:

- values are enclosed between double quotes : `"`
- CR character is `\n`
- TAB character is `\t`
- double quotes character is `\"`

Example:

```
01   { GroupBox 25 {  { text "this is a \"GroupBox\""} } {}}
```

Runtime System Commands

The Runtime System sends commands to the Front End in order to modify the User Interface. As discussed earlier, these commands are modifications of the AUI Tree. Modifications can be:

- Adding children to a node
- Changing node attributes
- Removing a node

Syntax:

```
om command-id
{
  [ { { appendNodeCommand | updateNodeCommand | removeNodeCommand } } ]
[...]
}
```

Notes:

- *command-id* is the number of the command. This number is automatically increased for each command sent by the application to the Front End. The very first *appendNodeCommand* used to initialize the Protocol is command 0.

Warning: There is no verification made about this order. The communication wire is supposed to be reliable and the Runner and the Front End do not perform verification about lost commands.

Append Node Command

The **an** command adds one or several children and their attributes to a specified node. Several children (and sub-children) and can be added in the same **an** command. This command is sent by the Runtime System when there are new graphical objects to display, and to initialize communication.

Syntax:

```
an parent-id new-node
```

Where *new-node* is :

```
tagName new-id { [ attribute-list ] } { [ child-list ] } }
```

Where *attribute-list* is :

```
{ attribute-name "attribute-value" } [...]
```

Where *children-list* is :

```
{ new-node } [...]
```

Notes:

1. *parent-id* identifies the existing node.
2. *tag-name* identifies the type of the added child. The list of possible children for each node is defined in the AUI Tree.
3. *new-id* is a unique id for the new node created.
4. *attribute-name* is the name of the attribute of the node.
5. *attribute-value* is the value of the attribute.

Example:

This example shows an `an` command that creates a Menu (a Menu node is added) :

```
01 an 0 Menu 356 { { active "1"} { text "MAIN"} { posY "0"} { selection
"357"} }
02 {
03   { MenuAction 357 { { name "Option1"} { text "Option1"} { comment
""} } {}}
04   { MenuAction 358 { { name "Flow"} { text "Flow"} { comment ""} }
{}}
05   { MenuAction 359 { { name "Window"} { text "Window"} { comment
"OPEN WINDOW"} } {}}
06   { MenuAction 360 { { name "Form"} { text "Form"} { comment "form:
scroll, erase..."} } {}}
07   { MenuAction 361 { { name "Dialog"} { text "Dialog"} { comment ""}
} {}}
08   { MenuAction 362 { { name "Display"} { text "Display"} { comment
""} } {}}
09   { MenuAction 363 { { name "Options"} { text "Options"} { comment
"OPTIONS"} } {}}
10   { MenuAction 364 { { name "Exit"} { text "Exit"} { comment ""} }
{}}
11 }
```

Remove Node Command

The **rn** command removes a specific node. This command is used when graphical objects are no longer required and need to be removed from the User Interface.

Syntax:

```
rn node-id
```

Notes:

1. *node-id* identifies the existing node to be deleted

Example:

This example shows a `rn` command that removes a node from the AUI Tree; in this example, the node removed would be a MenuAction node created by the `an` command in the previous example.

```
01 rn 357
```

Update Node Command

The **un** command modifies some attributes of a specific node. This command is used to modify the aspect of a widget, for example to validate the value entered by a user in a

form field. This command is also used to confirm a focus change, modifying the `focus` attribute of the UserInterface node.

Syntax:

```
un node-id { [ attribute-list ] }
```

Where *attribute-list* is :

```
{ attribute-name "attribute-value" } [...]
```

Notes:

1. *node-id* identifies the modified node.
2. *attribute-name* is the name of the attribute of the node.
3. *attribute-value* is the value of the attribute.

Example:

This example shows an `un` command confirming a focus change: the focus now goes to the Menu option identified by id "358", created by the `an` command described in the example below. The UserInterface node has always an id equal to 0 (zero).

```
01 un 0 { focus "358" }
```

Front End Events

The Front End sends "modification requests" represented as "Front End Events" to the Runtime system. A group of modification requests can be sent in the same `event _om` command.

These events can be:

- events associated to any defined action (`ActionEvent`). This type of event is sent if a user invokes an enabled Action: Action within Dialog, MenuAction within Menu or StartMenuCommand within StartMenu.
- events associated to closing the current window (`CloseWindowEvent`). This type of event is sent if a user wants to close the current Window.
- events associated to modifications of the User Interface (`ConfigureEvent`). This type of event is sent if a user modifies something in the User Interface. Typically, this is used for focus changes or when data is entered in a form field.
- events associated to Keyboard action which can not be handled by any other event (`KeyEvent`). This type of event is sent to notify the Runtime System that a user has pressed one of the following keys : `tab`, `shift+tab`, `key_up`, `key_down`, `page_up`, `page_down`.

- events associated to local functions (`FunctionCallEvent`). This type of event is sent when a local function is over, to sent back the result of this function. Typically, local functions are DDE Functions, winexec.
- events sent by the Front End to terminate an application (`DestroyEvent`). This type of event is sent when there is an error on the Front End side that needs that the application terminates.

A very basic Front End needs only to handle `KeyEvent` events, and can send all keys pressed by the user to the Runtime System. For performance and more enhancements, most of the key pressed events are handled locally by the Front End. Only the keys mentioned above are sent.

Syntax:

```
event _om command-id {}
{
  { { ConfigureEvent 0 { { idRef "object-id" } attribute-list  } }
  | { KeyEvent 0 { { keyName "key-value" } } }
  | { ActionEvent 0 { { idRef  "object-id" } } }
  | { FunctionCallEvent 0 { { result "result-value" } } }
  | { DestroyEvent 0 { { status "status-value" } { message "message-
value" } } }
  } [...]
}
```

Where *attribute-list* is :

```
{ attribute-name "attribute-value" } [...]
```

Notes:

1. *command-id* is the number of the event. This number is increased automatically for each `event _om` command sent by the Front End to Runtime System.
2. *object-id* is the id of the node which is concerned by the event. Typically, this is the id of the object which has been changed by the user, such as a form field.
3. *attribute-name* is the name of the attribute of the node.
4. *attribute-value* is the value of the attribute.
5. *key-value* is the value of the key pressed.
6. *result-value* is the value returned by the local function, after it completes execution.
7. *status-value* is the error identifier that causes the `DestroyEvent`.
8. *message-value* is the error message explaining the reason of the `DestroyEvent`.

Example:

This example shows an `event _om` command corresponding to the following interaction:

- the user enters a value into a field. A `ConfigureEvent` with the new value is sent.

- the user click with the mouse in another field. A `ConfigureEvent` with the position of the cursor in the new field is sent.

```
01 event _om 3 {}
02  {
03    { ConfigureEvent 0 { { idRef "35" } { value "someText" } { cursor
"4" } } }
04    { ConfigureEvent 0 { { idRef "32" } { cursor "6" } } }
05  }
```

Communication Initialization

Communication is initiated by the Runtime System, which sends some meta information to the Front End. The meta information sent is "`encoding`". The Front End replies with some information, to include the version of the Front End. With communication initialized, the Runtime System sends the first version of the AUI Tree, generated according to the interactive elements used in the program (see Menus, Windows and Forms).

The root node of the Tree is the UserInterface node. This node is sent once to the Front End. The append node command (`an`) is used to create the root node with an id of zero ('0'). The append node command is then used to add all the children needed by the Front End to build the initial Graphical User Interface.

DVM meta message syntax:

```
DVM :   meta Connection {
              { encoding "character-set" }
              { protocolVersion "protocol-version" }
              { interfaceVersion "interface-version" }
              { runtimeVersion "runtime-version" }
              { compression "zlib|none" }
              { encapsulation "0|1" }
        }
```

Notes:

1. *character-set* defines the encoding character code set used in the protocol.
2. The compression attribute defines the type of compression used. The compression method can be "zlib" or "none". When "zlib" is used the encapsulation must be enabled. If this attribute is not set the default value is "none". When the DVM send this attribute it is a request. The compression is only used once the front-end validates this request.
3. *protocol-version* defines the version of the protocol (commands).
4. *interface-version* defines the version of the user interface (nodes and attributes).
5. *runtime-version* defines the version of runtime system (VM).

Front-end meta message syntax:

```
FE :    meta Client {
              { name "client-type" }
              { version "client-version" }
              { host "hostname" }
              { port "tcpport" }
              { connections "count" }
              { frontEndID2 "frontend-ID2" }
              { compression "zlib|none" }
              { encapsulation "0|1" }
              { filetransfer "0|1" }
        }
```

Notes:

1. *client-type* is the type of the front end. Can be GDC, GJC, and so on.
2. *client-version* is the version of the Front End.
3. *hostname* is the network address (alphanumeric value) of the computer hosting the Front End.
4. *tcpport* is the network port number used by the connection.
5. *count* is the number of connections established with the front-end.
6. *frontend-ID2* identifies the front-end for authentication rules.

Protocol compression and encapsulation

Encapsulation is always active in the GUI protocol, to send packets by blocks over the network and allow secondary protocols over the channel, such as file transfer between the front-end and the application server.

When possible, the runtime system and the client try to compress the data exchanges in the GUI communication. The compression algorithm is provided by the standard ZLIB library of the system. The ZLIB library might not be installed on the system, especially on Windows platforms. If the runtime system is not able to find the standard ZLIB library of the system, it loads the fallback library provided in FGLDIR. This secondary library is located in $FGLDIR/lib/libzfgl.so on UNIX and %FGLDIR%\bin\LIBZFGL.DLL on Windows.

Notes:

1. By default if encapsulation / compression is enabled, the GUI communication is not pretty printed. To properly see the GUI communication in a front-end log window, set the environment variable FGLGUIDEBUG to 1.

The Application class

Summary:

- Syntax
- Methods
- Usage
  - Command line arguments
  - Program information
  - Runtime information
  - FGLPROFILE resource
  - Debugging

See also: Built-in classes

Syntax

The **Application** class is a built-in class providing an interface to the application internals.

Syntax:

```
base.Application
```

Notes:

1. This class does not have to be instantiated; it provides class methods for the current program.

Methods:

Class Methods	
Name	**Description**
`getArgumentCount()` `  RETURNING INTEGER`	Returns the number of arguments passed to the program.
`getArgument( position INTEGER )` `  RETURNING STRING`	Returns the argument passed to the program, according to its position.
`getProgramName()` `  RETURNING STRING`	Returns the name of the program.
`getProgramDir()` `  RETURNING STRING`	Returns the system-dependent path of the directory where the program files are located.
`getFglDir()`	Returns the system-dependent path

`RETURNING STRING`	of the installation directory of the runtime system (FGLDIR environment variable).
`getResourceEntry( name STRING ) RETURNING STRING`	Returns the value of an FGLPROFILE entry.
`getStackTrace() RETURNING STRING`	Returns the current stack trace of the program flow.

Usage:

The Application class groups a set of utility functions related to the program environment. Command line arguments, execution directory and FGLPROFILE resource entries are some of the elements you can query with this class.

Command line arguments

You can query command line arguments with the `getArgumentCount()` and `getArgument()` methods. The `getArgumentCount()` method returns the total number of arguments passed to the program, while `getArgument()` returns the argument value for the given position. First argument starts at 1 (argument number zero is the program name).

```
01 MAIN
02    DEFINE i INTEGER
03    FOR i=1 TO base.Application.getArgumentCount()
04       DISPLAY base.Application.getArgument(i)
05    END FOR
06 END MAIN
```

Program information

Basic program execution information can be queried with the `getProgramName()` and `getProgramDir()` methods. The `getProgramName()` method returns the name of the program. The `getProgramDir()` method returns the directory path where the **42r** program file is located. Note that the directory path is system-dependent.

Runtime information

Product information can be queried with the `getFglDir()` method. The `getFglDir()` method returns the installation directory path defined by FGLDIR. Note that the directory path is system-dependent.

FGLPROFILE resource

If needed you can query FGLPROFILE resource entries with the `getResourceEntry()` method. This method returns the fglprofile value of the entry passed as parameter.

```
01 MAIN
02   DISPLAY
base.Application.getResourceEntry("mycompany.params.logmode")
03 END MAIN
```

Debugging

In some situations - typically, to identify problems on a production site - you may want to known what functions have been called when a program raises an error. You can get and print the stack trace in a log file by using the `getStackTrace()` method. This method returns a string containing a formatted list of the current function stack.

You typically use this function in a WHENEVER ERROR CALL handler, as in the following code example:

```
01 MAIN
02    WHENEVER ERROR CALL my_handler
03    ...
04 END MAIN
05 ...
06 FUNCTION my_handler()
07    DISPLAY base.Application.getStackTrace()
08 END FUNCTION
```

Example of stack trace output:

```
#0 my_handler() at debug.4gl:173
#1 save_customer_data() at customer.4gl:1534
#2 edit_customer() at customer.4gl:542
#3 main at main.4gl:23
```

The Channel class

Summary:

- Syntax
- Methods
- Usage
  - Creating a Channel object
  - Setting the value delimiter
  - Opening a Channel
    - Opening a file Channel
    - Opening a pipe Channel
    - Opening a client socket Channel
  - Reading from / Writing to a Channel
  - Closing a Channel
  - Exception handling
  - Detecting end of file
  - Reading and Writing complete lines
  - Managing line terminators with read() / write()
  - Managing line terminators with readLine() / writeLine()
  - Line terminators on Windows platforms
- Examples
  - Example 1: Reading formatted data from a file
  - Example 2: Executing the ls UNIX command
  - Example 3: Reading lines from a file
  - Example 4: Communicating with an HTTP server

See also: Built-in classes

Syntax

The **Channel** class provides basic read/write functionality for access to files or communication with sub-processes.

Syntax:

```
base.Channel
```

Methods:

Class Methods	
Name	**Description**
`create() RETURNING base.Channel`	Creates a new Channel object.
Object Methods	

Name	Description
`isEof() RETURNING INTEGER`	Returns TRUE if end of file is reached.
`openFile( path STRING, flags STRING )`	Opens a Channel to a file identified by *path*, with options.
`openPipe( scmd STRING, flags STRING )`	Opens a Channel to a process by executing the command *scmd*, with options.
`openClientSocket( host STRING, port INTEGER, flags STRING, timeout INTEGER )`	Opens a Channel to a socket server identified by *host* and *port*, with options.
`setDelimiter( d STRING )`	Sets the field delimiter of the Channel.
`read( buffer-list ) RETURNING INTEGER`	Reads data from the input.
`write( buffer-list )`	Writes data to the output.
`readLine() RETURNING STRING`	Reads a complete line of data from the Channel and returns the string.
`writeLine( buffer STRING )`	Writes a complete line of data to the Channel.
`close()`	Closes the Channel.

Usage:

The Channel class is a built-in class that provides basic read/write functionality for accessing files or communicating with sub-processes.

Warning: As with other BDL instructions, when you are reading or writing strings the escape character is the backslash "\".

Creating a Channel object

First you must declare a `base.Channel` variable; then, create a Channel object and optionally set the field delimiter:

```
01 DEFINE ch base.Channel
02 LET ch = base.Channel.create()
```

Setting the value delimiter

After creating the Channel object, you typical set the field value delimiter with:

```
01 CALL ch.setDelimiter("^")
```

The default is DBDELIMITER, or `"|"` if DBDELIMITER is not defined. If you pass NULL, no delimiter is used.

Opening a Channel

Opening a file Channel

You can open a file for reading, writing, or both, by using the `openFile()` method:

```
01 CALL ch.openFile( "file.txt", "w" )
```

The parameters for this method are:

1. The path to the file
2. The opening flags

The opening *flags* can be one of:

- `r` : For Read mode: reads from a file (standard input if *path* is NULL).
- `w` : For Write mode: starts with an empty file (standard output if the *path* is NULL).
- `a` : For Append mode: writes at the end of a file (standard output if the *path* is NULL).
- `u` : For Read from standard input and Write to standard output (*path* must be NULL).

Any of the above options can be followed by:

- `b` : Open in binary mode, to avoid CR/LF (carriage return/line feed) translation.

When you use the `"w"` or `"a"` modes, the file is created if it does not exist.

Opening a pipe Channel

With the `openPipe()` method, you can read from the standard output of a sub-process, write to the standard input, or both.

```
01 CALL ch.openPipe( "ls", "r" )
```

The parameters for this method are:

1. The command to be executed
2. The opening flags

The opening *flags* can be one of:

- `r` : For Read Only from standard output of the command
- `w` : For Write Only to standard input of the command
- `a` : For Write Only to standard input of the command
- `u` : For Read from standard output and Write to standard input of the command

Opening a client socket Channel

Use the `openClientSocket()` method to establish a TCP connection to a server.

Warning: The network protocol must be based on ASCII, or must use the same character set as the application.

Example:

```
01 CALL ch.openClientSocket( "localhost", 80, "ub", 5 )
```

The parameters for this method are:

1. The name of the host machine you want to connect to
2. The port number of the service
3. The opening flags
4. The timeout in seconds. -1 indicates no timeout (wait forever)

The *timeout* is specified in seconds, -1 waits forever.

The opening *flags* can be one of:

- `r` : For Read mode: only to read from the socket
- `w` : For Write mode: only to write to the socket
- `u` : For Read and Write mode: To read and write from/to the socket

Any of the above options can be followed by:

- `b` : Open in binary mode, to avoid CR/LF (carriage return / line feed) translation.

Reading from / Writing to a Channel

When the Channel is open, you can read and/or write data from/to the input/output. You must provide a variable list by using the the square brace notation (`[param1,param2,...]`). The read function returns TRUE if data could be read.

```
01 DEFINE a,b INTEGER
02 DEFINE c,d CHAR(20)
03 WHILE ch1.read([a,b,c,d])
04   CALL ch2.write([a,b,c,d])
05 END WHILE
```

Closing a Channel

When you have finished with the Channel, close it with the `close()` method:

```
01 CALL ch.close()
```

A Channel is automatically closed when the last reference to the Channel object is deleted.

Exception handling

You can trap exceptions with the standard `WHENEVER ERROR` exception handler:

```
01 WHENEVER ERROR CONTINUE
02 CALL ch.write([num,label])
03 IF STATUS THEN
04    ERROR "An error occurred while reading from Channel"
05    CALL ch.close()
06    RETURN -1
07 END IF
08 WHENEVER ERROR STOP
```

Detecting end of file

To detect the end of a file while reading from a Channel, you can use the `isEof()` method:

```
01 DEFINE s STRING
02 WHILE TRUE
03    LET s = ch.readLine()
04    IF ch.isEof() THEN EXIT WHILE END IF
05    DISPLAY s
06 END WHILE
```

Warning: The End Of File can only be detected after the last read (first read, then check EOF and process if not EOF) .

Reading and writing complete lines

If the stream does not contain lines with fields (and field separators), you should use the `readLine()` and `writeLine()` methods. These methods read/write a complete line from/to the Channel, by ignoring the delimiter defined by `setDelimiter()`:

```
01 DEFINE buff STRING
02 CALL ch.writeLine("this is a complete line")
03 LET buff = ch.readLine()
```

The `readLine()` method must be used if the source stream does not contain lines with field separators.

The `readLine()` method returns an empty string if the line is empty.

Warning: The readLine() function returns NULL if end of file is reached. To distinguish empty lines from NULL, you must use the STRING data type. If you use a CHAR or VARCHAR, you will get NULL for empty lines. To properly detect end of file, you can use the isEof() method.

Managing line terminators with read() and write()

When using the `read()`/`write()` functions, the escaped line-feed (LF, \n) characters are written as BS + LF to the output. When reading data, BS + LF are detected and interpreted, to be restituted as if the value was assigned by a LET instruction, with the same string used in the `write()` function.

If you want to write a LF as part of a value, the string must contain the backslash and line-feed as two independent characters. You need to escape the backslash when you write the string constant in the BDL source file.

In the following code example, an empty delimiter is used to simplify explanation:

```
01 CALL ch.setDelimiter("")
02 CALL ch.write("aaa\\\nbbb")  -- [aaa<bs><lf>bbb]
03 CALL ch.write("ccc\nddd")    -- [aaa<lf>bbb]
```

... would generate the following output:

```
01 aaa\
02 bbb
03 ccc
04 ddd
```

where line `01` and `02` contain data for the same line, in the meaning of a Channel record.

When you read these lines back with a `read()` call, you get the following strings in memory:

```
Read 1 aaa<bs><lf>bbb
Read 2 ccc
Read 3 ddd
```

These reads would correspond to the following assignments when using string constants:

```
01 LET s = "aaa\\\nbbb"
02 LET s = "ccc"
03 LET s = "ddd"
```

Managing line terminators with readLine() and writeLine()

When using the `readLine()` and `writeLine()` functions, a LF character represents the end of a line.

Warning: LF characters escaped by a backslash are not interpreted as part of the line during a readLine() call.

When a line is written, any LF characters in the string will be written as is to the output. When a line is read, the LF escaped by a backslash is not interpreted as part of the line.

For example, the following code:

```
01 CALL ch.writeLine("aaa\\\nbbb")  -- [aaa<bs><lf>bbb]
02 CALL ch.writeLine("ccc\nddd")    -- [aaa<lf>bbb]
```

... would generate the following output:

```
01 aaa\
02 bbb
03 ccc
04 ddd
```

... and the subsequent `readLine()` will read four different lines, where the first line would be ended by a backslash:

```
Read 1 aaa<bs>
Read 2 bbb
Read 3 ccc
Read 4 ddd
```

Line terminators on Windows platforms

On Windows platforms, DOS formatted text files use CR/LF as line terminators. You can manage this type of files with the Channel class.

By default, on both Windows and Unix platforms, when records are read from a DOS file with the Channel class, the CR/LF line terminator is removed. When a record is written

to a file on Windows, the lines are terminated with CR/LF in the file; on UNIX, the lines are terminated with LF only.

If you want to avoid the automatic translation of CR/LF on Windows, you can use the `b` option of the `openFile()` and `openPipe()` methods. You typically combine the `b` option with `r` or `w`, based on the read or write operations that you want to do:

```
01 CALL ch.openFile( "mytext.txt", "rb" )
```

On Windows, when lines are read with the `b` option, only LF is removed from CR/LF line terminators; CR will be copied as a character part of the last field. In contrast, when lines are written with the `b` option, LF characters will not be converted to CR/LF.

On UNIX, writing lines with or without the binary mode option does not matter.

Examples

Example 1: Reading formatted data from a file

This program reads data from the "file.txt" file that contains two columns separated by a | (pipe) character, and re-writes this data at the end of the "fileout.txt" file, separated by "%"

```
01 MAIN
02     DEFINE buff1, buff2 STRING
03     DEFINE ch_in, ch_out base.Channel
04     LET ch_in = base.Channel.create()
05     CALL ch_in.setDelimiter("|")
06     LET ch_out = base.Channel.create()
07     CALL ch_out.setDelimiter("%")
08     CALL ch_in.openFile("file.txt","r")
09     CALL ch_out.openFile("fileout.txt","w")
10     WHILE ch_in.read([buff1,buff2])
11         CALL ch_out.write([buff1,buff2])
12     END WHILE
13     CALL ch_in.close()
14     CALL ch_out.close()
15 END MAIN
```

Example 2: Executing the ls UNIX command

This program executes the "ls" command and displays the filenames and extensions separately:

```
01 MAIN
02     DEFINE fn CHAR(40)
03     DEFINE ex CHAR(10)
04     DEFINE ch base.Channel
05     LET ch = base.Channel.create()
```

```
06     CALL ch.setDelimiter(".")
07     CALL ch.openPipe("ls -l","r")
08     WHILE ch.read([fn,ex])
09         DISPLAY fn, "   ", ex
10     END WHILE
11     CALL ch.close()
12 END MAIN
```

Example 3: Reading lines from a file:

```
01 MAIN
02     DEFINE i INTEGER
03     DEFINE s STRING
04     DEFINE ch base.Channel
05     LET ch = base.Channel.create()
06     CALL ch.openFile("file.txt","r")
07     LET i = 1
08     WHILE TRUE
09         LET s = ch.readLine()
10         IF ch.isEof() THEN EXIT WHILE END IF
11         DISPLAY i, " ", s
12         LET i = i + 1
13     END WHILE
14     CALL ch.close()
15 END MAIN
```

Example 4: Communicating with an HTTP server:

```
01 MAIN
02     DEFINE ch base.Channel, eof INTEGER
03     LET ch = base.Channel.create()
04     -- We open the Channel in binary mode to control CR+LF
05     CALL ch.openClientSocket("localhost",80,"ub", 30)
06     -- HTTP expects CR+LF: Note that LF is added by writeLine()!
07     CALL ch.writeLine("GET / HTTP/1.0\r")
08     -- No HTTP headers...
09     -- Empty line = end of headers
10     CALL ch.writeLine("\r")
11     WHILE NOT eof
12         DISPLAY ch.readLine()
13         LET eof = ch.isEof()
14     END WHILE
15     CALL ch.close()
16 END MAIN
```

The StringBuffer class

Summary:

- Syntax
- Methods
- Usage
  - Create a StringBuffer object
  - Append a string
  - Clear the string buffer
  - Compare strings
  - Compare strings ignoring case
  - Return the character at a specified position
  - Return the position of a substring
  - Return the length of a string
  - Returns the substring at the specified position
  - Convert string to lowercase
  - Convert string to uppercase
  - Trim the string
  - Trim the beginning of the string
  - Trim the end of the string
  - Replace part of the current string with another string
  - Replace one string with another
  - Insert a string
  - Convert buffer to a STRING value
- Examples

See also: Built-in classes

Syntax

The **StringBuffer** class is a built-in class designed to manipulate character strings.

Syntax:

```
base.StringBuffer
```

Methods:

Class Methods	
Name	**Description**
`create( )` `  RETURNING base.StringBuffer`	Creates a new empty StringBuffer object.
Object Methods	

Name	Description
`append( str STRING )`	Appends a string to the string buffer.
`clear( )`	Clears the string buffer.
`equals( src STRING ) RETURNING INTEGER`	Returns TRUE if the string passed as parameter matches the current string. If one of the strings is NULL the method returns FALSE.
`equalsIgnoreCase( src STRING ) RETURNING INTEGER`	Returns TRUE if the string passed as parameter matches the current string, ignoring character case. If one of the strings is NULL the method returns FALSE.
`getCharAt( pos INTEGER ) RETURNING STRING`	Returns the character at the byte position *pos* (starts at 1). Returns NULL if the position does not match a valid character-byte position in the current string.
`getIndexOf( str STRING, spos INTEGER ) RETURNING INTEGER`	Returns the position of the sub-string *str* in the current string, starting from byte position *spos*. Returns zero if the sub-string was not found.
`getLength( ) RETURNING INTEGER`	Returns the number of bytes of the current string, including trailing spaces.
`subString( spos INTEGER, epos INTEGER ) RETURNING STRING`	Returns the sub-string starting at the byte position *spos* and ending at *epos*. Returns NULL if the positions do not delimit a valid sub-string in the current string. First character-byte starts at 1.
`toLowerCase( )`	Converts the current string to lowercase.
`toUpperCase( )`	Converts the current string to uppercase.
`trim( )`	Removes white space characters from the beginning and end of the current string.
`trimLeft( )`	Removes white space characters from the beginning of the current string.
`trimRight( )`	Removes white space characters from the end of the current string.
`replaceAt( pos INTEGER, len INTEGER, str STRING )`	Replaces a part of the current string by another string, starting from byte position *pos* for *len* bytes. First character-byte position is 1.
`replace( oldString STRING,`	Replaces in the current string the

`newString STRING, occ INTEGER )`	STRING *oldString* by the STRING *newString*, *occ* time(s) (all occurrences, if *occ* set to 0).
`insertAt( pos INTEGER, str STRING )`	Inserts a string before the byte position *pos*. First character-byte position is 1.
`toString( ) RETURNING STRING`	Creates a STRING value from the current buffer.

Usage:

Create a StringBuffer object with the `base.StringBuffer.create()` class method.

The StringBuffer class is optimized for string operations. When using a StringBuffer object, you work directly on the internal string buffer. Use the StringBuffer class to implement heavy string manipulations. For example, if you need to process 500Kb of text (such as when you are performing a global search-and-replace of specific words), you get much better performances with a StringBuffer object than you would using a basic STRING variable.

Warning: The StringBuffer methods are all based on byte-semantics. In a multi-byte environment, the getLength() method returns the number of bytes, which can be different from the number of characters.

When you pass a StringBuffer object as function parameter, the function receives a variable that references the StringBuffer object. Passing the StringBuffer object by reference is much more efficient than using a `STRING` that is passed by value (i.e. data is copied on the stack). The function manipulates the original string, not a copy of the string as if it was a `STRING` variable.

A StringBuffer object has different semantics than a `STRING` variable: When using the `STRING` data type, the runtime system always creates a new buffer when you modify a string.

For example, when you concatenate strings with the `append()` method of a `STRING` variable, the runtime system creates a new buffer to hold the new string. This does not impact performances of programs with a user interface or even batch programs doing SQL, but can be an issue when you want to rapidly process large character strings.

The `append()` method appends a string to the string buffer.

The `clear()` method clears the string buffer.

The `equals()` method compares two strings.

The `equalsIgnoreCase()` method compares two strings ignoring case.

The `getCharAt( )` method returns the character at the specified position.

The `getIndexOf()` method returns the position of the specified substring.

The `getLength()` method returns the number of bytes in the current string.

The `subString()` method returns the substring at the specified position.

The `toLowerCase()` method converts the current string to lowercase.

The `toUpperCase()` method converts the current string to uppercase.

The `trim()` method removes white space from the beginning and end of the current string.

The `trimLeft()` method removes white space from the beginning of the current string.

The `trimRight()`method removes white space from the end of the current string.

The `replaceAt()` method freplaces part of the current string with another string.

The `replace()` method replaces the current string with another string.

The `insertAt()` method inserts a string before the specified position.

The `toString()` method creates a STRING value from the current buffer.

Examples

Example 1: Adding strings to a StringBuffer.

```
01 MAIN
02   DEFINE buf base.StringBuffer
03   LET buf = base.StringBuffer.create()
04   CALL buf.append("abc")
05   DISPLAY buf.toString()
06   CALL buf.append("def")
07   DISPLAY buf.toString()
08   CALL buf.append(123456)
09   DISPLAY buf.toString()
10 END MAIN
```

Example 2: Modifying a StringBuffer with a function.

```
01 MAIN
02   DEFINE buf base.StringBuffer
03   LET buf = base.StringBuffer.create()
```

```
04   CALL modify(buf)
05   DISPLAY buf.toString()
06 END MAIN
07
08 FUNCTION modify(sb)
09   DEFINE sb base.StringBuffer
10   CALL sb.append("more")
11 END FUNCTION
```

The StringTokenizer class

Summary:

- Syntax
- Methods
- Usage
  - Create a tokenizer
  - Extended create method
  - Count the number of tokens
  - Check if there are more tokens to return
  - Return the next token
- Examples

See also: Built-in classes, StringBuffer

Syntax

The **StringTokenizer** class is designed to parse a string to extract tokens according to delimiters.

Syntax:

```
base.StringTokenizer
```

Methods:

Class Methods	
Name	**Description**
`create( src STRING , delim STRING ) RETURNING base.StringTokenizer`	Returns a StringTokenizer object prepared to parse the *src* source string according to the *delim* delimiters. The *delim* parameter is a string that can hold one or more delimiters.
`createExt( src STRING , delim STRING, esc STRING, nulls INTEGER ) RETURNING base.StringTokenizer`	Same as create(), except for additional options. The *esc* parameter defines an escape character for the delimiter. The *nulls* parameter indicates if empty tokens are taken into account.
Object Methods	
Name	**Description**

`countTokens()` `  RETURNING INTEGER`	Returns the number of tokens left to be returned.
`hasMoreTokens()` `  RETURNING INTEGER`	Returns TRUE if there are more tokens to return.
`nextToken()` `  RETURNING STRING`	Parses the string and returns the next token.

Usage:

The StringTokenizer built-in class is provided to split a source string into tokens, according to delimiters. The following code uses the `base.StringTokenizer.create()` method to create a StringTokenizer that will generate 3 tokens with the values "aaa", "bbb", "ccc":

```
01 DEFINE tok base.StringTokenizer
02 LET tok = base.StringTokenizer.create("aaa|bbb|ccc","|")
```

The StringTokenizer can take a unique or multiple delimiters into account. A delimiter is always one character long. In the following example, 3 delimiters are used, and 4 tokens are extracted:

```
01 DEFINE tok base.StringTokenizer
02 LET tok = base.StringTokenizer.create("aaa|bbb;ccc+ddd","|+;")
```

If you create a StringTokenizer with the `base.StringTokenizer.create(`*`src`*`,`*`delim`*`)` method, the empty tokens are not taken into account, and no escape character is defined for the delimiters:

- No escape character can be used.
- The `nextToken()` method will never return NULL strings.
- In the source string, leading and trailing delimiters or the amount of delimiters between two tokens do not affect the number of tokens.

If you create a StringTokenizer with the `base.StringTokenizer.reateExt(`*`src`*`,`*`delim`*`,`*`esc`*`,`*`nulls`*`)` method, you can configure the StringTokenizer:

When passing a character to the *esc* parameter, the delimiters can be escaped in the source string.

When passing TRUE to the *nulls* parameter, the empty tokens are taken into account:

- The `nextToken()` method might return NULL strings.
- In the source string, leading and trailing delimiters or the amount of delimiters between two tokens affects the number of tokens.

Note that when you want to specify a backslash as a delimiter, you must use double backslashes in both the source string and as the delimiter, as shown in Example 3 below.

The `countTokens()` method counts the number of tokens left to be returned.

The `hasMoreTokens()` method returns TRUE if there are more tokens to return.

The `nextToken()` method parses the string and returns the next token.

Examples

Example 1: Split a UNIX directory path

```
01 MAIN
02   DEFINE tok base.StringTokenizer
03   LET tok = base.StringTokenizer.create("/home/tomy","/")
04   WHILE tok.hasMoreTokens()
05     DISPLAY tok.nextToken()
06   END WHILE
07 END MAIN
```

Example 2: Taking escaped delimiters and NULL tokens into account

```
01 MAIN
02   DEFINE tok base.StringTokenizer
03   LET tok =
base.StringTokenizer.createExt("||\\|aaa||bbc|","|","\\",TRUE)
04   WHILE tok.hasMoreTokens()
05     DISPLAY tok.nextToken()
06   END WHILE
07 END MAIN
```

Example 3: Specifying a backslash as the delimiter

```
01 MAIN
02   DEFINE tok base.StringTokenizer
03   LET tok = base.StringTokenizer.create("C:\\My Documents\\My
Pictures","\\")
04   WHILE tok.hasMoreTokens()
05     DISPLAY tok.nextToken()
06   END WHILE
07 END MAIN
```

The TypeInfo class

Summary:

- Syntax
- Methods
- Usage
    - Creating a TypeInfo object

See also: Built-in classes

Syntax

The **TypeInfo** class is a built-in class provided to serialize program variables.

Syntax:

```
base.TypeInfo
```

Notes:

1. This class does not have to be instantiated.

Methods

Class Methods	
Name	**Description**
`create( variable )` `  RETURNING om.DomNode`	Creates a DOM node from a program variable

Usage

Use the TypeInfo class to serialize program variables in an XML format. For example, you can fetch rows from a database table in an array, specify the array as the input into the `base.TypeInfo.create()` method, write the resulting DomNode to a file using the `node.writeXml()` method, and give the resulting file to any application that is able to read XML for input.

Creating a TypeInfo object

The `create()` method of this class builds a DomNode object from any kind of structured program variable, thus serializing the variable:

```
01 MAIN
02   DEFINE n om.DomNode
03   DEFINE r RECORD
04       key INTEGER,
05       lastname CHAR(20),
06       brithdate DATE
07   END RECORD
08   LET r.key = 234
09   LET r.lastname = "Johnson"
10   LET r.brithdate = MDY(12,24,1962)
11   LET n = base.TypeInfo.create( r )
12   CALL n.writeXml( "r.xml" )
13 END MAIN
```

The generated node contains variable values and data type information. The above example creates the following file:

```
<?xml version="1.0"? encoding="ISO-8859-1">
<Record>
  <Field type="INTEGER" value="234" name="key"/>
  <Field type="CHAR(20)" value="Johnson" name="lastname"/>
  <Field type="DATE" value="12/24/1962" name="birthdate"/>
</Record>
```

Note that data is formatted according to current environment settings (DBDATE, DBFORMAT, DBMONEY).

The Interface class

Summary:

- Syntax
- Methods
- Usage
- Examples
  - Getting the type and name of the front-end
  - Get the AUI root node and write it to XML file
  - Using the Windows Container Interface
  - Synchronizing the AUI tree with the front-end

See also: Built-in classes.

Syntax

The **Interface** class is a built-in class provided to manipulate the user interface.

Syntax:

```
ui.Interface
```

Notes:

1. This class does not have to be instantiated.

Methods:

Class Methods	
Name	**Description**
`frontCall( module STRING, name STRING, parameter-list, returning-list )`	Calls the front end function *name* of the module *module*. See Front End Functions for more details.
`getDocument()` `  RETURNING om.DomDocument`	Returns the DOM document owning the Abstract User Interface tree.
`getFrontEndName()` `  RETURNING STRING`	Returns the type of the front end (`'Gdc'`, `'Gwc'`, `'Gjc'`, `'Console'`).
`getFrontEndVersion()` `  RETURNING STRING`	Returns the front end version string.
`getRootNode()` `  RETURNING om.DomNode`	Returns the root DOM node of the Abstract User Interface tree.

`loadStartMenu( file STRING )`	Loads the start menu defined in an XML file into the AUI tree. See StartMenus for more details.
`loadToolBar( file STRING )`	Loads the toolbar defined in an XML file into the AUI tree. See Toolbars for more details.
`loadTopMenu( file STRING )`	Loads the topmenu defined in an XML file into the AUI tree. See TopMenus for more details.
`loadActionDefaults( file STRING )`	Loads the default decoration for actions from a specific XML file into the AUI tree. See Action Defaults for more details.
`loadStyles( file STRING )`	Loads styles defined in an XML file into the AUI tree. See Prensentation Styles for more details.
`setName( name STRING )`	Sets the name to identify the program on the front-end.
`getName()` `RETURNING STRING`	Returns the identifier of the program.
`setText( title STRING )`	Defines a title for the program.
`getText()` `RETURNING STRING`	Returns the title of the program.
`setImage( name STRING )`	Sets the name of the icon to be used for this program.
`getImage()` `RETURNING STRING`	Returns the name of the icon.
`setType( type STRING )`	Defines the type of program.
`getType()` `RETURNING STRING`	Returns the type of the program.
`setContainer( name STRING )`	Defines the name of the parent container of this program.
`getContainer()` `RETURNING STRING`	Returns the name of the parent container of this program.
`getChildCount()` `RETURNING INTEGER`	Returns the number of children in this container.
`getChildInstances( name STRING )` `RETURNING INTEGER`	Returns the number of children identified by *name*.
`refresh()`	Synchronizes the front end with the current AUI tree.

Usage

Getting the root DOM document

The `ui.Interface.getDocument()` class method returns the DomDocument object of the Abstract user Interface tree.

Getting the current front-end identifier

The `ui.Interface.getFronEndName()` class method returns the type of the front-end used by the application. This is mainly provided for debugging purposes.

Getting the current front-end version

The `ui.Interface.getFronEndVersion()` class method returns the version number of the front-end used by the application. This is mainly provided for debugging purposes.

Getting the root node of the DOM document

The `ui.Interface.getRootNode()` class method returns the root DomNode of the Abstract user Interface tree.

Defining the name of the application

The `ui.Interface.setName()` class method can be used to identify the application on the front-end. For example, this name is used in MDI configuration.

Getting the name of the application

Use the `ui.Interface.getName()` class method to get the name of the application previously set by `setName()`.

Defining the title of the application

The `ui.Interface.setText()` class method can be used to define a main title for the application on the front-end. This title is displayed in the main Window.

Getting the title of the application

Use the `ui.Interface.getText()` class method to get the title of the application previously set by `setText()`.

Defining the icon of the application

The `ui.Interface.setImage()` class method can be used to define the icon of the application on the front-end. This icon will be used in taskbars, for example.

Getting the icon of the application

Use the `ui.Interface.getImage()` class method to get the image name of the application previously set by `setImage()`.

Defining the type of the application

The `ui.Interface.setType()` class method can be used to define the type of the application, typically used in MDI configurations.

Possible values can be `'normal'`, `'container'` or `'child'`.

Getting the type of the application

Use the `ui.Interface.getType()` class method to get the type of the application, previously set by `setType()`.

Defining the parent container of the application

The parent container can be specified with the `ui.Interface.setContainer()` class method, typically used in MDI configurations.

Getting the parent container of the application

Use the `ui.Interface.getContainer()` method to get the name of the parent container of the application.

Getting the number of children in a parent container

Use the `ui.Interface.getChildCount()` class method to get the current number of child applications in this parent WCI.

See also MDI configuration.

Getting the number of child instances for a given application name

If you need to known how many child instances of the same application are started in the current WCI container, call the `ui.Interface.getChildInstances()` class method. This method takes the application name as a parameter (the one defined with setName())

See also MDI configuration.

Refreshing the user interface

Use the `ui.Interface.refresh()` class method to synchronize the server-side AUI tree with the frond-end AUI tree. For more details, see "When is the front-end synchronized?".

Examples

Example 1: Get the type and version of the front end.

```
01 MAIN
02   MENU "Test"
03     COMMAND "Get"
04       DISPLAY "Name = " || ui.Interface.getFrontEndName()
05       DISPLAY "Version = " || ui.Interface.getFrontEndVersion()
06     COMMAND "Exit"
07       EXIT MENU
08   END MENU
09 END MAIN
```

Example 2: Get the AUI root node and save it to a file in XML format.

```
01 MAIN
02   DEFINE n om.DomNode
03   MENU "Test"
04     COMMAND "SaveUI"
05       LET n = ui.Interface.getRootNode()
06       CALL n.writeXml("auitree.xml")
07     COMMAND "Exit"
08       EXIT MENU
09   END MENU
10 END MAIN
```

Example 3: Using the Window Container Interface

The WCI parent program:

```
01 MAIN
02   CALL ui.Interface.setName("main1")
03   CALL ui.Interface.setText("This is the MDI container")
04   CALL ui.Interface.setType("container")
05   CALL ui.Interface.loadStartMenu("appmenu")
06   MENU "Main"
07     COMMAND "Help" CALL help()
08     COMMAND "About" CALL aboutbox()
09     COMMAND "Exit"
10       IF ui.Interface.getChildCount()>0 THEN
11          ERROR "You must first exit the child programs."
12       ELSE
13          EXIT MENU
14       END IF
15   END MENU
16 END MAIN
```

The WCI child program:

```
01 MAIN
```

```
02   CALL ui.Interface.setName("prog1")
03   CALL ui.Interface.setText("This is module 1")
04   CALL ui.Interface.setType("child")
05   CALL ui.Interface.setContainer("main1")
06   MENU "Test"
07     COMMAND "Exit"
08       EXIT MENU
09   END MENU
10 END MAIN
```

Example 4: Synchronizing the AUI tree with the front end.

```
01 MAIN
02   DEFINE cnt INTEGER
03   OPEN WINDOW w WITH FORM "myform"
04   FOR cnt=1 TO 10
05     DISPLAY BY NAME cnt
06     CALL ui.Interface.refresh()
07     SLEEP 1
08   END FOR
09 END MAIN
```

The Window class

Summary:

- Syntax
- Methods
- Usage
  - Getting a window by name
  - Getting the current window
  - Getting the current form of a window
  - Getting the DOM node of a window
  - Search for a specific element in a window
  - Create a new empty form in a window
  - Setting the window title
  - Getting the window title
- Examples

See also: Built-in classes, Windows and Forms, Form Class

Syntax

The **Window** class is a built-in class providing an interface to the window objects.

Syntax:

```
ui.Window
```

Methods:

Class Methods	
Name	**Description**
`forName( name STRING ) RETURNING ui.Window`	Returns a Window object according to the name used in an OPEN WINDOW statement.
`getCurrent( ) RETURNING ui.Window`	Returns a Window object referencing the current window.
Object Methods	
Name	**Description**
`findNode( t STRING, n STRING ) RETURNING om.DomNode`	Returns the first descendant DOM node of type *t* and matching the name *n* in the abstract representation of this form object.
`createForm( n STRING )`	Creates an empty form and returns

`RETURNING ui.Form`	the new Form object.
`getForm( )` `RETURNING ui.Form`	Returns a Form object to handle the current form.
`getNode( )` `RETURNING om.DomNode`	Returns the DOM representation of this Window.
`setText( t STRING )`	Sets the title of this window object.
`getText( )` `RETURNING STRING`	Returns the title of this window object.

Usage:

Windows are created with the OPEN WINDOW instruction, identifying the window by a static handle:

```
01 OPEN WINDOW w1 WITH FORM "customer"
```

Getting a window object by name

You can get the window object corresponding to an identifier used in OPEN WINDOW with the `ui.Window.forName()` class method. You must declare a variable of type `ui.Window` to hold the window object reference:

```
01 DEFINE w ui.Window
02 LET w = ui.Window.forName("w1")
```

Getting the current window object

The `ui.Window.getCurrent()` class method returns a window object corresponding to the current window. You must declare a variable of type `ui.Window` to hold the window object reference:

```
01 DEFINE w ui.Window
02 LET w = ui.Window.getCurrent()
```

Getting the current form of a window

You can get a ui.Form instance of the current form with the `getForm()` method. This allows you to manipulate form elements by program. You can, for example, hide some parts of a form with setElementHidden().

Getting the DOM node of a window

The `getNode()` method returns the DOM node containing the abstract representation of the window.

Search for a specific element in the window

The `findNode()` method allows you to search for a specific DOM node in the abstract representation of the window content (i.e. the form). You search for a child node by giving its type and the name of the element (i.e. the tagname and the value of the 'name' attribute).

Create a new empty form in a window

The `createForm()` method can be used to create a new empty form. The method returns a new ui.Form instance or NULL if the form name passed as the parameter identifies an existing form used by the window.

Setting the window title

Use the `setText()` method to define the title of the window. By default, the tile of a window is defined by the TEXT attribute of the LAYOUT definition in form files.

Getting the window title

The `getText()` method can be used to get the title set by setText().

Examples

Example 1: Get a window by name and change the title.

```
01 MAIN
02    DEFINE w ui.Window
03    OPEN WINDOW w1 WITH FORM "customer" ATTRIBUTE(TEXT="Unknown")
04    LET w = ui.Window.forName("w1")
05    IF w IS NULL THEN EXIT PROGRAM 1 END IF
06    CALL w.setText("Customer")
07    MENU "Test"
08       COMMAND "exit" EXIT MENU
09    END MENU
10    CLOSE WINDOW w1
11 END MAIN
```

Example 2: Get a the current form and hide a groupbox.

```
01 MAIN
02    DEFINE w ui.Window
03    DEFINE f ui.Form
04    OPEN WINDOW w1 WITH FORM "customer"
05    LET w = ui.Window.getCurrent()
06    IF w IS NULL THEN EXIT PROGRAM 1 END IF
07    LET f = w.getForm()
08    MENU "Test"
09       COMMAND "hide" CALL f.setElementHidden("gb1",1)
```

```
10       COMMAND "exit" EXIT MENU
11    END MENU
12    CLOSE WINDOW w1
13 END MAIN
```

The Form class

Summary:

- Syntax
- Methods
- Usage
  - Defining the default initializer for all forms
  - Getting the DOM node of the form
  - Loading Action Defaults form the form
  - Loading the form ToolBar
  - Loading the form TopMenu
  - Searching for a specific child node in the form
  - Changing the text of a form element
  - Changing the image of a form element
  - Changing the style of a form element
  - Hiding or showing a form element
  - Hiding or showing a form field
  - Changing the style of a form field
- Examples
  - Example 1: Implement a global form initialization function
  - Example 2: Hiding form elements dynamically

See also: Built-in classes, Window class, Windows and Forms

Syntax

The **Form** class is a built-in class that provides an interface to the forms used by the program.

Syntax:

```
ui.Form
```

Methods:

Class Methods	
Name	**Description**
`setDefaultInitializer( fn STRING )`	Defines the default initialization function for all forms used by the program. The function gets a `ui.Form` object as parameter.
Object Methods	

Name	Description
`findNode( t STRING, n STRING ) RETURNING om.DomNode`	Returns the first descendant DOM node of type *t* and matching the name *n* in the abstract representation of this window object.
`getNode( ) RETURNING om.DomNode`	Returns the DOM representation of this Form.
`loadActionDefaults( fn STRING )`	Loads the decoration for actions from a specific XML file into the AUI tree. These action defaults are local to the form and take precedence over action defaults defined at the Interface level. See Action Defaults for more details.
`loadToolBar( fn STRING )`	Loads a Toolbar XML definition into this Form, where *fn* is the name of the file, without extension. If a toolbar exists already in the form, it is replaced.
`loadTopMenu( fn STRING )`	Loads a Topmenu XML definition into this Form, where *fn* is the name of the file, without extension. If a topmenu exists already in the form, it is replaced.
`setElementHidden( name STRING, v INTEGER )`	Changes the 'hidden' attribute of all elements identified by *name*. Values of *v* can be 0,1 or 2.
`setElementText( name STRING, v STRING )`	Changes the 'text' attribute of all elements identified by *name*.
`setElementImage( name STRING, v STRING )`	Changes the 'image' attribute of all elements identified by *name*.
`setElementStyle( name STRING, v STRING )`	Changes the 'style' attribute of all elements identified by *name*.
`setFieldHidden( name STRING, v INTEGER )`	Changes the 'hidden' attribute of a form field identified by *name*. The *name* is a string containing the field qualifier, with an optional prefix ("`[table.]column`"). Values of *v* can be 0,1 or 2.
`setFieldStyle( name STRING, v STRING )`	Changes the 'style' attribute of the view of a form field identified by *name*. The *name* is a string containing the field qualifier, with an optional prefix ("`[table.]column`").

Usage:

The Form class provides an interface to form objects created by an OPEN WINDOW WITH FORM or DISPLAY FORM instruction.

A form object allows you to manipulate form elements by program, typically to hide some parts of a form with the `setElementHidden()` method. The runtime system is able to handle hidden fields during a dialog instruction. You can, for example, hide a GRO containing fields and labels.

You can get a ui.Form instance of the current form with the ui.Window.getForm() method.

Warning: The OPEN FORM instruction does not create a `ui.Form` object; it just declares a handle. It is actually the DISPLAY FORM instruction that instantiates a `ui.Form` object that will be attached to the current window. When a form is displayed with DISPLAY FORM in different windows, the same form specification file will be used to create different `ui.Form` objects.

Defining the default initializer for all forms

With the `setDefaultInitializer()` method, you can specify a default initialization function to implement global processing when a form is opened. The method takes the name of the initialization function as a parameter. That function will be called with a `ui.Form` object as a parameter.

Warning: You must give the initialization function name in lower-case letters to the `setDefaultInitializer()` method. The BDL syntax allows case-insensitive functions names, but the runtime system must reference functions in lower-case letters internally.

Getting the DOM node of the form

The `getNode()` method returns the DOM node containing the abstract representation of the window.

Loading Action Defaults for the form

You may want to load form specific Action Defaults at runtime with the `loadActionDefaults()` method. This method takes the file name as parameter without the **.4ad** suffix. Existing action defaults will be replaced.

Loading the form ToolBar

The `loadToolBar()` method can be used to load a Toolbar XML definition file into the form; for example, in the initialization function. If the form already contains a toolbar, it will be replaced by the new toolbar loaded from this function.

Loading the form TopMenu

The `loadTopMenu()` method can be used to load a Topmenu XML definition file into the form; for example, in the initialization function. If the form alreadycontains a topmenu, it will be replaced by the new topmenu loaded by this function.

Searching for a specific child node in the form

The `findNode()` method allows you to search for a specific DOM node in the abstract representation of the form. You search for a child node by giving its type and the name of the element (i.e. the tagname and the value of the 'name' attribute).

Changing the text of a form element

You may want to modify the text of a static label or group box during program execution. This can be done with the `setElementText()` method. You must pass the identifier of the form element (i.e. the element must have a name in the **.per** file).

Changing the image of a form element

The image of a form element can be changed with the `setElementImage()` method. You must pass the identifier of the form element (i.e. the element must have a name in the **.per** file).

Changing the style of a form element

To change the decoration style of a form element, you can use the `setElementStyle()` method, by passing the identifier of the form element (i.e. the element must have a name in the **.per** file).

Hiding or showing a form element

The `setElementHidden()` method changes the 'hidden' attribute of all form elements identified by a name.

You specify an integer value for the 'hidden' attribute as follows:

Hidden Value	Description
`0`	The element is visible.
`1`	The element is hidden and the user cannot make it visible, typically used to hide information the user is not allowed to see.
`2`	The element is hidden and the user can make it visible.

Hiding or showing a form field

You can use the `setFieldHidden()` method to change the 'hidden' attribute of a form field. Form fields are identified by a fully qualified field name in lower case letters (`"table.column"` or `"formonly.field"`).

You specify an integer value for the 'hidden' attribute as follows:

Hidden Value	Description
`0`	The field is visible.
`1`	The field is hidden and the user cannot make it visible, typically used to hide information the user is not allowed to see.
`2`	The field is hidden and the user can make it visible (for example with a contextual menu, as in tables). This allows you to define columns that are hidden by default, but can be shown by the user.

Changing the style of a form field

To change the style of the view node of a form field, you must use the `setFieldStyle()` method. The form field is identified by a name with an optional prefix (`"table.column"` or `"column"`).

Examples

Example 1: Implement a global form initialization function.

```
01 MAIN
02    CALL ui.Form.setDefaultInitializer("init")
03    OPEN FORM f1 FROM "items"
04    DISPLAY FORM f1 -- Form appears in the default SCREEN window
05    OPEN WINDOW w1 WITH FORM "customer"
06    OPEN WINDOW w2 WITH FORM "orders"
07    DISPLAY FORM f1 -- Form appears in w2 window
08    MENU "Test"
09       COMMAND "exit" EXIT MENU
10    END MENU
11 END MAIN
12
13 FUNCTION init(f)
14    DEFINE f ui.Form
15    DEFINE n om.DomNode
16    CALL f.loadTopMenu("mymenu")
17    LET n = f.getNode()
18    DISPLAY "Init: ", n.getAttribute("name")
19 END FUNCTION
```

Example 2: Hiding form elements dynamically.

```
01 MAIN
02    DEFINE w ui.Window
03    DEFINE f ui.Form
04    DEFINE custid INTEGER
05    DEFINE custname CHAR(10)
06    OPEN WINDOW w1 WITH FORM "customer"
07    LET w = ui.Window.getCurrent()
08    LET f = w.getForm()
09    INPUT BY NAME custid, custname
10      ON ACTION hide
11        CALL f.setFieldHidden("customer.custid",1)
12        CALL f.setElementHidden("label_custid",1)
13      ON ACTION show
14        CALL f.setFieldHidden("customer.custid",0)
15        CALL f.setElementHidden("label_custid",0)
16    END INPUT
17 END MAIN
```

The Dialog class

Summary:

- Syntax
- Methods
- Usage
  - The DIALOG object instance
  - Terminating the dialog
  - Passing the dialog object to a function
  - Getting the total number of rows in a list
  - Setting the total number of rows of a list
  - Registering the next field to jump to
  - Getting the current row of a list
  - Setting the current row in a list
  - Getting the current item having focus
  - Getting the input buffer of a field
  - Getting the touched flag of a field
  - Setting the touched flag of a field
  - Current form used by the dialog
  - Enabling/disabling Actions
  - Enabling/disabling Fields
  - Inserting a new row in a list
  - Appending a new row in a list
  - Deleting a row from a list
  - Deleting all rows from a list
  - Checking form level validation rules
  - Handling default action view visibility
  - Setting the UNBUFFERED mode
  - Setting TTY attributes for cells in lists
- Examples
  - Disable fields dynamically
  - Get form and hide fields
  - Pass a dialog object to a function
  - Set display attributes to cells

See also: Built-in classes, Interaction Model, Multiple Dialogs, Windows and Forms

---

The **Dialog** class is a built-in class that provides an interface to an interactive instruction such as INPUT.

Syntax:

```
ui.Dialog
```

Methods:

Class Methods	
Name	**Description**
`setDefaultUnbuffered( v INTEGER )`	Sets the default of the `UNBUFFERED` attribute for the next dialogs.
Object Methods	
Name	**Description**
`accept( )`	Validates all dialog fields.
`appendRow( arrname STRING )`	Append a row at the end of the list.
`deleteRow( arrname STRING, pos INTEGER )`	Deletes row at position *pos*.
`getArrayLength( arrname STRING ) RETURNING INTEGER`	Returns the total number of rows in a list.
`getCurrentItem( ) RETURNING STRING`	Returns the current item having the focus. This can be a field, list or action.
`getCurrentRow( arrname STRING ) RETURNING INTEGER`	Returns the current row of a list.
`getFieldBuffer( fieldname STRING ) RETURNING STRING`	Returns the current input buffer of the field identified by *fieldname*.
`getFieldTouched( fieldname STRING ) RETURNING STRING`	Returns TRUE if the TOUCHED flag of the specified field is set.
`getForm() RETURNING ui.Form`	Returns the current form used by this dialog.
`insertRow( arrname STRING, pos INTEGER )`	Inserts a row before position *pos*.
`nextField( fieldname STRING )`	Registers the name of the next field that must get the focus when control returns to the dialog.
`setArrayLength( arrname STRING, v INTEGER )`	Sets the total number of rows in a list for paged mode.
`setActionHidden( actname STRING, v INTEGER )`	Hides or shows the default action view identified by *actname*.
`setActionActive( actname STRING, v INTEGER )`	Enables or disables the action identified by *actname*.
`setCurrentRow( arrname STRING, row INTEGER)`	Change the current row in a list.
`setFieldActive( fieldname STRING, v INTEGER )`	Enables or disables the field identified by *fieldname*.
`setFieldTouched( fieldname STRING, v INTEGER )`	Sets the TOUCHED flag of the field identified by *fieldname*.
`setCellAttributes( attarr ARRAY OF RECORD )`	Defines decoration attributes for each cell (singular dialogs only).

`setArrayAttributes( arrname STRING, attarr ARRAY OF RECORD )`	Defines decoration attributes for each cell (multiple dialogs).

Usage:

The DIALOG object instance

This class provides an interface to the interactive instructions INPUT, INPUT ARRAY, DISPLAY ARRAY, CONSTRUCT, and MENU.

The `DIALOG` keyword is a pre-defined object variable. To get an instance of this class, use `DIALOG` inside the interactive instruction block:

```
01 INPUT BY NAME custid, custname
02   ON ACTION disable
03     CALL DIALOG.setFieldActive("custid",0)
04 END INPUT
```

Passing the dialog object to a function

The dialog object is only valid during the execution of the interactive instruction. Using the `DIALOG` keyword outside a dialog instruction block results in a compilation error. However, you can pass the object to a function, in order to write common dialog configuration code:

```
01 INPUT BY NAME custid, custname
02   BEFORE INPUT
03     CALL setupDialog(DIALOG)
04 END INPUT
05
06 FUNCTION setupDialog(d)
07   DEFINE d ui.Dialog
08   IF user_group = "admin" THEN
09      CALL d.setActionActive("delete",1)
10      CALL d.setActionActive("convert",1)
11   ELSE
12      CALL d.setActionActive("delete",0)
13      CALL d.setActionActive("convert",0)
14   END IF
15 END FUNCTION
```

Terminating the dialog

You can use the `accept()` method to validate field input and terminate the dialog. This method is equivalent to the `ACCEPT INPUT` / `ACCEPT DISPLAY` / `ACCEPT DIALOG` instructions. The method is provided as a 3GL alternative to the `ACCEPT` control

instructions, if you need for example to terminate the dialog in a function, outside the context of a dialog block, where control instructions cannot be used.

See ACCEPT DIALOG for more details.

Getting the total number of rows in a list

The `getArrayLength()` method can be used to retrieve the total number of rows of an `INPUT ARRAY` or `DISPLAY ARRAY` list. You must pass the name of the screen array to identify the list:

```
01  DIALOG
02       DISPLAY ARRAY custlist TO sa_custlist.*
03         BEFORE ROW
04          MESSAGE "Row count: " ||
DIALOG.getArrayLength("sa_custlist")
05          ...
06       END DISPLAY
07       INPUT ARRAY ordlist TO sa_ordlist.*
08         BEFORE ROW
09          MESSAGE "Row count: " ||
DIALOG.getArrayLength("sa_ordlist")
10          ...
11       END INPUT
12          ...
```

Setting the total number of rows in a list

The `setArrayLength()` method is used to specify the total number of rows of an `INPUT ARRAY` or `DISPLAY ARRAY` list when using the paged mode. You must pass the name of the screen array to identify the list, followed by an integer expression defining the number of rows. When using a dynamic array without `ON FILL BUFFER` you don't need to specify the total number of rows to the `DIALOG` instruction: It is defined by the number of elements in the array. However, when using the paged mode in a `DISPLAY ARRAY`, the total number of rows does not correspond to the elements in the program array, because the program array holds only a page of the whole list. In any other cases, a call to this method is just ignored.

Registering the next field to jump to

The `nextField()` method can be used register the name of the next field that must get the focus when control goes back to the dialog. This method is similar to the NEXT FIELD instruction, except that it does not implicitly break the program flow. If you want to get the same behavior as `NEXT FIELD`, the method call must be followed by a `CONTINUE DIALOG / INPUT / CONSTRUCT` instruction.

Since this method takes an expression as parameter, you can write generic code, when the name of the target field is not known at compile time. In the next example, the `check_value()` function returns a field name where the value does not satisfy the validation rules:

```
01    DEFINE fn STRING
02    ...
03    ON ACTION save
04          IF ( fn := check_values() ) IS NOT NULL THEN
05              CALL DIALOG.nextField(fn)
06              CONTINUE DIALOG
07          END IF
08          CALL save_data()
09          ...
```

Getting the current row of a list

The `getCurrentRow()` method can be used to retrieve the current row of an `INPUT ARRAY` or `DISPLAY ARRAY` list. You must pass the name of the screen array to identify the list:

```
01  DIALOG
02       DISPLAY ARRAY custlist TO sa_custlist.*
03         BEFORE ROW
04          MESSAGE "Current row: " ||
DIALOG.getCurrentRow("sa_custlist")
05          ...
06       END DISPLAY
07       INPUT ARRAY ordlist TO sa_ordlist.*
08         BEFORE ROW
09          MESSAGE "Current row: " ||
DIALOG.getCurrentRow("sa_ordlist")
10          ...
11       END INPUT
12          ...
```

Setting the current row in a list

If you want to change the current row in an `INPUT ARRAY` or `DISPLAY ARRAY` list, you can use the `setCurrentRow()` method. You must pass the name of the screen array to identify the list, and the new row number:

```
01  DEFINE x INTEGER
02  DIALOG
03       DISPLAY ARRAY custlist TO sa_custlist.*
04          ...
05       END DISPLAY
06       ON ACTION goto_x
07          CALL DIALOG.setCurrentRow("sa_custlist", x)
08          ...
```

Note that moving to a different row with `setCurrentRow()` will not trigger control blocks such as `AFTER ROW`. This method will not set the focus either; You need to use `NEXT FIELD` to set the focus to a list (this works with `DISPLAY ARRAY` as well as with `INPUT ARRAY`).

Getting the current item having focus

The `getCurrentItem()` method returns the name of the current form item having the focus. This can be a simple field, a list or an action view.

- If the focus is on an action view (typically, a BUTTON in the form layout), `getCurrentItem()` returns the name of the corresponding action. Note that if several action views a bound to the same action handler with a unique name, there is no way to distinguish which action view has the focus.
- If the focus is in a simple field driven by an INPUT or CONSTRUCT sub-dialogs, `getCurrentItem()` returns the *field-name* of that current field.
- If the focus is in a list driven by a DISPLAY ARRAY sub-dialog, `getCurrentItem()` returns the *screen-array* name identifying the list.
- If the focus is in a list driven by an INPUT ARRAY sub-dialog, `getCurrentItem()` returns *screen-array.field-name*, identifying both the list and the current field.

Getting the input buffer of a field

The `getFieldBuffer()` method returns the current input buffer of the specified field. The input buffer is used by the dialog to synchronize form fields and program variables. In some situations, especially when using the BUFFERED mode or in a CONSTRUCT, you may want to access that input buffer.

The *fieldname* is a string containing the field qualifier, with an optional prefix ("`[table.]column`").

```
01    LET buff = DIALOG.getFieldBuffer("customer.cust_name")
```

The input buffer can be set with:

- A DISPLAY TO or DISPLAY BY NAME instruction
- The fgl_dialog_setbuffer() function (only for the current field)

Getting the TOUCHED flag of a field

The `getFieldTouched()` method returns TRUE if the TOUCHED flag of the specified field is set.

The *fieldname* is a string containing the field qualifier, with an optional prefix ("`[table.]column`"), or a table prefix followed by a dot and a start ("`table.*`").

```
01    AFTER FIELD cust_name
02       IF DIALOG.getFieldTouched("customer.cust_address") THEN
03          ...
```

If the parameter is a screen record following by dot-star, the method checks the touched flags of all the fields that belong to the screen record:

```
01    ON ACTION quit
02       IF DIALOG.getFieldTouched("customer.*") THEN
03          ...
```

Setting the TOUCHED flag of a field

The `setFieldTouched()` method can be used to change the TOUCHED flag of the specified field(s).

The *fieldname* is a string containing the field qualifier, with an optional prefix ("`[table.]column`"), or a table prefix followed by a dot and a start ("`table.*`").

You typically use this method to set the touched flag when assigning a variable, to emulate a user input. Remember when using the UNBUFFERED mode, you don't need to DISPLAY the value to the fields. The `setFieldTouched()` method is provided as a 3GL replacement for the DISPLAY instructions to set the touched flags.

```
01    ON ACTION zoom_city
02       LET p_cust.cust_city = zoom_city()
02       CALL DIALOG.setFieldTouched("customer.cust_city", TRUE)
03          ...
```

If the parameter is a screen record following by dot-star, the method checks the touched flags of all the fields that belong to the screen record. You typically use this to reset the touched flags of a group of fields, after modifications have been saved to the database, to get back to the initial state of the dialog:

```
01    ON ACTION save
02       CALL save_cust_record()
03       CALL DIALOG.setFieldTouched("customer.*", FALSE)
04          ...
```

Note that the touched flags are reset to false when using an INPUT ARRAY list, every time you leave the modified row.

Current form used by the dialog

The `getForm()` method returns a ui.Form object as a handle to the current form used by the dialog. Use this form object to modify elements of the current form. For example, you can hide some parts of the form with the ui.Form.setElementHidden() method. The runtime system is able to detect hidden fields and exclude them from the input list. See Example 2.

Enabling/Disabling Actions

Dialog actions can be enabled and disabled with the `setActionActive()` dialog method:

```
01    BEFORE DIALOG
02       CALL DIALOG.setActionActive("zoom",FALSE)
```

In GUI applications, push-buttons triggering actions should normally be disabled if the current context / situation makes the corresponding action invalid or unusable. For example, a "print" button should be disabled if there is nothing to print. The `setActionActive()` method is typically used to enable/disable actions according to the context.

The action name must be passed in lowercase letters. Note that sub-dialog actions can be qualified with the sub-dialog prefix within a `DIALOG` instruction. When the `setActionActive()` method is called in the context of the sub-dialog, the prefix can be omitted.

The second parameter of the method must be a boolean expression that evaluates to 0 (`FALSE`) or 1 (`TRUE`).

To simplify action activation, you can write a common "setup" function which centralizes all rules to enable/disable actions. This function can then be called from any place in the `DIALOG` instruction, passing the DIALOG object as parameter. See Example 3.

In the DIALOG instruction, actions can be prefixed with the sub-dialog identifier. The `setActionActive()` method can be take a full-qualified action name as parameter.

Enabling/Disabling Fields

Form fields used by the dialog can be enabled/disabled with the `setFieldActive()` method. The fields are identified by the field name used in the dialog, with an optional table prefix ("`column`", "`table.column`" or "`formonly.field`"). When a field is disabled, it is still visible, but the user cannot edit the value.

```
01    ON CHANGE cust_name
02       CALL DIALOG.setFieldActive( "customer.cust_addr",
(rec.cust_name IS NOT NULL) )
```

See also Example 1.

Inserting a new row in a list

The `insertRow()` method is just a basic row creation function, to insert a row in the list at a given position. It does not set the current row or raise any `BEFORE ROW` / `BEFORE INSERT` / `AFTER INSERT` / `AFTER ROW` control blocks. It is quite similar to inserting a new element in the program array, except the internal registers are automatically updated (like the total number of rows returned by getArrayLength()). If the list was decorated with cell attributes, the program array defining the attributes will also be synchronized.

Warning: This method should not be called in control blocks such as `BEFORE ROW`, `AFTER ROW`, `BEFORE INSERT`, `AFTER INSERT`, `BEFORE DELETE`, `AFTER DELETE`, but it can safely be used in an `ON ACTION` block.

TO REMOVE: Default values defined in the form file with the DEFAULT attribute are applied when calling this method.

After the method is called, a new row is created in the program array, so you can assign values to the variables before the control goes back to the user. The `getArrayLength()` method will return the new row count.

Note that the method does not set the current row and does not give the focus to the list; you need to call `setCurrentRow()` and execute NEXT FIELD if you want to give the focus.

The method takes two parameters:

1. The name of the screen array identifying the list.
2. The index where the row has to be inserted (starts at 1).

If the index is greater as the number of rows, a new row will be appended at the end or the list. This will be equivalent to call the `appendRow()` method.

Warning: If the list is empty, getCurrentRow() will return zero. So you must test this and use 1 instead to reference the first row otherwise you can get -1326 errors when using the program array.

Following code example shows a user-define action to insert ten rows in the list at the current position:

```
01    ON ACTION insert_some_rows
02       LET r = DIALOG.getCurrentRow("sa")
03       IF r == 0 THEN LET r = 1 END IF
04       FOR i = 10 TO 1 STEP -1
05          CALL DIALOG.insertRow("sa", r)
06          LET p_items[r].item_quantity = 1.00
07       END FOR
```

Appending a new row in a list

The `appendRow()` method is just a basic row creation function to append a row to the end of the list. It does not set the current row or raise any `BEFORE ROW` / `BEFORE INSERT` / `AFTER INSERT` / `AFTER ROW` control blocks. It is quite similar to appending a new element to the program array, except the internal registers are automatically updated (like the total number of rows returned by getArrayLength()). If the list was decorated with cell attributes, the program array defining the attributes will also be synchronized.

Warning: This method should not be called in control blocks such as `BEFORE ROW`, `AFTER ROW`, `BEFORE INSERT`, `AFTER INSERT`, `BEFORE DELETE`, `AFTER DELETE`, but it can safely be used in an `ON ACTION` block.

TO REMOVE: Default values defined in the form file with the DEFAULT attribute are applied when calling this method.

After the method is called, a new row is created in the program array, so you can assign values to the variables before the control goes back to the user. The `getArrayLength()` method will return the new row count.

Note that the method does not set the current row and does not give the focus to the list; you need to call `setCurrentRow()` and execute NEXT FIELD if you want to give the focus.

The `appendRow()` method does not create a temporary row as the implicit append action of `INPUT ARRAY`; The row is considered as permanent once it is added.

The method takes only one parameter:

1. The name of the screen array identifying the list.

Following code example implements a user-defined to append ten rows at the end of the list:

```
01    ON ACTION append_some_rows
02       FOR i = 1 TO 10
03          CALL DIALOG.appendRow("sa")
04          LET r = DIALOG.getArrayLength("sa")
05          LET p_items[r].item_quantity = 1.00
06       END FOR
```

Deleting a row from a list

The `deleteRow()` method is just a basic row deletion function. It does not reset the current row or raise any `BEFORE DELETE` / `AFTER DELETE` / `AFTER ROW` / `BEFORE ROW` control blocks (except in some particular situation as described below). It is quite similar to deleting an element to the program array, except that internal registers are automatically updated (like the total number of rows returned by getArrayLength()). If the list was decorated with cell attributes, the program array defining the attributes will also be synchronized.

Warning: This method should not be called in control blocks such as `BEFORE ROW`, `AFTER ROW`, `BEFORE INSERT`, `AFTER INSERT`, `BEFORE DELETE`, `AFTER DELETE`, but it can safely be used in an `ON ACTION` block.

After the method is called, the row does no more exist in the program array, and the `getArrayLength()` method will return the new row count.

If the `deleteRow()` method is called during an `INPUT ARRAY`, and after the call no more rows are in the list, the dialog will automatically append a new temporary row if the focus is in the list, to let the user enter new data. **When using AUTO APPEND = FALSE attribute, no temporary row will be created and the current row register will be automatically changed to make sure that it will not be greater as the total number of rows.**

If deleteRow() method is called during an INPUT ARRAY or DISPLAY ARRAY that has the focus, the BEFORE ROW control block will be executed if you delete the current row. This is required to reset the internal state of the dialog.

The method takes two parameters:

1. The name of the screen array identifying the list.
2. The index of the row to be deleted (starts at 1).

If you pass zero as row index, the method does nothing (if no rows are in the list, getCurrentRow() returns zero).

Following code example implements a user-defined action to remove the rows that have a specific property:

```
01    ON ACTION delete_invalid_rows
02       FOR r = 1 TO DIALOG.getArrayLength("sa")
03          IF NOT s_orders[t].is_valid THEN
04            CALL DIALOG.deleteRow("sa",r)
05            LET r = r - 1
06          END IF
07       END FOR
```

Deleting all rows of a list

The deleteAllRows() method removes all the rows of a list driven by a DISPLAY ARRAY or INPUT ARRAY. This is equivalent to a deleteRow() call, but instead of deleting one particular row, it removes all rows of the specified list.

Warning: This method should not be called in control blocks such as BEFORE ROW, AFTER ROW, BEFORE INSERT, AFTER INSERT, BEFORE DELETE, AFTER DELETE, but it can safely be used in an ON ACTION block.

After the method is called, all rows are deleted from the program array, and the getArrayLength() method will return zero.

The method takes the name of the screen-array as parameter.

If the deleteAllRows() method is called during an INPUT ARRAY, the dialog will automatically append a new temporary row if the focus is in the list, to let the user enter new data. **When using AUTO APPEND = FALSE attribute, no temporary row will be created and the current row register will be automatically changed to make sure that it will not be greater as the total number of rows.**

If deleteAllRows() method is called during an INPUT ARRAY or DISPLAY ARRAY that has the focus, the BEFORE ROW control block will be executed if you delete the current row. This is required to reset the internal state of the dialog.

Checking form level validation rules

In order to execute NOT NULL, REQUIRED and INCLUDE validation rules defined in the form specification files, you can call the `validate()` method by passing a list of fields or screen records as parameter. The method returns zero if success and the input error code of the first field which does not satisfy the validation rules.

Note that the current field is always checked, even if it is not part of the validation field list. This is mandatory, otherwise the current field may be left with invalid data.

If an error occurs, the `validate()` method automatically displays the corresponding error message, and registers the next field to jump to when the interactive instruction gets the control back.

Warning: The validate() method does not stop code execution if an error is detected. You must execute a `CONTINUE DIALOG` or `CONTINUE INPUT` instruction to cancel the code execution.

A typical usage is for a "save" action:

```
01    ON ACTION save
02       IF DIALOG.validate("cust.*") < 0 THEN
03         CONTINUE DIALOG
04       END IF
05       CALL customer_save()
```

Handling default action view visibility

The Default View on an action can made visible or invisible with the `setActionHidden()` dialog method:

```
01    ON ACTION hide
02       CALL DIALOG.setActionHidden( "confirm", 1 )
```

Values of the second parameter can be 1 or 0.

Setting the UNBUFFERED mode

By default dialogs are not sensitive to variable changes. To make a dialog sensitive, use the `UNBUFFERED` attribute in the dialog instruction definition. However, you can define the default for all subsequent dialogs by using the `setDefaultUnbuffered()` class method:

```
01 CALL ui.Dialog.setDefaultUnbuffered(TRUE)
```

Setting TTY attributes for cells in lists

Multiple Dialogs

In an INPUT ARRAY or DISPLAY ARRAY instruction, the `setArrayAttributes()` method can be used to specify display attributes for each cell. You must define an array with the same number of record elements as the data array used by the `INPUT ARRAY` or `DISPLAY ARRAY`. Each element must have the same name as in the data array, and must be defined with a character data type (you typically use STRING).

Fill the display attributes array with color and video attributes. These must be specified in lowercase characters and separated by a blank (ex: "`red reverse`"). Then, pass the array to the dialog with the `setArrayAttributes()` method, in a `BEFORE INPUT` or `BEFORE DISPLAY` block. Display attributes can be changed dynamically during the dialog:

```
01    ON ACTION set_attributes
02       CALL DIALOG.setArrayAttributes( "sr", attarr )
```

For a complete example, see Example 4.

Singular Dialogs

An equivalent method called `setCellAttributes()`, takes only the program array as argument. The `setCellAttributes()` method is designed for singular dialogs, where only one screen array is used.

Examples

Example 1: Disable fields dynamically.

```
01 FUNCTION input_customer()
02    DEFINE custid INTEGER
03    DEFINE custname CHAR(10)
04    INPUT BY NAME custid, custname
05      ON ACTION enable
06        CALL DIALOG.setFieldActive("custid",1)
07      ON ACTION disable
08        CALL DIALOG.setFieldActive("custid",0)
09    END INPUT
10 END FUNCTION
```

Example 2: Get the form and hide fields.

```
01 FUNCTION input_customer()
02    DEFINE f ui.Form
03    DEFINE custid INTEGER
04    DEFINE custname CHAR(10)
```

```
05    INPUT BY NAME custid, custname
06      BEFORE INPUT
08        LET f = DIALOG.getForm()
09        CALL f.setElementHidden("customer.custid",1)
10    END INPUT
11 END FUNCTION
```

Example 3: Pass a dialog object to a function.

```
01 FUNCTION input_customer()
02    DEFINE custid INTEGER
03    DEFINE custname CHAR(10)
04    INPUT BY NAME custid, custname
05      BEFORE INPUT
06        CALL setup_dialog(DIALOG)
07    END INPUT
08 END FUNCTION
09
10 FUNCTION setup_dialog(d)
11    DEFINE d ui.Dialog
12    CALL d.setActionActive("print",user.can_print)
13    CALL d.setActionActive("query",user.can_query)
14 END FUNCTION
```

Example 4: Set display attributes for cells.

```
01 FUNCTION display_customer()
02    DEFINE i INTEGER
03    DEFINE arr DYNAMIC ARRAY OF RECORD
04                  key INTEGER,
05                  name CHAR(10)
06               END RECORD
07    DEFINE att DYNAMIC ARRAY OF RECORD
08                  key STRING,
09                  name STRING
10               END RECORD
11
12    FOR i=1 TO 10
13       CALL arr.appendElement()
14       LET arr[i].key = i
15       LET arr[i].name = "name " || i
16       CALL att.appendElement()
17       IF i MOD 2 = 0 THEN
18          LET att[i].key = "red"
19          LET att[i].name = "blue reverse"
20       ELSE
21          LET att[i].key = "green"
22          LET att[i].name = "magenta reverse"
23       END IF
24    END FOR
25
26    DIALOG ATTRIBUTES(UNBUFFERED)
27      DISPLAY ARRAY arr TO sr.*
28      ON ACTION att_set
29        CALL DIALOG.setArrayAttributes("sr", att)
30      ON ACTION att_clear
```

```
31        CALL DIALOG.setArrayAttributes("sr", NULL)
32    END DIALOG
33
34 END FUNCTION
```

The ComboBox class

Summary:

- Syntax
- Methods
- Usage
  - Defining the default initializer for ComboBoxes
  - Searching for a ComboBox in the current form
  - Clearing the item list of a ComboBox
  - Adding an element to the ComboBox item list
  - Getting the name of the form field table prefix
  - Getting the name of the form field
  - Getting the ComboBox tag value
  - Getting the number of items in a ComboBox
  - Getting an item name by position
  - Getting an item position by name
  - Getting an item text by position
  - Getting an item text by item name
  - Removing an item from the ComboBox
- Examples

See also: Built-in Classes, Form Specification File

Syntax

The **ComboBox** class provides an interface to the COMBOBOX form field view in the Abstract User Interface tree.

Syntax:

```
ui.ComboBox
```

Methods:

Class Methods	
Name	**Description**
`forName( fieldname STRING ) RETURNING ui.ComboBox`	Returns the combobox object identified by *fieldname* in the current form.
`setDefaultInitializer( funcname STRING )`	Defines the default initialization function for all COMBOBOX form fields. See also the INITIALIZER attribute in form specification files.

Object Methods	
Name	**Description**
`clear( )`	Clears the list of combobox items.
`addItem( name STRING, text STRING )`	Adds an item to the combobox item list.
`getColumnName()` `  RETURNING STRING`	Returns the name of the column name of the form field associated with this combobox.
`getIndexOf( name STRING )` `  RETURNING STRING`	Returns the position of a given item by name.
`getItemCount()` `  RETURNING INTEGER`	Returns the current number of items defined in the combobox .
`getItemName( index INTEGER )` `  RETURNING STRING`	Returns the name of an item at a given position. First is 1.
`getItemText( index INTEGER )` `  RETURNING STRING`	Returns the text of an item at a given position. First is 1.
`getTableName()` `  RETURNING STRING`	Returns the name of the table, alias or FORMONLY of the form field associated to this combobox.
`getTag()` `  RETURNING STRING`	Returns the user-defined tag of this object.
`getTextOf( name STRING )` `  RETURNING STRING`	Returns the text for a given item name.
`removeItem( name STRING )`	Removes the item identified by *name*.

Usage:

When you declare a COMBOBOX form field in the form specification file, you declare both a form field and a view for that model. The ComboBox class is an interface to the view of a COMBOBOX form field.

Defining the default initializer for ComboBoxes

Use the `ui.ComboBox.setDefaultInitializer()` class method to define the default initialization function that will be called each time a ComboBox object is created. That function is called with the ComboBox object as the parameter:

```
01 CALL ui.ComboBox.setDefaultInitializer("initcombobox")
02
03 FUNCTION initcombobox(cb)
04   DEFINE cb ui.ComboBox
05   CALL cb.clear()
06   CALL cb.addItem(1,"Paris")
07   CALL cb.addItem(2,"London")
08   CALL cb.addItem(3,"Madrid")
09 END FUNCTION
```

Warning: You must give the initialization function name in lower-case letters to the `setDefaultInitializer()` method. The BDL syntax allows case-insensitive function names, but the runtime system must reference functions in lower-case letters internally.

You can also define a specific initialization function in the form specification file, by using the INITIALIZER attribute (as shown in the second example below).

Searching for a ComboBox in the current form

The `ui.ComboBox.forName()` class method searches for a COMBOBOX object by form field name in the current form. Typically, after loading a form with OPEN WINDOW WITH FORM, you use the class method to retrieve a COMBOBOX view object into a variable defined as a `ui.ComboBox`:

```
01 DEFINE cb ui.ComboBox
02 LET cb = ui.ComboBox.forName("formonly.airport")
```

It is recommended that you verify if that function has returned an object, because the form field may not exist:

```
01 IF cb IS NULL THEN
02   ERROR "Form field not found in current form"
03   EXIT PROGRAM
04 END IF
```

Once instantiated, the ComboBox object can be used; for example, to set up the items of the drop down list:

```
01 CALL cb.clear()
02 CALL cb.addItem(1,"Paris")
03 CALL cb.addItem(2,"London")
04 CALL cb.addItem(3,"Madrid")
```

Clearing the item list of a ComboBox

The `clear()` method clears the item list. If the item list is empty, the ComboBox drop-down button will show an empty list on the client side.

Adding an element to the ComboBox item list

The `addItem()` method adds an item to the end of the list. It takes two parameters: the first is the real form field value, and the second is the value to be displayed in the drop down list. If the second parameter is NULL, the runtime system automatically uses the first parameter as the display value:

Getting the name of the form field table prefix

The `getTableName()` method returns the name of the form field table prefix. Not that this prefix can be NULL if not defined at the form field level.

Getting the name of the form field

The `getColumnName()` method returns the name of the form field table prefix. Not that this prefix can be NULL if not defined at the form field level.

You can use the `getTableName()` and `getColumnName()` methods to get the table and column name of the form field associated with the ComboBox:

```
01 DISPLAY cb.getTableName()||"."||cb.getColumnName()
```

Getting the ComboBox tag value

The `getTag()` method returns the value of the TAG attribute if defined in the form file.

Getting an item position by name

The `getIndexOf()` method takes an item name as parameter and returns the position of the item in the list. Returns 0 if the item name does not exist.

With this method you can check if an item exists:

```
01 IF cb.getIndexOf("SFO") == 0 THEN
02   CALL cb.addItem("SFO", "San Francisco International Airport, CA" )
03 END IF
```

Getting the number of items in a ComboBox

You can get the current number of items defined in a ComboBox with the `getItemCount()` method. If no items are defined, the method returns zero.

Getting an item name by position

The `getItemName()` method takes an item position as parameter and returns the identifier of that item. First item starts at position 1.

Getting an item text by position

The `getItemText()` method takes an item position as parameter and returns the value of that item. First item starts at position 1.

Getting an item text by name

The `getTextOf()` method takes an item name as parameter and returns the value of that item. Returns NULL if the item name does not exist.

Removing an item by name

Use the `removeItem()` method to delete an item from the ComboBox. Item name must be passed as parameter.

Examples

Example 1: Get a COMBOBOX form field view and fill the item list:

Form Specification File:

```
01 DATABASE FORMONLY
02 LAYOUT
03 GRID
04 {
05   Airport:  [cb01                ]
06 }
07 END
08 END
09 ATTRIBUTES
10 COMBOBOX cb01 = FORMONLY.airport TYPE CHAR;
11 END
```

Program File:

```
01 MAIN
02   DEFINE cb ui.ComboBox
03   DEFINE airport CHAR(3)
04
05   OPEN FORM f1 FORM "combobox"
06   DISPLAY FORM f1
07   LET cb = ui.ComboBox.forName("formonly.airport")
08   IF cb IS NULL THEN
09      ERROR "Form field not found in current form"
10      EXIT PROGRAM
11   END IF
12   CALL cb.clear()
13   CALL cb.addItem("CDG", "Paris-Charles de Gaulle, France")
14   CALL cb.addItem("LCY", "London-City Airport, UK")
15   CALL cb.addItem("LHR", "London-Heathrow, UK")
16   CALL cb.addItem("FRA", "Frankfurt Airport, Germany")
17   IF cb.getIndexOf("SFO") == 0 THEN
18      CALL cb.addItem("SFO", "San Francisco International Airport,
CA" )
19   END IF
20
21   INPUT BY NAME airport
22
23 END MAIN
```

Example 2: Using the INITIALIZER attribute in the form file:

Form Specification File:

```
01 DATABASE FORMONLY
02 LAYOUT
```

```
GRID
{
  Airport:  [cb01                ]
}
END
END
ATTRIBUTES
COMBOBOX cb01 = FORMONLY.airport TYPE INTEGER,
INITIALIZER=initcombobox;
END
```

Initialization function:

```
FUNCTION initcombobox(cb)
  DEFINE cb ui.ComboBox
  CALL cb.clear()
  CALL cb.addItem("CDG", "Paris-Charles de Gaulle, France")
  CALL cb.addItem("LCY", "London-City Airport, UK")
  CALL cb.addItem("LHR", "London-Heathrow, UK")
  CALL cb.addItem("FRA", "Frankfurt Airport, Germany")
  CALL cb.addItem("SFO", "San Francisco International Airport, CA" )
END MAIN
```

The DomDocument class

Summary:

- Syntax
- Methods
- Usage
  - Create a new DomDocument object
  - Create a new DomNode object
  - Return the root node of the DomDocument
  - Return a specific node of the DomDocument
  - Remove a DomNode object
- Examples

See also: Built-in Classes, XML Utils

Syntax

Purpose:

The **DomDocument** class provides methods to manipulate a data tree, following the DOM standards.

Syntax:

```
om.DomDocument
```

Methods:

Class Methods	
Name	**Description**
`create( tag STRING ) RETURNING om.DomDocument`	Creates a new, empty DomDocument object, where *tag* identifies the tag name of the root element.
`createFromXmlFile( file STRING ) RETURNING om.DomDocument`	Creates a new DomDocument object using an XML file specified by the parameter *file*. Returns NULL if an error occurs.
`createFromString( source STRING ) RETURNING om.DomDocument`	Creates a new DomDocument object by parsing the XML string passed as parameter. Returns NULL if an error occurs.
Object Methods	

Name	Description
`copy( src om.DomNode, deep INTEGER ) RETURNING om.DomNode`	Clones a DomNode (with child nodes if *deep* is TRUE).
`createChars( text STRING ) RETURNING om.DomNode`	Creates a DomNode as a text node.
`createEntity( text STRING ) RETURNING om.DomNode`	Creates a DomNode as an entity node.
`createElement( tag STRING ) RETURNING om.DomNode`	Creates a new empty DomNode object with a tag name specified by *tag*.
`getDocumentElement( ) RETURNING om.DomNode`	Returns the root node of the DOM document.
`getElementById( id INTEGER ) RETURNING om.DomNode`	Gets an element using its id, the internal integer identifier automatically assigned to each DomNode object.
`removeElement( e om.DomNode )`	Removes a DomNode object and any descendent DomNodes from the document.

Usage:

A DomDocument object holds a DOM tree of DomNode objects.

A unique root DomNode object is owned by a DomDocument object.

Create a new DomDocument object

To create an instance of the DomDocument class, you must first declare a variable with the type om.DomDocument.

Then, use the method `om.DomDocument.create()` to instantiate a new, empty DomDocument object.

To create a document from an existing XML file, use the method `om.DomDocument.createFromXmlFile()`. You can also use the `om.DomDocument.createFromString()` method to create a document from a string in memory.

Create a new DomNode object

New nodes can be created with the `createElement()` method.

New text nodes can be created with the `createChars()` method.

New entity nodes can be created with the `createEntity()` method.

Clone a DomNode object using the `copy()` method.

Once a new DomNode object is created, you can for example inserted it in the DOM tree with the `insertBefore()` method of a DomNode object

Return the root node of the DomDocument

You can get the root node with the `getDocumentElement()` method. Once you have the root element, you can recursively manipulate child nodes with the DomNode class methods

Return a specific node of the DomDocument

You can get a specific node of the DomDocument by using its internal identifier with the `getElementById()` method.

Remove a DomNode object

Use the `removeElement()` method to remove an Element from a DomDocument.

Examples

Example 1:

```
01 MAIN
02   DEFINE d om.DomDocument
03   DEFINE r om.DomNode
04   LET d = om.DomDocument.create("MyDocument")
05   LET r = d.getDocumentElement()
06 END MAIN
```

The DomNode class

Summary:

- Syntax
- Methods
- Usage
  - Node creation/removal
  - In/Out utilities
  - Node Identification
  - Attributes management
  - Tree navigation
- Examples

See also: Built-in Classes, XML Utils, NodeList

Syntax

The **DomNode** class provides methods to manipulate a node of a data tree, following the DOM standards.

Syntax:

```
om.DomNode
```

Notes:

1. A DomNode object is a node (or element) of a DomDocument.

Methods:

Object Methods	
Name	**Description**
Node creation	
`appendChild( src om.DomNode )`	Adds a DomNode at the end of the list of children in this node.
`createChild( tag STRING ) RETURNING om.DomNode`	Creates a DomNode and adds it to the children list of this node.
`insertBefore( new om.DomNode, exn om.DomNode )`	Inserts a DomNode just before the existing node referenced by *exn*.
`removeChild( node om.DomNode )`	Removes the child node referenced by *node* and removes any of its descendents.

`replaceChild( new om.DomNode, old om.DomNode )`	Replaces the child node referenced by *old* by the node *new*.
In/Out Utilities	
`loadXml( file STRING ) RETURNING om.DomNode`	Creates a new DomNode object by loading an XML file and attaches it to this node as a child. The new created node object is returned from the function.
`parse( source STRING )`	Parses a source string in XML format and creates a new DomNode from it.
`toString( ) RETURNING STRING`	Serializes the DomNode to a string in XML format.
`writeXml( file STRING )`	Writes an XML file with the current node.
`write( shd om.SaxDocumentHandler )`	Outputs an xml-tree to a sax document handler.
Node identification	
`getId( ) RETURNING INTEGER`	Returns the internal integer identifier automatically assigned to an Abstract User Interface DomNode. Returns -1 for nodes that are not part of the Abstract User Interface tree.
`getTagName( ) RETURNING STRING`	Returns the tag name of the node.
Attributes management	
`setAttribute( att STRING, val STRING )`	Sets the attribute *att* with value *val*.
`getAttribute( att STRING ) RETURNING STRING`	Returns the value of the attribute having the name *att*.
`getAttributeInteger(att STRING, def INTEGER) RETURNING INTEGER`	Returns the value of the attribute having the name *att* as an integer value. Returns *def* if the attribute is not defined.
`getAttributeString(att STRING, def STRING) RETURNING INTEGER`	Returns the value of the attribute having the name *att* as a string value. Returns *def* if the attribute is not defined.
`getAttributeName( pos INTEGER ) RETURNING STRING`	Returns the name of the attribute at the position *pos* (1 = first).
`getAttributesCount() RETURNING INTEGER`	Returns the number of attributes of this DomNode.
`getAttributeValue( pos INTEGER ) RETURNING STRING`	Returns the value of the attribute at the position *pos* (1 = first).
`removeAttribute( att STRING )`	Deletes the attribute identified by *att*.
Tree navigation	
`getChildCount( ) RETURNING INTEGER`	Returns the number of children.

`getChildByIndex( pos INTEGER ) RETURNING om.DomNode`	Returns the child node at index *pos* (1 = first).
`getFirstChild( ) RETURNING om.DomNode`	Returns the first child node.
`getLastChild( ) RETURNING om.DomNode`	Returns the last DomNode in the list of children.
`getNext( ) RETURNING om.DomNode`	Returns the next sibling DomNode of this node.
`getParent( ) RETURNING om.DomNode`	Returns the parent DomNode of this node.
`getPrevious( ) RETURNING om.DomNode`	Returns the previous sibling DomNode of this node.
`selectByTagName( name STRING ) RETURNING om.NodeList`	Creates a list of nodes by recursively searching nodes by tag name.
`selectByPath( path STRING ) RETURNING om.NodeList`	Creates a list of nodes by recursively searching nodes matching an XPath-like pattern.

Usage:

Node creation/removal

To create an instance of the DomNode class from scratch, you must instantiate the object using one of the methods provided in the DomNode class:

`createChild()` creates a child node and adds it to the children list.

`appendChild()` creates a child node and adds it to the end of the children list.

`insertBefore()` inserts a child node before the specified existing node.

`replaceChild()` replaces the specified child node with a different child node.

Other methods to create DomNode objects are available from the DomDocument class. For example, to create a *text node*, use the `createChars()` method of a DomDocument object; to create an *entity node*, use the `createEntity()` method of a DomDocument object.

`removeChild()` removes the specified child node.

In/Out utilities

The DomNode class provides the `writeXml()` method to save the DOM tree into a file in XML format.

The method `write()` outputs an xml-tree to a sax document handler.

The method `loadXml()` creates a new DomNode object by loading an XML file and attaches it to this node as a child.

Use the `toString()` method to generate a string in XML format from the DomNode. To scan an XML source string and create a DomNode from, use the `parse()` method.

Attributes management

A DomNode object can have attributes with values, except if it is a *text node*. In this case, you can only get/set the text of the node, since text nodes cannot have attributes. The DomNode class provides a complete set of methods to modify attribute values:

`getAttribute()` returns the value of the attribute having the specified name.

`setAttribute()` sets the value of the specified attribute.

`getAttributeInteger()` returns the value of the specified attribute as an integer value.

`getAttributeString()` returns the value of the specified attribute as a string value.

`getAttributeName()` returns the name of the attribute at the specified position.

`getAttributesCount()`returns the number of attributes of this DomNode.

`getAttributeValue()` returns the value of the attribute at the specified position.

`removeAttribute()` deletes the specified attribute.

Node Identification

To get the tag name of the DOM node, use the `getTagName()` method.

The method `getId()` returns the internal integer identifier automatically assigned to an DomNode.

Tree navigation

A DomNode object can have zero or more DomNode children that can have, in turn, other children. The DomNode class provides a complete set of methods to manipulate *DomNode* child objects.

`getChildCount()` returns the number of children.

`getChildByIndex()` returns the child node at the specified index position.

`getFirstChild()` returns the first child node.

`getLastChild()` returns the last DomNode in the list of children.

`getNext()` returns the next sibling DomNode of this node.

`getParent()` returns the parent DomNode of this node.

`getPrevious()` returns the previous sibling DomNode of this node.

The `selectByTagName()` and `selectByPath()` methods allow you to search for children nodes according to a tag name (i.e. a type of node) or by using an XPath-like search criteria. See the NodeListclass for more details.

Warnings:

1. Tag and attribute names are case sensitive; "Wheel" is not the same as "wheel".
2. Text nodes cannot have attributes, but they have plain text.
3. In text nodes, the characters can be accessed with the `@chars` attribute name.
4. In XML representation, a text node is the text itself. Do not confuse it with the parent node. For example, `<Item id="32">Red shoes</Item>` represents 2 nodes: The parent 'Item' node and a text node with string 'Red shoes'.

Tips:

1. If you need to identify an element, use a common attribute like "name".
2. If you need to label an element, use a common attribute like "text".

Examples

Example 1:

To create a DOM tree with the following structure (represented in XML format):

```
<Vehicles>
   <Car name="Corolla" color="Blue" weight="1546">Nice car! Yes,
very nice!
   </Car>
   <Bus name="Maxibus" color="Yellow" weight="5278">
      <Wheel width="315" diameter="925" />
      <Wheel width="315" diameter="925" />
      <Wheel width="315" diameter="925" />
      <Wheel width="315" diameter="925" />
   </Bus>
</Vehicles>
```

You write the following:

```
01 MAIN
02   DEFINE d om.DomDocument
```

```
03   DEFINE r, n, t, w om.DomNode
04   DEFINE i INTEGER
05
06   LET d = om.DomDocument.create("Vehicles")
07   LET r = d.getDocumentElement()
08
09   LET n = r.createChild("Car")
10   CALL n.setAttribute("name","Corolla")
11   CALL n.setAttribute("color","Blue")
12   CALL n.setAttribute("weight","1546")
13
14   LET t = d.createChars("Nice car!")
15   CALL n.appendChild(t)
16   LET t = d.createEntity("nbsp")
17   CALL n.appendChild(t)
18   LET t = d.createChars("Yes, very nice!")
19   CALL n.appendChild(t)
20
21   LET n = r.createChild("Bus")
22   CALL n.setAttribute("name","Maxibus")
23   CALL n.setAttribute("color","yellow")
24   CALL n.setAttribute("weight","5278")
25   FOR i=1 TO 4
26     LET w = n.createChild("Wheel")
27     CALL w.setAttribute("width","315")
28     CALL w.setAttribute("diameter","925")
29   END FOR
30
31   CALL r.writeXml("Vehicles.xml")
32
33 END MAIN
```

Example 2:

The following example displays a DOM tree content recursively:

```
01 FUNCTION displayDomNode(n,e)
02   DEFINE n om.DomNode
03   DEFINE e, i, s INTEGER
04
05   LET s = e*2
06   DISPLAY s SPACES || "Tag: " || n.getTagName()
07
08   DISPLAY s SPACES || "Attributes:"
09   FOR i=1 TO n.getAttributesCount()
10      DISPLAY s SPACES || "  " || n.getAttributeName(i) || "=[" ||
n.getAttributeValue(i) ||"]"
11   END FOR
12   LET n = n.getFirstChild()
13
14   DISPLAY s SPACES || "Child Nodes:"
15   WHILE n IS NOT NULL
16     CALL displayDomNode(n,e+1)
17     LET n = n.getNext()
18   END WHILE
19
```

```
20 END FUNCTION
```

Example 3:

The following example outputs a Dom tree without indentation.

```
01 MAIN
02   DEFINE d om.DomDocument
03   DEFINE r, n, t, w om.DomNode
04   DEFINE dh om.SaxDocumentHandler
05
06   DEFINE i INTEGER
07
08   LET dh = om.XmlWriter.createPipeWriter("cat")
09   CALL dh.setIndent(FALSE)
10
11   LET d = om.DomDocument.create("Vehicles")
12   LET r = d.getDocumentElement()
13
14   LET n = r.createChild("Car")
15   CALL n.setAttribute("name","Corolla")
16   CALL n.setAttribute("color","Blue")
17   CALL n.setAttribute("weight","1546")
18
19   LET t = d.createChars("Nice car!")
20   CALL n.appendChild(t)
21
22   LET n = r.createChild("Bus")
23   CALL n.setAttribute("name","Maxibus")
24   CALL n.setAttribute("color","yellow")
25   CALL n.setAttribute("weight","5278")
26   FOR i=1 TO 4
27     LET w = n.createChild("Wheel")
28     CALL w.setAttribute("width","315")
29     CALL w.setAttribute("diameter","925")
30   END FOR
31
32   CALL r.write(dh)
33
34 END MAIN
```

The NodeList class

Summary:

- Syntax
- Methods
- Usage
- Examples

See also: Built-in Classes, XML Utils

Syntax

The **NodeList** class holds a list of DomNode objects created from a selection method.

Syntax:

```
om.NodeList
```

Notes:

1. A *NodeList* object is created from a DomNode.selectByTagName() or DomNode.selectByPath() method.

Methods:

Object Methods	
Name	**Description**
`item( index INTEGER )` `  RETURNING om.DomNode`	Returns the DomNode object at the given position (first is 1). Returns NULL if the item does not exist.
`getLength( )` `  RETURNING INTEGER`	Returns the number of items in the list.

Usage

A NodeList object contains a list of the child objects of the DomNode from which it was created, selected by Tag Name or Path. Use the DomNode methods to create the NodeList, as shown in the examples below.

Once the NodeList object is created, the following Object methods are available:

`item()` - This method returns the DomNode that is at the specified position in the list.

`getLength()` - This method returns the total number of DomNodes in the list.

Examples

Example 1: Search for child nodes by tag name:

```
01 MAIN
02   DEFINE nl om.NodeList
03   DEFINE r, n om.DomNode
04   DEFINE i INTEGER
05
06   LET r = ui.Interface.getRootNode()
07   LET nl = r.selectByTagName("Form")
08
09   FOR i=1 to nl.getLength()
10     LET n = nl.item(i)
11     DISPLAY n.getAttribute("name")
12   END FOR
13
14 END MAIN
```

Example 2: Search for child nodes by XPath:

```
01 MAIN
02   DEFINE nl om.NodeList
03   DEFINE r, n om.DomNode
04   DEFINE i INTEGER
05
06   LET r = ui.Interface.getRootNode()
07   LET nl = r.selectByPath("//Window[@name=\"screen\"]")
08
09   FOR i=1 to nl.getLength()
10     LET n = nl.item(i)
11     DISPLAY n.getAttribute("name")
12   END FOR
13
14 END MAIN
```

The SaxAttributes class

Summary:

- Syntax
- Methods
- Usage
  - Creating a SaxAttributes object
  - Returning the value of an Attribute
  - Returning the number of Attributes
  - Returning the name of an Attribute
  - Adding Attributes
  - Removing Attributes
  - Replacing the Attributes list
- Examples

See also: Built-in Classes, XML Utils

Syntax

The **SaxAttributes** class provides methods to manipulate XML element attributes.

Syntax:

```
om.SaxAttributes
```

Methods:

Class Methods	
Name	**Description**
`copy( src SaxAttributes ) RETURNING om.SaxAttributes`	Clones an existing SaxAttributes object.
`create( ) RETURNING om.SaxAttributes`	Creates a new, empty SaxAttributes object.
Object Methods	
Name	**Description**
`addAttribute( n STRING, v STRING )`	Adds an attribute to the end of the list.
`clear( )`	Clears the attribute list.
`getLength( ) RETURNING INTEGER`	Returns the number of attributes in the list.
`getName( pos INTEGER ) RETURNING STRING`	Returns the name of the attribute at position *pos*.

`getValue( att STRING ) RETURNING STRING`	Returns the value of the attribute identified by the name *att*.
`getValuebyIndex( pos INTEGER ) RETURNING STRING`	Returns the value of the attribute at position *pos*.
`removeAttribute( pos INTEGER ) RETURNING INTEGER`	Removes the attribute at position *pos*.
`setAttributes( atts om.SaxAttributes )`	Clears the current attribute list and adds all attributes of *atts*.

Usage:

This class provides basic methods to manipulate attributes of an XML element. The SaxAttributes object is a list containing the attributes of the element.

Creating a SaxAttributes object

To process element attributes, a SaxAttributes object can be used in cooperation with an XmlReader object. You get an instance of SaxAttributes with the `getAttributes()` method.

The following SaxAttributes Class methods are also provided:

The `om.SaxAttributes.create()` method creates a new, empty SaxAttributes object.

The `om.SaxAttributes.copy()` method clones an existing SaxAttributes object.

Returning the value of an attribute

The `getValue()` method returns the value of the attribute specified by name.

The `getValueByIndex()` method returns the value of the attribute at the specified position.

Returning the number of attributes

The `getLength()` method returns the number of attributes in the list.

Returning the name of an attribute

The `getName()` method returns the name of the attribute at the specified position in the list.

Adding attributes

The `addAttribute()` method adds a new attribute to the end of the attributes list.

Removing attributes

The `clear()` method clears the attributes list.

The `removeAttribute()` method removes the attributes at the given position in the list.

Replacing the attributes list

The `setAttributes()` method clears the current list and adds all the attributes of the specified SaxAttributes object.

Examples

Example 1:

```
01 FUNCTION displayAttributes( a )
02    DEFINE a om.SaxAttributes
03    DEFINE i INTEGER
04    FOR i=1 to a.getLength()
05       DISPLAY a.getName(i) || "=[" || a.getValueByIndex(i) || "]"
06    END FOR
07 END FUNCTION
```

The SaxDocumentHandler class

Summary:

- Syntax
- Methods
- Usage
  - Creating a SaxDocumentHandler object
  - Loading the document
  - Processing the document
- Examples

See also: Built-in Classes, XML Utils, XmlWriter

Syntax

The **SaxDocumentHandler** class provides an interface to write an XML filter, following the SAX standards.

Warning: You must either write a BDL module dedicated to the implementation of the filter methods, or create the SaxDocumentHandler object with a XmlWriter creation method; see usage for more details.

Syntax:

```
om.SaxDocumentHandler
```

Methods:

Class Methods	
Name	**Description**
`createForName( module STRING ) RETURNING om.SaxDocumentHandler`	Creates a SaxDocumentHandler object using a BDL module.
Object Methods	
Name	**Description**
`readXmlFile( file STRING )`	Reads an XML file and applies the filter.
`setIndent ( indenting BOOLEAN )`	Enables output indentation if `indenting` is TRUE. Indentation is enabled by default
`startDocument( )`	Processes the beginning of the document.

`startElement( tag STRING, atts SaxAttributes )`	Processes the beginning of an element having the tag name *tag* and the attributes *atts*.
`characters( text STRING )`	Processes characters of a text node.
`skippedEntity( text STRING )`	Processes an unresolved entity.
`endElement( tag STRING )`	Processes the end of an element having the tag name *tag*.
`endDocument( )`	Processes the end of the document.
`processingInstruction( n STRING, a STRING )`	Processes a processing instruction with the name *n* and attributes *a*.

Usage:

This class can be used in two different ways:

1. To implement an XML SAX filter, using BDL functions defined in a module.
2. To write an XML document to a file, process or socket output, using XmlWriter creation methods.

This page describes the first usage; see XmlWriter for more details about the second usage.

With the SaxDocumentHandler class, you can implement a SAX filter by using a BDL module to write the methods handling the standard SAX events. The SaxDocumentHandler class also provides methods to process all SAX events by hand. This is useful if you want to chain SAX filters.

Creating a SaxDocumentHandler object

First, you create the SaxDocumentHandler object with the `om.SaxDocumentHandler.createForName(`*`module`*`)` method, which takes a BDL module as a parameter:

```
01 DEFINE filter om.SaxDocumentHandler
02 LET filter = om.SaxDocumentHandler.createForName("module1")
```

When doing this, the runtime system loads the BDL module and attaches its functions to the SaxDocumentHandler methods by name.

Loading the document

To process a document, you typically load it from an XML file:

```
01 CALL filter.readXmlFile("xmlsource")
```

Processing the document

The module must implement the following functions to match the SAX filter events:

- `startDocument()`: Called one time at the beginning of the document processing.
- `processingInstruction(name,data)`: Called when a processing instruction is reached, identified by *name*, with *data* information.
- `startElement(name,attr)`: Called when an XML element is reached, identified by the tag *name*, having the *attr* attributes (SaxAttributes).
- `characters(chars)`: Called when a text node is reached, having the characters *chars*.
- `skippedEntity(chars)`: Called when an unknown entity node is reached (like &xxx; for example). Entity name is stored in *chars*.
- `endElement(name)`: Called when the end of an XML element is reached, identified by the tag *name*.
- `endDocument()`: Called one time at the beginning of the document processing.

In these functions, you are free to process the XML document as you wish. You can use the SaxAttributes methods to get the attributes of an element, transform the values or ignore some attributes, and write directly to a file or to the database, or even chain directly with another SaxDocumentHandler. If you want to write to an XML file, you typically use an XmlWriter object.

By default, the SaxDocumentHandler object outputs XML with indentation. If you want to disable indentation, use the `setIndent()` method:

```
01 CALL myhdlr.setIndent(FALSE)
```

Examples

Example 1: Extracting phone numbers from a directory.

This example shows how to write a SAX filter to extract phone numbers from a directory file written in XML.

```
01 MAIN
02   DEFINE f om.SaxDocumentHandler
03   LET f = om.SaxDocumentHandler.createForName("module1")
04   CALL f.readXmlFile("customers")
05 END MAIN
```

Notes:

1. In Line `03`, the input parameter specifies the name of a source file that has been compiled into a .42m file ("module1.42m" in our example).

The module "module1.4gl":

```
01 FUNCTION startDocument()
02 END FUNCTION
03
04 FUNCTION processingInstruction(name,data)
05   DEFINE name,data STRING
06 END FUNCTION
07
08 FUNCTION startElement(name,attr)
09   DEFINE name STRING
10   DEFINE attr om.SaxAttributes
11   DEFINE i INTEGER
12   IF name="Customer" THEN
13      DISPLAY attr.getValue("lname")," ",
14             attr.getValue("fname"),":",
15             COLUMN 60, attr.getValue("phone")
16   END IF
17 END FUNCTION
18
19 FUNCTION endElement(name)
20   DEFINE name STRING
21 END FUNCTION
22
23 FUNCTION endDocument()
24 END FUNCTION
25
26 FUNCTION characters(chars)
27   DEFINE chars STRING
28 END FUNCTION
29
30 FUNCTION skippedEntity(chars)
31   DEFINE chars STRING
32 END FUNCTION
```

The XML file "customers":

```
<Customers>
  <Customer customer_num="101" fname="Ludwig" lname="Pauli"
    company="All Sports Supplies" address1="213 Erstwild Court"
    address2="" city="Sunnyvale" state="CA" zipcode="94086"
    phone="408-789-8075" />
  <Customer customer_num="102" fname="Carole" lname="Sadler"
    company="Sports Spot" address1="785 Geary St"
    address2="" city="San Francisco" state="CA" zipcode="94117"
    phone="415-822-1289" />
  <Customer customer_num="103" fname="Philip" lname="Currie"
    company="Phil&apos;s Sports" address1="654 Poplar"
    address2="P. O. Box 3498" city="Palo Alto" state="CA"
    zipcode="94303" phone="415-328-4543" />
</Customers>
```

The XmlReader class

Summary:

- Syntax
- Methods
- Usage
  - Creating an XMLReader object
  - Processing Events
- Examples

See also: Built-in Classes, XML Utils

Syntax

The **XmlReader** class provides methods to read and process a file written in XML format, following the SAX standards.

Syntax:

```
om.XmlReader
```

Methods:

Class Methods	
Name	**Description**
`createFileReader( file STRING ) RETURNING om.XmlReader`	Creates an XmlReader reading from a file.
Object Methods	
Name	**Description**
`getAttributes( ) RETURNING om.SaxAttributes`	Returns the attribute list of the current element.
`getCharacters( ) RETURNING STRING`	Returns the string value of the current text element.
`skippedEntity( ) RETURNING STRING`	Returns the name of the entity.
`getTagName( ) RETURNING STRING`	Returns the tag name of the current element.
`read( ) RETURNING STRING`	Reads the next XML fragment and returns the name of the SAX event that occurs. This can be one of `StartDocument`, `StartElement`, `Characters`, `EndElement`,

`EndDocument`.

Usage:

The processing of the XML file is streamed-data based; the file is loaded and processed sequentially with events. To process element attributes, an XmlReader object must cooperate with a SaxAttributes object. The XmlReader class can only read from a file. To write to a file, you must use the XmlWriter class.

Creating an XMLReader object

First, you must declare a variable of type *om.XmlReader*, and use the `om.XmlReader.createFileReader(`*`filename`*`)` method to create the object; where *filename* is a string expression defining the name of the file to be read.

As with the standard SAX API, the XML file is parsed on the basis of events. Once the XmlReader object is created with the `om.XmlReader.createFileReader()` method, you can successively call the `read()` method to get named events indicating how to parse the XML file.

Processing Events

The following events can be returned by the `read()` method:

Event name	Description	Action
`StartDocument`	Beginning of the document	Prepare processing (allocate resources)
`StartElement`	Beginning of a node	Get current element's tagname or attributes `XmlReader.getTagName()` `XmlReader.getAttributes()`
`Characters`	Value of the current element	Get current element's value `XmlReader.getCharacters()`
`SkippedEntity`	Name of the entity	Get current element's value `XmlReader.skippedEntity()`
`EndElement`	Ending of a node	Get current element's tagname `XmlReader.getTagName()`
`EndDocument`	Ending of the document	Finish processing (release resources)

To process element attributes, you must declare a variable of type SaxAttributes. This object represents a set of attributes of an element. You get an object of this class with the `getAttributes()` method. Once created from the XmlReader, the SaxAttributes

object is automatically updated based on the element currently processed by the XmlReader.

Examples

Example:

```
01 MAIN
02    DEFINE i, l INTEGER
03    DEFINE r om.XmlReader
04    DEFINE e String
05    DEFINE a om.SaxAttributes
06    LET r = om.XmlReader.createFileReader("myfile.xml")
07    LET a = r.getAttributes()
08    LET l = 0
09    LET e = r.read()
10    WHILE e IS NOT NULL
11      CASE e
12        WHEN "StartDocument"
13          DISPLAY "StartDocument:"
14        WHEN "StartElement"
15          LET l=l+1
16          DISPLAY l SPACES, "StartElement:", r.getTagName()
17          FOR i=1 to a.getLength()
18            DISPLAY l SPACES,"  ",
19              a.getName(i)," = ",
20              a.getValueByIndex(i)
21          END FOR
22        WHEN "Characters"
23          DISPLAY l SPACES, "  Characters:'",r.getCharacters(),"'"
24        WHEN "EndElement"
25          DISPLAY l SPACES, "EndElement:", r.getTagName()
26          LET l=l-1
27        WHEN "EndDocument"
28          DISPLAY "EndDocument:"
29        OTHERWISE
30          DISPLAY "Invalid event: ",e
31      END CASE
32      LET e=r.read()
33    END WHILE
34 END MAIN
```

The XmlWriter class

Summary:

- Syntax
- Methods
- Usage
  - Create a SaxDocumentHandler object writing to a file
  - Create a SaxDocumentHandler object writing to a pipe
  - Create a SaxDocumentHandler object writing to a socket
- Examples

See also: Built-in Classes, XML Utils, SaxDocumentHandler

Syntax

The **XmlWriter** class allows you to write XML documents to different types of output, following the SAX standards.

Syntax:

```
om.XmlWriter
```

Methods:

Class Methods	
Name	**Description**
`createFileWriter( file STRING ) RETURNING om.SaxDocumentHandler`	Creates a SaxDocumentHandler object writing to a file.
`createPipeWriter( exec STRING ) RETURNING om.SaxDocumentHandler`	Creates a SaxDocumentHandler object writing to a pipe created for a process.
`createSocketWriter( host STRING, port STRING ) RETURNING om.SaxDocumentHandler`	Creates a SaxDocumentHandler object writing to a socket.

Usage:

This class is only used to create a SaxDocumentHandler object, by using one of the class methods described in the above table.

First, define a variable of type *om.SaxDocumentHander* and create the object with one of the "create" methods, according to the output destination:

- The method `om.XmlWriter.createFileWriter(filename)` creates an object writing to the file identified by *filename*.
- The method `om.XmlWriter.createPipeWriter(progname)` creates an object writing to a pipe opened by a sub-process identified by *progname*.
- The method `om.XmlWriter.createSocketWriter(hostname,portnum)` creates an object writing to the TCP socket identified by the host *hostname* and the TCP port *portnum*.

To handle element attributes, define a variable of type om.SaxAttributes.

Create the SaxAttributes object with `om.SaxAttributes.create()` class method. See the class definition for more details about attributes definition.

Use SaxDocumentHandler methods

Use the method `startDocument()` to start writing to the output. From this point, the order of method calls defines the structure of the XML document.

To write an element, fill the SaxAttributes object with attributes. Then, initiate the element output with the method `startElement(name,attset)`, where *name* is the tag name of the element and *attset* is the SaxAttributes object defining element attributes. After this call, you can write text nodes with the `characters()` method. You can write an entity node with the `entity()` method. Finish element output with a call to the `endElement()` method. Repeat these steps as many times as you have elements to write.

Instead of using the `startElement()` method, you can generate processing instruction elements with `processingInstruction(name,attset)`, where name is the *name* of the application. The resulting XML output is in the form: `<?name attribute="value1" ... ?>`

Finally, you must finish the document output with a `endDocument()` call.

Examples

Example 1:

The following code writes an HTML page to a file using the XmlWriter class:

```
01 MAIN
02   DEFINE w om.SaxDocumentHandler
03   DEFINE a,n om.SaxAttributes
04
05   LET w = om.XmlWriter.createFileWriter("sample.html")
```

```
06   LET a = om.SaxAttributes.create()
07   LET n = om.SaxAttributes.create()
08
09   CALL n.clear()
10
11   CALL w.startDocument()
12
13     CALL w.startElement("HTML",n)
14
15       CALL w.startElement("HEAD",n)
16
17         CALL w.startElement("TITLE",n)
18         CALL w.characters("HTML page generated with XmlWriter")
19         CALL w.endElement("TITLE")
20
21         CALL a.clear()
22         CALL a.addAttribute("type","text/css")
23         CALL w.startElement("STYLE",a)
24         CALL w.characters("\nBODY { background-color:#c0c0c0; }\n")
25         CALL w.endElement("STYLE")
26
27       CALL w.endElement("HEAD")
28
29       CALL w.startElement("BODY",n)
30
31         CALL addHLine(w)
32         CALL addTitle(w,"What is XML?",1,"55ff55")
33         CALL addParagraph(w,"XML = eXtensible Markup Language ...")
34
35         CALL addHLine(w)
36         CALL addTitle(w,"What is SAX?",1,"55ff55")
37         CALL addParagraph(w,"SAX = Simple Api for XML ...")
38
39       CALL w.endElement("BODY")
40
41     CALL w.endElement("HTML")
42
43   CALL w.endDocument()
44
45 END MAIN
46
47 FUNCTION addHLine(w)
48   DEFINE w om.SaxDocumentHandler
49   DEFINE a om.SaxAttributes
50   LET a = om.SaxAttributes.create()
51   CALL a.clear()
52   CALL a.addAttribute("width","100%")
53   CALL w.startElement("HR",a)
54   CALL w.endElement("HR")
55 END FUNCTION
56
57 FUNCTION addTitle(w,t,x,c)
58   DEFINE w om.SaxDocumentHandler
59   DEFINE t VARCHAR(100)
60   DEFINE x INTEGER
61   DEFINE c VARCHAR(20)
62   DEFINE a om.SaxAttributes
```

```
63   DEFINE n varchar(10)
64   LET a = om.SaxAttributes.create()
65   LET n = "h" || x
66   CALL a.clear()
67   CALL w.startElement(n,a)
68   IF c IS NOT NULL THEN
69     CALL a.addAttribute("color",c)
70   END IF
71   CALL w.startElement("FONT",a)
72   CALL w.characters(t)
73   CALL w.endElement("FONT")
74   CALL w.endElement(n)
75 END FUNCTION
76
77 FUNCTION addParagraph(w,t)
78   DEFINE w om.SaxDocumentHandler
79   DEFINE t VARCHAR(2000)
80   DEFINE a om.SaxAttributes
81   LET a = om.SaxAttributes.create()
82   CALL a.clear()
83   CALL w.startElement("P",a)
84   CALL w.characters("Text is:")
84   CALL w.skippedEntity("nbsp")         # Add a non breaking space :
 
84   CALL w.characters("is")
84   CALL w.characters(t)
85   CALL w.endElement("P")
86 END FUNCTIO
```

Environment Variables

Summary:

- Setting Environment Variables on UNIX
- Setting Environment Variables on Windows
- Operating System Environment Variables
  - PATH
  - LD_LIBRARY_PATH
  - LC_ALL
- Database Client Environment Variables
- Genero BDL Environment Variables
  - DBDATE
  - DBCENTURY
  - DBDELIMITER
  - DBEDIT
  - DBFORMAT
  - DBMONEY
  - DBPATH
  - DBPRINT
  - FGLDBPATH
  - FGLDIR
  - FGLIMAGEPATH
  - FGLLDPATH
  - FGLGUI
  - FGLGUIDEBUG
  - FGLPROFILE
  - FGLSERVER
  - FGLSOURCEPATH
  - FGLSQLDEBUG
  - FGLWRTUMASK

See also: Tools, Localization Support, Connections, Installation and Setup

---

Setting Environment Variables on UNIX

On UNIX platforms, environment variables can be set through the following methods, depending on to the command interpreter used:

Bourne shell:

```
VAR=value; export VAR
```

Korn shell:

```
export VAR=value
```

C shell:

```
setenv VAR=value
```

For more details, refer to the documentation for your UNIX system.

Setting Environment Variables on Windows

On Windows platforms, environment variables can be set by one of the following methods:

- In a command window, with the SET command.
- In the registry, for the current user in `HKEY_CURRENT_USER` or a global setting in `HKEY_LOCAL_MACHINE`.

For more details, refer to the documentation of your Windows system.

Warnings:

1. When using Informix, some variables related to the database engine must be set using the SETNET32 utility.
2. On Windows, double quotes do not have the same meaning as on UNIX systems. For example, if you set a variable with the command `SET VAR="abc"`, the value of the variable will be `"abc"` (with double quotes), and not `abc`.

Operating System Environment Variables

This section describes a couple of well-known system environment variables that are used by Genero software components.

PATH

Purpose:

This variable defines the list of search path for executables files.

Notes:

1. On Unix platforms, **PATH** defines search path list for executable programs.
2. On Windows platforms, **PATH** defines search path for programs and DLLs.
3. The path separator is a colon (`:`) on UNIX and a semicolon `(;)` on Windows.

LD_LIBRARY_PATH

Purpose:

This variable defines the list of search path for shared libraries loaded by the *dynamic linker* on Unix platforms.

Notes:

1. On some operating systems, the environment variable defining the shared library search path may have a different name. For example, on a system where a 32b and a 64b environment coexist, you may need to set LD_LIBRARY_PATH_64 to execute the 64b programs.

LC_ALL

Purpose:

This variable defines the locale (language, territory and code-set) for Unix programs.

Notes:

1. This variable is used by the runtime system to handle character strings. It is important to set this variable properly according to the character set used by your application.
2. If LC_ALL is not defined, LANG is used instead.
3. Read the Unix manual of the setlocale C function for more details about this variable. See also the Localization page.

Database Client Environment Variables

If your Genero programs connect to a database server, you will probably have to set database vendor specific environment variables (or registry settings on Windows platforms). You must for example set INFORMIXDIR for an Informix client, ORACLE_HOME for Oracle, etc. Read carefully the database client software documentation. You get also some details in the Database Connections page of this documentation.

Genero Environment Variables

This section lists and describes in detail all Genero specific environment variables.

DBDATE

Purpose:

Defines the default display format for DATE values and the default picture for automatic string-to-DATE conversions.

Values:

Values can be a restricted combination of several symbols described in the following table:

Symbol	Meaning in DBDATE format string
`D`	Day of month as one or two digits
`M`	Month as one or two digits
`Y2`	Year as two digits
`Y4`	Year as four digits
`/`	Default time-unit separator for the default locale
`-`	Minus time-unit separator
`,`	Coma time-unit separator
`.`	Period time-unit separator
`0`	Indicates no time-unit separator

The combinations must follow a specific order:

```
{ DM | MD } { Y2 | Y4 } { / | - | , | . | 0 }
{ Y2 | Y4 } { DM | MD } { / | - | , | . | 0 }
```

Notes:

1. **DBDATE** defines the order of the *month*, *day*, and *year* time units within a date, whether the *year* is printed with two digits (Y2) or four digits (Y4) and the time-unit separator between the *month*, *da*y, and *year.*
2. In programs, when you assign a string representing a date to a variable defined with the DATE data type, automatic string-to-DATE conversion takes place based on the **DBDATE** definition.
3. In the default locale, the default setting for **DBDATE** is `DMY4/`.
4. Date formatting specified in a USING clause or FORMAT attribute overrides the formatting specified in **DBDATE**.

Example:

```
DBDATE="DMY4/"
export DBDATE
```

DBDELIMITER

Purpose:

The **DBDELIMITER** environment variable defines the value delimiter for LOAD and UNLOAD instructions.

Notes:

1. If **DBDELIMITER** is not defined, the default delimiter is a (|) pipe.

Warnings:

1. Do not use backslash or hex digits (0-9, A-F, a-f).

Example:

```
DBDELIMITER="@"
export DBDELIMITER
```

DBCENTURY

Purpose:

The **DBCENTURY** environment variable specifies how to expand abbreviated one- and two-digit *year* specifications within DATE and DATETIME values.

Values:

Symbol	Algorithm for Expanding Abbreviated Years
C	Use the past, future, or current year closest to the current date.
F	Use the nearest year in the future to expand the entered value.
P	Use the nearest year in the past to expand the entered value.
R	Prefix the entered value with the first two digits of the current year.

Notes:

1. Default value is "R" (prefix the entered value with the first two digits of the current year).
2. Values are case sensitive; only the four uppercase letters are valid.
3. Three-digit years are not expanded.
4. If a year is entered as a single digit, it is first expanded to two digits by prefixing it with a zero; **DBCENTURY** then expands this value to four digits.
5. Years before 99 AD (or CE) require leading zeros (to avoid expansion).

Warnings:

1. If the database server and the client system have different settings for DBCENTURY, the client system setting takes precedence for abbreviations of years in dates entered through the application. Expansion is sensitive to the time of execution and to the accuracy of the system clock-calendar. You can avoid the need to rely on DBCENTURY by requiring the user to enter four-digit years or by setting the CENTURY attribute in the form specification of DATE and DATETIME fields.

DBEDIT

Purpose:

The **DBEDIT** environment variable defines the editor program to be used for TEXT fields.

DBFORMAT

Purpose:

The **DBFORMAT** environment variable defines the input and display format for numbers.

See also DBMONEY.

Syntax:

```
front:thousands:decimal:back
```

Notes:

1. *front* is the leading currency symbol, can be an asterisk (*).
2. *throusands* is a character that you specify as a valid thousands separator, can be an asterisk (*).
3. *decimal* is a character that you specify as a valid decimal separator.

4. *back* is the trailing currency symbol, can be an asterisk (*).

Usage:

The DBFORMAT environment variable specifies the format in which values are entered, displayed, or passed to the database for number data types: `MONEY`, `DECIMAL`, `INTEGER`, `SMALLINT`, `FLOAT`, `SMALLFLOAT`.

The default format specified in DBFORMAT affects numeric and monetary values in display, input, and output operations.

DBFORMAT can specify the leading and trailing currency symbols (but not their default positions within a monetary value) and the decimal and thousands separators. The decimal and thousands separators defined by DBFORMAT apply to both monetary and other numeric data.

Features of BDL affected by the setting in DBFORMAT include (but are not restricted to) the following items:

- USING operator or FORMAT attribute
- DISPLAY or PRINT statement
- LET statement, where a CHAR, VARCHAR or STRING variable is assigned a monetary or number value
- LOAD and UNLOAD statements that use ASCII files (or whatever the locale regards as a *flat* file) to pass data to or from the database
- PREPARE statements that process number values

The asterisk (*) specifies that a symbol or separator is not applicable; it is the default for any *fron*t, *thousand*s, or *back* term that you do not define.

If you specify more than one character for *decimal* or *thousand*s, the values in the *decimal* or *thousand*s list cannot be separated by spaces (nor by any other symbols). BDL uses the first value specified as the thousands or decimal separator when displaying the number or currency value in output. The user can include any of the decimal or thousands separators when entering values.

Any printable character that your locale supports is valid for the thousands separator or for the decimal separator, except:

- Digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
- <, >, |, ?, !, =, [,]

The same character cannot be both the thousands and decimal separator. A blank space (ASCII 32) can be the thousands separator (and is conventionally used for this purpose in some locales). The asterisk (*) symbol is valid as the decimal separator, but is not valid as the thousands separator.

The colon (:) symbol is supported as *thousands* or *decimal* separator but must be preceded by a backslash (\ ) symbol, as in the specification `:\:.:DM`. .

You must include all three colons. Enclosing the DBFORMAT specification in a pair of single quotation marks is recommended to prevent the shell from attempting to interpret (or execute) any of the DBFORMAT characters.

The setting in DBFORMAT also affects how formatting masks of the FORMAT attribute and of the USING operator are interpreted. In formatting masks of FORMAT and USING, the following symbols are not literal characters but are placeholders for what DBFORMAT specifies:

- The dollar ($) sign is a placeholder for the *front* currency symbol.
- The comma is a placeholder for the thousands separator.
- The period is a placeholder for the decimal separator.

In formatting masks of the FORMAT attribute, the at (@) sign is a placeholder for the *back* currency symbol. (The @ symbol has no special significance in formatting masks for the USING operator.)

The following table illustrates the results of different combinations of DBFORMAT setting and format string on the same value.

Value	Format String	DBFORMAT	Result
1234.56	$$#,###.##	$:,:.:	$1,234.56
1234.56	$$#,###.##	:.:,:DM	1.234,56
1234.56	#,###.##@	$:,:.:	1,234.56
1234.56	#,###.##@	:.:,:DM	1.234,56DM

When the user enters number or currency values, the runtime system behaves as follows:

- It disregards any *front* (leading) or *back* (trailing) currency symbol and any thousands separators that the user enters.
- If a symbol is entered that was defined as a decimal separator in DBFORMAT, it is interpreted as the decimal separator.

When the runtime system displays or prints values:

- The DBFORMAT-defined leading or trailing currency symbol is displayed for MONEY values.
- If a leading or trailing currency symbol is specified by the FORMAT attribute for non-MONEY data types, the symbol is displayed.
- The thousands separator is not displayed unless it is included in a formatting mask of the FORMAT attribute or of the USING operator.

When MONEY values are converted to character strings by the LET statement, both automatic data type conversion and explicit conversion with a USING clause insert the DBFORMAT-defined separators and currency symbol into the converted strings.

For example, suppose DBFORMAT is set as follows:

```
*:.:,:DM
```

The value 1234.56 will print or display as follows:

```
1234,56DM
```

Here DM stands for deutsche marks. Values input by the user into a screen form are expected to contain commas, not periods, as their decimal separator because DBFORMAT has `*:.:,:DM` as its setting in this example.

DBMONEY

Purpose:

The **DBMONEY** environment variable defines the currency symbol and the decimal separator for MONEY values.

See also DBFORMAT.

Syntax:

front{.|,}*back*

Notes:

1. *front* is a character string representing a leading currency symbol that precedes the value.
2. *back* is a character string representing a trailing currency symbol that follows the value.
3. The currency symbol can be up to seven characters long and can contain any character except a comma or a period.
4. The currency symbol can be non-ASCII characters if your current locale supports a code set that defines the non-ASCII characters you use.

Usage:

This variable is used to display or input MONEY values and for implicit data conversion between MONEY values and character strings.

Currency symbol and decimal separator characters must be specified in this environment variable.

The position of the currency symbol (relative to the decimal separator) indicates whether the currency symbol appears before or after the MONEY value. When the currency symbol is positioned in DBMONEY before the decimal separator, it is displayed before the value ($1234.56). When it is positioned after the decimal separator, it is displayed after the value (1234.56F).

The runtime system recognizes the period (.) and the comma (,) as decimal separators. All other characters are considered to be part of the currency symbol. For example, ", FR" defines a MONEY format with the comma as decimal separator and the string " FR" (including the space) as the currency symbol.

The default value for DBMONEY is "$.", defining the currency symbol as the dollar sign ($) and the decimal separator as the period (.).

Because only its position within a DBMONEY setting indicates whether a symbol is the *front* or *back* currency symbol, the decimal separator is required. If you use DBMONEY to specify a *back* symbol, for example, you must supply a decimal separator (a comma or period). Similarly, if you use DBMONEY to change the decimal separator from a period to a comma, you must also supply a currency symbol.

To avoid ambiguity in displayed numbers and currency values, do not use the thousands separator of DBFORMAT as the decimal separator of DBMONEY. For example, specifying comma as the DBFORMAT thousands separator dictates using the period as the DBMONEY decimal separator.

Example:

```
DBMONEY="$."
export DBMONEY
DBMONEY=",F"
export DBMONEY
```

DBPATH

Purpose:

The **DBPATH** environment variable defines the paths to search for program resource files.

Notes:

1. If **DBPATH** is not defined, the default is the current directory.
2. You can provide a list of paths, using system-specific path separators.

Usage:

The **DBPATH** environment variable contains the search paths for the following type of files:

1. Form files loaded by OPEN FORM,
2. Message files used by OPTIONS HELP FILE,
3. Resource files of Localized strings.

Warnings:

1. The path separator is platform specific (**":"** on UNIX platforms and **";"** on Windows platforms).

Example (UNIX):

```
DBPATH="/user/forms1:/user/form2:/usr/strings/french"
export DBPATH
```

DBPRINT

Purpose:

The **DBPRINT** environment variable specifies the print device to be used by reports defined `TO PRINTER`.

Notes:

1. On UNIX systems, the **DBPRINT** environment variable typically contains the printer queue command (such as **lp**).
2. To have the DVM print to the printer on the client running the Genero Desktop Client (GDC), set `DBPRINT=FGLSERVER`.

Example (UNIX):

```
DBPRINT="lpr"
export DBPRINT
```

Example (Client):

```
DBPRINT=FGLSERVER
export DBPRINT
```

FGLDIR

Purpose:

The **FGLDIR** environment variable defines the Genero BDL software installation directory.

Warnings:

1. The **FGLDIR** environment variable must be set in order to use the product components.

FGLIMAGEPATH

Purpose:

The **FGLIMAGEPATH** environment variable defines the search paths to find images for the front-end.

Notes:

1. By default, the image directory is the current directory where the program was started.
2. You can provide a list of paths, using system-specific path separators.

Warnings:

1. The path separator is platform specific (**":"** on UNIX platforms and **";"** on Windows platforms).

Usage:

When the front-end needs to display an image which is specified with a simple file name (not an URL), the front end first looks for local image files on the user workstation. If the image file is not found locally, the front-end sends an image request to the runtime system, which provides the image from the server file-system.

You define the search path for images with the **FGLIMAGEPATH** environment variable.

The runtime system searches for image files in the locations described below. The search depends from the name of the image file, the list of directories defined in **FGLIMAGEPATH**, and the expected file extensions provided by the front-end:

- Name of the image file is: "*file*"
- Content of FGLIMAGEPATH: "*dir1*:*dir2*"
- List of extensions provided by the front-end: .gif, .png

The image file would be searched in the following locations:

1. ./*file*
2. ./*file*.gif
3. ./*file*.png
4. *dir1*/*file*
5. *dir1*/*file*.gif
6. *dir1*/*file*.png
7. *dir2*/*file*
8. *dir2*/*file*.gif
9. *dir2*/*file*.png

Example:

```
FGLIMAGEPATH="/user/myimages:/user/myicones"
export FGLIMAGEPATH
```

FGLLDPATH

Purpose:

The **FGLLDPATH** environment variable defines the search paths to load C extensions and modules.

Usage:

A Genero program can be composed by several p-code modules (**42m**) and can use C extensions. When linking and when executing the program, the runtime system must known where to search for these modules. You can use the **FGLLDPATH** environment variable to define the search paths to load C extensions and p-code modules.

Modules are searched in several implicit directories in the following order:

1. The directory where the program (**42r**) file resides.
2. A path defined in the **FGLLDPATH** environment variable.
3. The FGLDIR/lib directory.
4. The current directory.

Warnings:

1. The path separator is platform specific (**":"** on UNIX platforms and **";"** on Windows platforms).
2. This variable is used at link time and at run time.

Example:

```
FGLLDPATH="/user/modules1:/user/modules2"
export FGLLDPATH
```

See also: IMPORT.

FGLGUI

Purpose:

The **FGLGUI** environment variable indicates whether the applications are run in TUI or GUI mode.

Notes:

1. When set to 0 (zero), the application executes in TUI mode.
2. When set to 1 (one), the application executes in GUI mode.

FGLGUIDEBUG

Purpose:

The **FGLGUIDEBUG** environment variable defines the debug level in GUI mode.

Notes:

1. When set to 0 (zero), no debug information is generated.

Usage:

By default, the GUI protocol commands are compressed and not easy to read on the client debug log. If you set this variable to a value different from zero, the protocol commands are indented for better read.

If **FGLGUIDEBUG** is not set to 0, debug information about the compression initialization is generated.

FGLSERVER

Purpose:

The **FGLSERVER** environment variable defines the hostname and port of the graphical front end to be used by the runtime system to display the application windows.

Syntax:

```
{hostname|ipaddress}[:servnum]
```

Notes:

1. *hostname* is the name of a machine on the network.
2. *ipaddress* is the IP V4 address (Ex: 10:0:0:105).
3. *servnum* identifies the front end. This number also implicitly defines the TCP port number the front end is listening to, as an offset for the base port 6400.

Warnings:

1. The *servnum* parameter defines the front end server number (first is 0, second is 1, and so on) and implicitly the TCP port. The port number base is 6400. For example, when using 1, the runtime system connects to the TCP port 6401.

Example:

```
FGLSERVER="mars:0"
export FGLSERVER
```

See also: Automatic front-end startup.

FGLSOURCEPATH

Purpose:

The **FGLSOURCEPATH** environment variable defines the path to source files for the debugger.

Notes:

1. By default, source files are searched in the current directory and in the directories defined by FGLLDPATH.
2. You can provide a list of paths, using system-specific path separators.

Warnings:

1. The path separator is platform specific (**":"** on UNIX platforms and **";"** on Windows platforms).

FGLDBPATH

Purpose:

The **FGLDBPATH** environment variable contains the path to database schema files.

Notes:

1. If **FGLDBPATH** is not defined, the current directory is the default path for the database schema files.
2. You can provide a list of paths, separating the paths with a colon (":").
3. **FGLDBPATH** is only used in development.

Warnings:

1. The path separator is platform specific (**":"** on UNIX platforms and **";"** on Windows platforms).

FGLSQLDEBUG

Purpose:

The **FGLSQLDEBUG** environment variable defines the debug level for tracing SQL instructions.

Notes:

1. If **FGLSQLDEBUG** is set to a value greater than zero, you get a debug trace in the standard error channel.
2. **FGLSQLDEBUG** is only used in development.

FGLPROFILE

Purpose:

The **FGLPROFILE** environment variable defines the current configuration file to be used by the runtime system.

Notes:

1. If **FGLPROFILE** is not set, the runtime system reads entries from the default configuration file located in FGLDIR/etc/fglprofile, or from the program-specific configuration file.
2. For more information, refer to the FGLPROFILE section of this manual.

FGLWRTUMASK

Purpose:

The **FGLWRTUMASK** environment variable defines the umask for the FGLDIR/lock directory.

Notes:

1. The **FGLWRTUMASK** environment variable is used by the license manager **fglWrt**.
2. This variable defines the umask to create the **FGLDIR/lock** directory.
3. The default is 000, which creates a directory with **rwxrwxrwx** rights.

For more information, refer to the Installation section of this manual.

The FGLPROFILE configuration file

Summary:

- Basics
- FGLPROFILE Entry Syntax
- Supported Entries

See also: Programs

Basics

The runtime system uses one or more configuration files in which you can define entries to change the behavior of the programs.

There is no specific naming convention for the configuration files, however, we recommend to use an extension such as **prf**.

There are three different ways to specify the configuration file, with the following order of precedence:

1. By default, the runtime system reads the configuration file provided in FGLDIR/etc/fglprofile. This file contains all supported entries, identifies the possible values for an entry, and documents default values.
2. If the FGLPROFILE environment variable is set, the runtime system reads entries from the file specified by this environment variable.
3. If the *program-specific profile directory* contains a file with the same name as the current program, the runtime system reads the entries from that file. By default, the *program-specific profile directory* is FGLDIR/defaults. This directory can be changed with the fglrun.defaults entry in one of the previously mentioned configuration files (FGLDIR/etc/fglprofile or the one defined by the FGLPROFILE environment variable).

FGLPROFILE Entry Syntax

Purpose:

An FGLPROFILE entry is a parameter that can be changed in the configuration file.

Syntax:

```
entry = value
```

Notes:

1. *entry* identifies the name of the entry.
2. *value* is the value of the entry; it might be a numeric value, a string literal, or a Boolean value (true/false), depending on what is valid for the entry.

Warnings:

1. Entry names are converted to lower case when loaded by the runtime system. In order to avoid any confusion, it is recommended to write FGLPROFILE entry names in lower case.
2. If an entry is defined several times in the same file, the last entry found in the file is used. No error is raised.

Usage:

The entries are defined by a name composed of a list of words separated by a dot character.

By using names like *domain.sub-domain.param*, entries can be organized by domains and sub-domains.

The value can be a numeric literal, a string literal, or a Boolean (true/false).

Numeric values are composed by an optional sign, followed by digits, followed by an optional decimal point and digits:
`[-|+]{digits}[.digits]`

String values must be delimited by double quotes. The escape character is backslash:
`"characters"`

Boolean values must be either the *true* or *false* keyword:
`{true|false}`

If an entry is defined in different levels of configuration files (default/specific/program), the runtime system searches for the entry value in the following order:

1. Program (FGLDIR/default/*progname*)
2. Specific (file specified by the FGLPROFILE environment variable)
3. Default (FGLDIR/etc/fglprofile)

For more details about supported entries, see Supported Entries.

Example:

```
01 rtm.memory.cachelocals = true
02 rtm.default.logfile = "mytrace1.log"
03 rtm.installation.path = "C:\\progra~1\\fourjs\\fgl"
04 dbi.database.stores.prefetch.rows = 200
```

Supported Entries

The following table shows the a partial list of supported FGLPROFILE entries. You can find the complete usage for an entry in the corresponding documentation section referenced in the description of the entry.

Entry	Values	Default	Description
`dbi.*`	N/A	N/A	Database interface configuration. See Connections for more details.
`gui.chartable`	string	NULL	Defines the character conversion table file. See Dynamic User Interface for more details.
`gui.connection.timeout`	integer	30	Defines the timeout delay (in seconds) the runtime system waits when it establishes a connection to the front-end. After this delay the program stops with an error. See Dynamic User Interface for more details.
`gui.protocol.pingTimeout`	integer	600	Defines the timeout delay (in seconds) the runtime system waits for a front-end ping when there is no user activity. After this delay the program stops with an error. See Dynamic User Interface for more details.
`gui.server.autostart.*`	N/A	N/A	Defines automatic front-end startup parameters. See Dynamic User Interface for more details.
`fglrun.default`	string	NULL	Defines the directory where program specific

			configuration files are located. See Basics for more details.
`fglrun.ignoreLogoffEvent`	boolean	false	Defines whether the DVM ignores a CTRL_LOGOFF_EVENT on Windows platforms. See Programs for more details.
`fglrun.ignoreDebuggerEvent`	boolean	false	Defines whether the DVM ignores a SIGTRAP (Unix) or CTRL-Break (Windows) to switch into debug mode. See Debugger for more details.
`fglrun.localization.*`	N/A	N/A	Defines load parameters for localized string resource files. See Localized Strings for more details.
`fglrun.mmapDisable`	boolean	false	Memory mapping control. When set to true, memory mapping is disabled and standard memory allocation method takes place. For more details about memory mapping, run "man mmap" on UNIX. See Basics for more details.
`flm.*`	N/A	N/A	License management related entries. See installation notes for more details.
`Dialog.currentRowVisibleAfterSort`	boolean	false	Forces current row to be shown after a sort in a table. See Runtime Configuration for more details.
`Dialog.fieldOrder`	boolean	false	Defines if the intermediate field triggers must be executed when a new

			field gets the focus with a mouse click. See Runtime Configuration for more details.
`key.key-name.text`	string	N/A	Defines a label for an action defined with an ON KEY clause. *Provided for V3 compatibility only.* See Settings Key labels for more details.
`Report.aggregateZero`	boolean	false	Defines if the report aggregate functions must return zero or NULL when all values are NULL. *Provided for V3 compatibility only.* See Report Configuration for more details.

The Debugger

Summary:

- Basics
- Usage
  - Starting fglrun in debug mode
  - Stack Frames
  - Using ddd as graphical interface
  - Invoking the debugger at runtime
  - Setting a breakpoint programmatically
- Commands

See also: Programs, Tools.

Basics

The debugger is a tool built in the runtime system that allows you to stop a program before it terminates, or before it encounters a problem, so that you can locate logical and runtime errors.

Syntax:

```
fglrun -d program[.42r]
```

Notes:

1. *program* is the name of the BDL program.
2. The command-line arguments for *program* (if any) have to be passed to the "run" command later on.

Usage

Starting fglrun in debug mode

In order to use the debugger, you must start the virtual machine with the **-d** option:

```
fglrun -d myprog
```

The debugger can be used alone in the command line mode or with a graphical shell compatible with gdb such as **ddd**:

```
ddd --debugger "fglrun -d myprog"
```

The debugger supports a subset of the standard GNU C/C++ debugger called **gdb**.

In command line mode, the debugger shows the following prompt

```
(fgldb)
```

A command is a single line of input. It starts with a command name, which may be followed by arguments whose meaning depends on the command name. For example, the command `step` accepts as an argument the number of times to step:

```
(fgldb) step 5
```

You can use command abbreviations. For example, the 'step' command abbreviation is 's':

```
(fgldb) s 5
```

Possible command abbreviations are shown in the command's syntax.

A blank line as input to the debugger (pressing just the RETURN or ENTER keys) usually causes the previous command to repeat. However, commands whose unintentional repetition might cause problems will not repeat in this way.

Stack Frames

Each time your program performs a function call, information about the call is saved in a block of data called a *stack frame*. Each frame contains the data associated with one call to one function.

The stack frames are allocated in a region of memory called the *call stack*. When your program is started, the stack has only one frame, that of the function **main**. This is the initial frame, also known as the *outermost frame*. As the debugger executes your program, a new frame is made each time a function is called. When the function returns, the frame for that function call is eliminated.

The debugger assigns numbers to all existing stack frames, starting with zero for the innermost frame, one for the frame that called it, and so on upward. These numbers do not really exist in your program; they are assigned by the debugger to allow you to designate stack frames in commands.

Each time your program stops, the debugger automatically selects the currently executing frame and describes it briefly. You can use the frame command to select a different frame from the current call stack.

Using ddd as graphical interface

To use **ddd** with the debugger:

1. Create a script "**fglddd**" containing the following command; make the script executable.

```
exec ddd --debugger "fglrun -d" "$@"
```

2. Start **fglddd** as a replacement for **fglrun** when you want to debug a BDL program.

Invoking the debugger in a running instance of fglrun

Even if fglrun has not been started with the -d option, it is possible to switch the running program to debug mode: On Unix platforms, you must send a **SIGTRAP** signal to the process. On Windows, you must use the **CTRL-BREAK** key in the console window which has started the program.

Example (Unix):

```
shell 1> fglrun func
```

From another session, send the SIGTRAP signal to this process.

```
shell 2> kill -TRAP pid_of_fglrun
```

Then the instance running the program will receive this signal and enter in debugging mode. The (fgldb) prompt is displayed and waits for instructions.

```
shell 1> fglrun func
func1() at func.4gl:15
15 for g_cpt=1 to 1000000
(fgldb)
```

In production sites, you can avoid the runtime system to trap the debugger signal by setting the following FGLPROFILE entry:

```
fglrun.ignoreDebuggerEvent = true
```

Setting a breakpoint programmatically

You can set a breakpoint in the program source code with the BREAKPOINT instruction. If the program flow encounters this instruction, the program stops as if the break point was set by the break command:

```
01 MAIN
02   DEFINE i INTEGER
03   LET i=123
04   BREAKPOINT
05   DISPLAY i
06 END MAIN
```

The BREAKPOINT instruction is simply ignored when running in normal mode.

Commands

Summary of the debugger commands:

Command	Description
backtrace/where	Print a summary of how your program reached the current state (back trace of all stack frames).
break	Set a break point at the specified line or function.
call	Call a function in the program.
clear	Clear breakpoint at some specified line or function.
continue	Continue program being debugged.
define	Define a new command name.
delete	Delete some breakpoints or auto-display expressions.
disable	Disable some breakpoints.
display	Print the values of expression *EXP* each time the program stops.
down	Select and print the function called by the current function.
echo	Print the specified text.
enable	Re-activate breakpoints that have previously been disabled.
finish	Execute until selected stack frame returns.
frame	Select and print a stack frame.
help	Print list of debugger commands.
ignore	Set ignore-count of a breakpoint number N to COUNT.
info	Provide information about the status of the program.
list	List specified function or line.
next	Step program; continue with the next source code line at the same level.
output	Print the current value of the specified expression; do not include value history and do not print new-line.
print	Print the current value of the specified expression.
ptype	Print the type of a variable
quit	Exit the debugger.
run	Start the debugged program.

set	Evaluate an expression and assign the result to a variable.
source	Execute a file of debugger commands.
signal	Continue program giving it the signal specified by the argument.
step	Step program until it reaches a different source line.
tbreak	Set a temporary breakpoint.
tty	Set terminal for future runs of program being debugged.
undisplay	Cancel some expressions to be displayed when the program stops.
until	Continue running until a specified location is reached.
up	Select and print the function that called the current function.
watch	Set a watchpoint for an expression. A watchpoint stops the execution of your program whenever the value of an expression changes.
whatis	Prints the data type of a variable.

---

backtrace / where

This commands prints a summary of how your program reached the current state.

Syntax:

```
backtrace
```

Usage:

The `backtrace` command prints a summary of your program's entire stack, one line per frame. Each line in the output shows the frame number and function name.

Example:

```
(fgldb) backtrace
#1 addcount() at mymodule.4gl:6
#2 main() at mymodule.4gl:2
(fgldb)
```

Tips:

- `bt` and `where` are aliases for the `backtrace` command.

---

break

This command defines a break point to stop the program execution at a given line or function.

Syntax:

```
break [ { function | [ module : ] line } ] [ if condition ]
```

Notes:

1. *function* is a function name.
2. *module* is a specific source file.
3. *line* is a source code line.
4. *condition* is an expression evaluated dynamically.

Usage:

The `break` command sets a break point at a given position in the program.

When the program is running, the debugger stops automatically at breakpoints defined by this command.

If a *condition* is specified, the program stops at the breakpoint only if the *condition* evaluates to TRUE.

Warning: If you do not specify any location (function or line number), the breakpoint is created for the current line. For example, if you write "break if var = 1", the debugger adds a conditional breakpoint for the current line, and the program will only stop if the variable is equal to 1 when reaching the current line again.

Example:

```
(fgldb) break mymodule:5
Breakpoint 2 at 0x00000000: file mymodule.4gl, line 5.
```

---

call

This command calls a function in the program.

Syntax:

```
call function-name ( expression [ , expression [...] ] )
```

Notes:

1. *function-name* is the name of the function to call.
2. *expression* is an expression argument provided to the function after evaluation.
3. The return values are printed when the function returns.

Example:

```
01 MAIN
02   DEFINE i INTEGER
03
04   LET i = 1
05   DISPLAY i
06
07 END MAIN
08
09 FUNCTION hello ()
10     RETURN "hello", "world"
11 END FUNCTION
(fgldb) br main
Breakpoint 1 at 0x00000000: file t.4gl, line 4.
(fgldb) run
Breakpoint 1, main() at t.4gl:4
4         LET i = 1
(fgldb) call hello()
$1 =  { "hello" , "world" }
(fgldb)
```

---

clear

This command clears the breakpoint at a specified line or function.

Syntax:

```
clear [ { function | [ module : ] line } ]
```

Notes:

1. *function* is a function name.
2. *module* is a specific source file.
3. *line* is a source code line.

Usage:

With the `clear` command, you can delete breakpoints according to where they are in your program.

Use the `clear` command with no arguments to delete any breakpoints at the next instruction to be executed in the selected stack frame.

See the delete command to delete individual breakpoints by specifying their breakpoint numbers.

Example:

```
(fgldb) clear mymodule:5
Deleted breakpoint 2
(fgldb)
```

---

continue

This command continues the execution of the program after a breakpoint.

Syntax:

```
continue [ignore-count]
```

Notes:

1. *ignore-count* defines the number of times to ignore a breakpoint at this location.

Usage:

The `continue` command continues the execution of the program until the program completes normally, another breakpoint is reached, or a signal is received.

Example:

```
(fgldb) continue
   <..program output..>
Program exited normally.
```

Tips:

- `c` is an alias for the `continue` command.

---

define

This command allows you to specify a user-defined sequence of commands.

Syntax:

```
define command-name
command
```

```
[...]
end
```

Notes:

1. *command-name* is the name assigned to the command sequence.
2. *command* is a valid debugger command.
3. **end** indicates the end of the command sequence.

Usage:

The `define` command allows you to create a user-defined command by assigning a command name to a sequence of debugger commands that you specify. You may then execute the command that you defined by entering the command name at the debugger prompt.

User commands may accept up to ten arguments separated by white space.

Example:

```
(fgldb) define myinfo
> info breakpoints
> info program
> end
(fgldb)
```

---

delete

This command allows you to remove breakpoints that you have specified in your debugger session.

Syntax:

```
delete breakpoint
```

Notes:

1. *breakpoint* is the number assigned to the breakpoint by the debugger.

Usage:

The `delete` command allows you to remove breakpoints when they are no longer needed in your debugger session.

If you prefer you may disable the breakpoint instead, see the `disable` command.

Example:

```
(fgldb) delete 1
(fgldb) run
Program exited normally.
(fgldb)
```

Tips:

- `d` is an alias for the `delete` command.

---

disable

This command disables the specified breakpoint.

Syntax:

```
disable breakpoint
```

Notes:

1. *breakpoint* is the number assigned to the breakpoint by the debugger.

Usage:

The `disable` command instructs the debugger to ignore the specified breakpoint when running the program.

Use the `enable` command to re-activate the breakpoint for the current debugger session.

Example:

```
(fgldb) disable 1
(fgldb) run
Program exited normally.
(fgldb)
```

---

display

This command displays the specified expression's value each time your program stops.

Syntax:

```
display expression
```

Notes:

1. *expression* is your program's expression that you wish to examine.

Usage:

The `display` command allows you to add an expression to an automatic display list. The values of the expressions in the list are printed each time your program stops. Each expression in the list is assigned a number to identify it.

This command is useful in tracking how the values of expressions change during the program's execution.

Example:

```
(fgldb) display a
1: a = 6
(fgldb) display i
2: i = 1
(fgldb) step
2: i = 1
1: a = 6
16       for i = 1 to 10
(fgldb) step
2: i = 2
1: a = 6
17       let a = a+1
(fgldb)
```

---

down

This command selects and prints the function called by the current function, or the function specified by the frame number in the call stack.

Syntax:

```
down [frames]
```

Notes:

1. *frames* is the number of frames to move down the stack. The default is 1.

Usage:

This command moves down the call stack, to the specified frame, and prints the function identified with that frame. To print the function called by the current function, use the `down` command without an argument. See Stack Frames for a brief description of frames.

```
(fgldb) down
#0 query_cust() at custquery.4gl:22
22    CALL cleanup()
(fgldb)
```

---

echo

This command prints the specified text as prompt.

Syntax:

```
echo text
```

Notes:

1. *text* is the specific text to be output.

Usage:

The `echo` command allows you to generate exactly the output that you want. Nonprinting characters can be included in text using C escape sequences, such as '\n' to print a new-line. No new-line is printed unless you specify one. In addition to the standard C escape sequences, a backslash followed by a space stands for a space. A backslash at the end of text can be used to continue the command onto subsequent lines.

Example:

```
(fgldb) echo hello
hello (fgldb)
```

---

enable

This command enables breakpoints that have previously been disabled.

Syntax:

```
enable breakpoint
```

Notes:

1. *breakpoint* is the number assigned to the breakpoint by the debugger.

Usage:

The `enable` command allows you to re-activate a breakpoint in the current debugger session. The breakpoint must have been disabled using the `disable` command.

Example:

```
(fgldb) disable 1
(fgldb) run
Program exited normally.
(fgldb) enable 1
(fgldb) run
Breakpoint 1, at mymodule.4gl:5
```

---

finish

This command continues the execution of a program until the current function returns normally.

Syntax:

```
finish
```

Usage:

The `finish` command instructs the program to continue running until just after the function in the selected stack frame returns, and then stop.

The returned value, if any, is printed.

Example:

```
(fgldb) finish
Run till exit myfunc() at module.4gl:10
Value returned is $1 = 123
(fgldb)
```

---

frame

This command selects and prints a stack frame.

Syntax:

```
frame [ address | number ]
```

Notes:

1. *address* is the address of the frame that you wish to select.
2. *number* is the stack frame number of the frame that you wish to select.

Usage:

The `frame` command allows you to move from one stack frame to another, and to print the stack frame that you select. Each stack frame is associated with one call to one function within the currently executing program. Without an argument, the current stack frame is printed. See Stack Frames for a brief discussion of frames.

Example:

```
(fgldb) frame
#0 query_cust() at testquery.4gl:42
(fgldb)
```

---

help

This command provides information about debugger commands.

Syntax:

```
help [command]
```

Notes:

1. *command* is the name of the debugger command for which you wish information.

Usage:

The `help` command displays a short explanation of a specified command.

Enter the `help` command with no arguments to display a list of debugger commands.

Example:

```
(fgldb) help delete
Delete some breakpoints or auto-display expressions
```

---

ignore

Set the ignore-count of a breakpoint number N to COUNT.

Syntax:

```
ignore breakpoint count
```

Notes:

1. *breakpoint* is the breakpoint number.
2. *count* is the number of times the breakpoint will be ignored.

Usage:

The `ignore` command defines the number of times a breakpoint is ignored when the program flow reaches that breakpoint.

The next *count* times the breakpoint is reached, the program execution will continue, and no breakpoint condition is checked.

Tips:

1. You can specify a *count* of zero to make the breakpoint stop the next time it is reached.
2. When using the continue command to resume the execution of the program from a breakpoint, you can specify a an ignore count directly as an argument.

Example:

```
(fgldb) br main
Breakpoint 1 at 0x00000000: file t.4gl, line 4.
(fgldb) ignore 1 2
Will ignore next 2 crossings of breakpoint 1.
(fgldb) run
          1
Program exited normally.
(fgldb) run
          1
Program exited normally.
(fgldb) run
Breakpoint 1, main() at t.4gl:4
4         LET i = 1
(fgldb)
```

---

info

This command describes the current state of your program.

Syntax:

```
info { breakpoints
     | sources
```

```
| program
| variables
| locals
| files
| line { function | module:line }
}
```

Notes:

1. *function* is a function name of the program.
2. *module:line* defines a source code line in a module.

Usage:

The `info` command describes the state of your program.

- `info breakpoints` lists the breakpoints that you have set.
- `info sources` prints the names of all the source files in your program**.**
- `info program` displays the status of your program.
- `info variables` displays global variables.
- `info locals` displays the local variables of the current function.
- `info files` lists the files from which symbols were loaded.
- `info line function` prints the program addresses for the first line of the function named *function*.
- `info line module:line` prints the starting and ending addresses of the compiled code for the source line specified. See the `list` command for all the ways that you can specify the source code line.

Example:

```
(fgldb) info sources
Source files for which symbols have been read in:
mymodule.4gl, fglwinexec.4gl, fglutil.4gl, fgldialog.4gl,
fgldummy4js.4gl
(fgldb)
```

---

list

This command prints source code lines of the program being executed.

Syntax:

```
list [ function | [module:]line ]
```

Usage:

The `list` command prints source code lines of your program, by default it begins with the current line.

Example:

```
(fgldb) run
Breakpoint 1, at mymodule.4gl:5
5    call addlist()
(fgldb) list
5    call addlist()
6    call addname()
.
14  end function
(fgldb)
```

---

next

This command continues running the program by executing the next source line in the current stack frame, and then stops.

Syntax:

```
next
```

Usage:

The `next` command allows you to execute your program one line of source code at a time. The `next` command is similar to `step`, but function calls that appear within the line of code are executed without stopping. When the next line of code at the original stack level that was executing when you gave the `next` command is reached, execution stops.

After reaching a breakpoint, the `next` command can be used to examine a troublesome section of code more closely.

Example:

```
(fgldb) next
5 call addlist()
(fgldb) next
6 call addname()
(fgldb)
```

Tips:

- `n` is an alias for the `next` command.

---

output

This command prints only the value of the specified expression, suppressing any other output.

Syntax:

```
output expression
```

Notes:

1. *expression* is your program's expression that you wish to examine.

Usage:

The `output` command prints the current value of the expression and nothing else, no new-line character, no "expr=", etc.

The usual output from the debugger is suppressed, allowing you to print only the value.

Example:

```
(fgldb) output b
123(fgldb)
```

---

print

This command displays the current value of the specified expression.

Syntax:

```
print expression
```

Notes:

1. *expression* is your program's expression that you wish to examine.

Usage:

The `print` command allows you to examine the data in your program.

It evaluates and prints the value of the specified expression from your program, in a format appropriate to its data type.

Example:

```
(fgldb) print b
 $1 = 5
(fgldb)
```

Tips:

- `p` is an alias for the `print` command.

---

ptype

This command prints the data type or structure of a variable.

Syntax:

```
ptype variable-name
```

Notes:

1. *variable-name* is the name of the variable.

Example:

```
(fgldb) ptype cust_rec
type = RECORD
    cust_num INTEGER,
    cust_name VARCHAR(10),
    cust_address VARCHAR(200)
END RECORD
```

---

quit

This command terminates the debugger session.

Syntax:

```
quit
```

Usage:

The `quit` command allows you to exit the debugger.

Example:

```
(fgldb) quit
```

Tips:

- `q` is an alias for the `quit` command.

---

run

This command starts the program.

Syntax:

```
run [argument [...] ]
```

Notes:

1. *argument* is an argument to be passed to the program.

Usage:

The run command causes your program to execute until a breakpoint is reached or the program terminates normally.

Example:

```
(fgldb) run a b c
Breakpoint 1, at mymodule.4gl:3
3      call addcount()
(fgldb)
```

---

set

This command allows you to configure your debugger session and change program variable values.

Syntax:

```
set { prompt ptext
    | annotate {1|0}
    | verbose {on|off}
    | variable varname=value
    | environment envname[=value]
    }
```

Notes:

1. *ptext* is the string to which the prompt should be set.
2. *varname* is the program variable to be set to *value*.
3. *envname* is the environment variable to be set to *value*.

Usage:

The `set` command allows to change program variables and/or the environment.

`set variable` sets an program variable, to be taken into account when continuing program execution.

`set prompt` changes the prompt text. The text can be set to any string. A space is not automatically added after the prompt string, allowing you to determine whether to add a space at the end of the prompt string.

`set environment` sets an environment variable, where *value* may be any string. If the *value* parameter is omitted, the variable is set to a null value. The variable is set for your program, not for the debugger itself.

`set verbose on` forces the debugger to display additional messages about its operations, allowing you to observe that it is still working during lengthy internal operations.

`set annotate 1` switches the output format of the debugger to be more machine readable (this command is used by GUI front-ends like *ddd* or *xxgdb*)

Example:

```
(fgldb) set prompt ($)
($)
```

Warning: On Unix systems, if your SHELL variable names a shell that runs an initialization file, any variables you set in that file affect your program. You may wish to move setting of environment variables to files that are only run when you sign on, such as .login or .profile.

---

source

This command executes a file of debugger commands.

Syntax:

```
source commandfile
```

Notes:

1. *commandfile* is the name of the file containing the debugger commands.

Usage:

The `source` command allows you to execute a command file of lines that are debugger commands. The lines in the file are executed sequentially. The commands are not printed as they are executed, and any messages are not displayed. Commands are executed without asking for confirmation. An error in any command terminates execution of the command file.

Example:

Using the text file **mycommands**, which contains the single line: break 10

```
(fgldb) source mycommands
Breakpoint 2 @ 0x00000000: file mymod.4gl, line 10.
(fgldb)
```

---

signal

This command sends an INTERRUPT signal to your program.

Syntax:

```
signal signal
```

Usage:

Resume execution where your program stopped, but immediately give it the signal *signal*. *signal* can be the name or the number of a signal. For example, on many systems signal 2 and signal SIGINT are both ways of sending an interrupt signal. The `signal SIGINT` command resumes execution of your program where it has stopped, but immediately sends an INTERRUPT signal. The source line that was current when the signal was received is displayed.

Notes:

1. The current version only allows then signal `SIGINT`.

Example:

```
(fgldb) signal SIGINT
Program exited normally.
16      for i = 1 to 10
(fgldb)
```

---

step

This command continues running the program by executing the next line of source code, and then stops.

Syntax:

```
step [count]
```

Notes:

1. *count* defines the number of lines to execute before stopping.

Usage:

The `step` command allows you to "step" through your program, executing one line of source code at a time. When a function call appears within the line of code, that function is also stepped through. A common technique is to set a breakpoint prior to the section or function that is causing problems, run the program till it reaches the breakpoint, and then step through it line by line.

Example:

```
(fgldb) step
4 call addlist(a)
(fgldb)
```

Tips:

- `s` is an alias for the `step` command.

---

tbreak

This command sets a temporary breakpoint.

Syntax:

```
tbreak [ { function | [ module : ] line } ] [ if condition ]
```

Notes:

1. *function* is a function name.
2. *module* is a specific source file.
3. *line* is a source code line.
4. *condition* is an expression evaluated dynamically.

Usage:

The `tbreak` command sets a breakpoint for one stop only. The breakpoint is set in the same way as with the break command, but the breakpoint is automatically deleted after the first time your program stops there.

If a *condition* is specified, the program stops at the breakpoint only if the *condition* evaluates to TRUE.

Warning: If you do not specify any location (function or line number), the breakpoint is created for the current line. For example, if you write "tbreak if var = 1", the debugger adds a conditional breakpoint for the current line, and the program will only stop if the variable is equal to 1 when reaching the current line again.

Example:

```
(fgldb) tbreak 12
Breakpoint 2 at 0x00000000: file custmain.4gl, line 12.
(fgldb)
```

---

tty

This command resets the default program input and output for future run commands.>

Syntax:

```
tty filename
```

Notes:

1. *filename* is the file which is to be the default for program input and output.

Usage:

The tty command instructs the debugger to re-direct program input and output to the specified file for future run commands.

The re-direction is for your program only; your terminal is still used for debugger input and output.

Example:

```
(fgldb) tty /dev/ttyS0
(fgldb)
```

---

undisplay

This command cancels expressions to be displayed when the program stops.

Syntax:

```
undisplay itemnum [...]
```

Notes:

1. *itemnum* is the number of the expressions for which the display is cancelled.

Usage:

When the display command is used, each expression displayed is assigned an item number. The `undisplay` command allows you to remove expressions from the list to be displayed, using the item number to specific the expression to be removed.

Example:

```
(fgldb) step
2: i = 2
1: a = 20
9     FOR  i = 1 TO 10
(fgldb) undisplay 2
(fgldb) step
1: a = 20
10   Let cont = TRUE
(fgldb)
```

---

until

This command continues running the program until the specified location is reached.

Syntax:

```
until [ { function | [ module : ] line } ]
```

Notes:

1. *function* is a function name.
2. *module* is a specific source file.
3. *line* is a source code line.

Usage:

The `until` command continues running your program until either the specified location is reached, or the current stack frame returns. This can be used to avoid stepping through a loop more than once.

Example:

```
(fgldb) until addcount()
```

---

up

This command selects and prints the function that called this one, or the function specified by the frame number in the call stack.

Syntax:

```
up [frames]
```

Notes:

1. *frames* says how many frames up to go in the stack. The default is 1.

Usage:

The `up` command moves towards the outermost frame, to frames that have existed longer. To print the function that called the current function, use the `up` command without an argument. See Stack Frames for a brief description of frames.

Example:

```
(fgldb) up
#1 main() at customain.4gl:14
14    CALL query_cust()
(fgldb)
```

---

watch

This command sets a watchpoint for an expression. A watchpoint stops execution of your program whenever the value of an expression changes.

Syntax:

```
watch expression [boolean-expression]
```

Notes:

1. *expression* is the expression to watch.
2. *boolean-expression* is an optional boolean expression.

Usage:

The watchpoint stops the program execution when the value of the expression changes.

If *boolean-expression* is provided, the watchpoint stops the execution of the program if the expression value has changed and the *boolean-expression* evaluates to TRUE.

Warning: The watchpoint cannot be set if the program is not in the context where *expression* can be evaluated. Before using a watchpoint, you typically set a breakpoint in the function where the *expression* makes sense, then you run the program, and then you set the watchpoint. The example below illustrates this procedure.

Example:

```
01 MAIN
02   DEFINE i INTEGER
03
04   LET i = 1
05   DISPLAY i
06   LET i = 2
07   DISPLAY i
08   LET i = 3
09   DISPLAY i
10
11 END MAIN
(fgldb) break main
breakpoint 1 at 0x00000000: file test.4gl, line 4
(fgldb) run
Breakpoint 1, main() at test.4gl:4
4      LET i = 1
(fgldb) watch i if i >= 3
Watchpoint 1:  i
(fgldb) continue
          1
          2
Watchpoint 1:  i

Old value = 2
New value = 3
main() at t.4gl:9
9         DISPLAY i
(fgldb)
```

whatis

This command prints the data type of a variable.

Syntax:

```
whatis variable-name
```

Notes:

1. *variable-name* is the name of the variable.

Example:

```
(fgldb) run
Breakpoint 1, main() at t.4gl:4
4         LET i = 1
(fgldb) whatis i
type = INTEGER
(fgldb)
```

The Profiler

Summary:

- Basics
- Syntax
- Usage
  - Profiler output: Flat profile
  - Profiler output: Call graph
- Example

See also: Programs, Tools

Basics

The profiler is a tool built in the runtime system that allows you to know where your program spends time, and which function calls which function.

The profiler can help to identify pieces of your program that are slower than expected.

Syntax

```
fglrun -p program[.42r] [argument [...]]
```

Notes:

1. *program* is the name of the BDL program.
2. *argument* is a command line argument passed to the program.

Usage

In order to use the profiler, you must start the virtual machine with the **-p** option:

```
fglrun -p myprog
```

When the program ends, the profiler dumps profiling information to standard error.

Warnings:

1. Times reported by the profiler can change from one execution to the other, depending on the available system resources. You better execute the program several times to get an average time.
2. The profiler does not support parent/child recursive calls, when a child function calls its parent function (i.e. Function P calls C which calls P again). In this case the output will show negative values, because the time spend in the parent function is subtracted from the time spend in the child function.

Profiler output: Flat profile

The section "flat profile" contains the list of the functions called while the programs was running. It is presented as a five-column table.

Flat profile	
Column	**Description**
`count`	number of calls for this function
`%total`	Percentage of time spent in this function. Includes time spent in subroutines called from this function.
`%child`	Percentage of time spent in the functions called from this function.
`%self`	Percentage of time spent in this function excluding the time spent in subroutines called from this function.
`name`	Function name

Note : 100% represents the program execution time.

Profiler output: Call graph

The section "Call graph" provides for each function:

1. The functions that called it, the number of calls, and an estimation of the percentage of time spent in these functions.
2. The functions called, the number of calls, and an estimation of the time that was spent in the subroutines called from this function.

Call graph	
Name	**Description**
`index`	Each function has an index which appears at the beginning of its primary line.
`%total`	Percentage of time spent in this function. Includes time spent in subroutines called from this function.
`%self`	Percentage of time spent in this function excluding the time spent in subroutines called from this function.
`%child`	Percentage of time spent in the functions called from this function.
`calls/of`	Number of calls / Total number of calls
`name`	Function name

Output example:

```
index     %total  %self  %child  calls/of      name
...
           1.29    0.10    1.18      1/2        <-- main
          24.51    1.18   23.33      1/2        <-- fb
[4]       25.80    1.29   24.51        2        *** fc
          24.51    1.43   23.08      7/8        --> fa
```

Description:

- The function **fc** has been called two times (by **main** and **fb**) and has called the function **fa** 7 times.
- The function **fa** has been called 8 times in the program.

Example

Sample program

```
01 MAIN
02   DISPLAY "Profiler sample"
03   CALL fB()
04   CALL fC(2)
05 END MAIN
06
07 FUNCTION fA(from,n_a)
08   DEFINE n_a,i INTEGER
09   DEFINE from STRING
10   FOR i=1 TO n_a
11     DISPLAY "fA "||from||" n:"||i
12   END FOR
13 END FUNCTION
14
15 FUNCTION fB()
16   CALL fA("fB",10)
17   CALL fC(5)
18 END FUNCTION
19
20 FUNCTION fC(n_c)
21   DEFINE n_c INTEGER
22   WHILE n_c > 0
23     CALL fA("fC",2)
24     LET n_c=n_c-1
25   END WHILE
26 END FUNCTION
```

Running the profiler

```
Flat profile (order by self)
   count  %total  %child   %self name
      25    88.0     0.0    88.0  rts_display
```

```
      72     6.3     0.0     6.3  rts_Concat
       8    85.4    82.0     3.4  fa
       2    25.8    24.5     1.3  fc
       8     0.3     0.0     0.3  rts_forInit
       1    85.6    85.4     0.2  fb
       1    99.9    99.6     0.3  main
Call graph
index     %total  %self  %child  calls/of      name
           12.69   12.69    0.00      1/25       <-- main
           75.29   75.29    0.00     24/25       <-- fa
[1]        87.98   87.98    0.00        25       *** rts_display
-------------------
            6.35    6.35    0.00     72/72       <-- fa
[2]         6.35    6.35    0.00        72       *** rts_Concat
-------------------
           60.90    2.02   58.88      1/8        <-- fb
           24.51    1.43   23.08      7/8        <-- fc
[3]        85.41    3.45   81.96        8        *** fa
           75.29   75.29    0.00     24/25       --> rts_display
            6.35    6.35    0.00     72/72       --> rts_Concat
            0.33    0.33    0.00      8/8        --> rts_forInit
-------------------
            1.29    0.10    1.18      1/2        <-- main
           24.51    1.18   23.33      1/2        <-- fb
[4]        25.80    1.29   24.51        2        *** fc
           24.51    1.43   23.08      7/8        --> fa
-------------------
            0.33    0.33    0.00      8/8        <-- fa
[5]         0.33    0.33    0.00        8        *** rts_forInit
-------------------
           85.61    0.20   85.41      1/1        <-- main
[6]        85.61    0.20   85.41        1        *** fb
           24.51    1.18   23.33      1/2        --> fc
           60.90    2.02   58.88      1/8        --> fa
-------------------
           99.94    0.35   99.59      1/1        <-- <top>
[7]        99.94    0.35   99.59        1        *** main
            1.29    0.10    1.18      1/2        --> fc
           85.61    0.20   85.41      1/1        --> fb
           12.69   12.69    0.00      1/25       --> rts_display
-------------------
```

Optimization

Summary:

- Genero runtime system basics
  - Dynamic module loading
  - Objects shared by multiple programs
  - Objects shared by multiple modules
  - Objects private to a program
- Size information of a program
- Check runtime system memory leaks
- Programming Tips
  - Finding program bottlenecks with the profiler
  - Optimizing SQL statements
  - Passing small CHAR parameters to function
  - Compiler removes unused variables
  - Saving memory by splitting modules
  - Saving memory by using STRING variables
  - Saving memory by using dynamic arrays

See also: Tools, SQL Programming.

Genero runtime system basics

Dynamic module loading

A Genero program is typically constructed by linking several 42m modules together. Except when using the debugger, modules are loaded dynamically as needed. For example, when executing a CALL instruction, the runtime system checks if the module of the function is already in memory. If not, the module is first loaded, then module variables are instantiated, and then the function is called.

Objects shared by multiple programs

The p-code instructions and the constants are shared among several Genero programs running on the same machine. These elements are loaded with the system memory mapping facility, which allows multiple processes to access the same unique memory area.

Objects shared by multiple modules

By definition, global variables are visible to all modules of a program, and thus shared among all modules of the program. While global variables are an easy way to share data among multiple modules, it is not recommended that you use too many global variables.

Since version **2.00**, the data type definitions (DEFINE or RECORDs and ARRAYs) are now shared by all modules of a program instance. By data type definition we mean the type descriptions, not the data itself. This applies only to the same data types is used in different modules. In versions prior to **2.00**, all data type definitions were private to a module and required un-necessary memory. For example, when defining the same RECORD structure needing 150 bytes for its definition in 20 modules, this was - in version **1.33** - 150 x 20 = 3Kb for each process, while in version **2.00** it is only using 150 bytes.

Objects private to a program

Program objects such as global variables, module variables as well as resources used by the user interface and SQL connections and cursors, are private to a program. This implies that each of these objects requires private memory to be allocated. If memory is an issue, do not allocate unnecessary resources. For example, don't create windows / load forms or declare / prepare cursors until these are really needed by the program.

Size information of a program

When a **42m** module is loaded, the runtime system allocates memory for module components such as variables, types, constants and code.

The size information of 42m modules can be extracted by using fglrun with the **-s** option; The sizes are displayed in bytes.

When using the **-s** option on a **42r** program, fglrun searches for all the modules used by the program.

Example:

```
$ fglrun -s t.42r

== Module:  t ==
                        function    local
                            main  1024004

                  module   global   module     code    types
                       t      268   512004       92      660

== Module:  t2 ==
                        function    local
                             foo        8

                  module   global   module     code    types
                      t2      472        0       43      360

== Program globals ==
                         GLOBALS     size
                               h      201
```

```
                              garr      264
                                 v        4
                              sqlca      116
                           quit_flag       4
                            int_flag       4
                              status       4

                              TOTAL     size
                                         597

== Program totals ==
                     name   global   module     code    types
                    t.42r      597   512004      135     1020

== Program types ==
 PROGRAM TYPES                                        (types)     1020
- UNIQUE TYPES                                      - (size)      864
                                                           =      156
```

First the dvm will display each module statistics in the *Module* section: Each Module section displays the list of functions declared by the module and the size of its local variables in the column *local*. Then the module statistics are displayed:

Column	Description
`module`	The 42m module name
`global`	Size used by the global variables imported by the module.
`module`	Size used by the module variables.
`code`	Size used by the code itself (shared by all program instances running on the same machine).
`types`	Size used by data types.

The next section *Program Globals* displays all the global variables referenced by the program with their size (column *size*). The *TOTAL* line shows the total amount of memory needed by the program global variables.

The section *Program totals* shows the program totals.

The last section called *Program types* provides additional information about the memory consumed by *data types*. Identical type definitions are shared between all modules of a program. This amount of memory is showed by the line *UNIQUE TYPES*.

When several instances of the same program are started, the memory used by the *code* and *constants* is shared.

Check runtime system memory leaks

To improve the quality of the runtime system, we have implemented a memory leak checker in the runtime system.

You can enable this feature by using the **-M** or **-m** options of fglrun.

```
$ fglrun -M stores.42r
FunctionI          :       10 -      10 =      0
Module             :        3 -       3 =      0
...
FieldType          :       19 -      19 =      0
```

The **-M** option displays memory counters at the end of the program execution.

The **-m** option checks for memory leaks, and displays memory counters at the end of the program execution if leaks were found.

Each line shows the number of objects allocated, and the number of objects freed. If the difference is not zero, there is a memory leak.

If you are doing automatic regression tests, we recommend that you run all your programs with **fglrun -m** to check for memory leaks in the runtime system.

Programming Tips

This section lists some programming tips and tricks to optimize the execution of your application.

Finding program bottlenecks with the profiler

The best way to find out why a program is slow (and also, to optimize an already fast-running program), it to use the Profiler. This tool is included in the runtime system, and generates a report that shows what function in your program is the most time-consuming. For more details, see Profiler.

Optimizing SQL statements

SQL statement execution is often the code part of the program that consumes a lot of processor, disk and network resources. Therefore, it is critical to pay attention to SQL execution. Advice for this can be found in SQL Programming.

Passing small CHAR parameters to functions

In Genero, function parameters of most data types are passed by value (i.e. the value of the caller variable is copied on the stack, and then copied back into a local variable of the called function.) When large data types are used, this can introduce a performance issue.

For example, the following code defines a logging function that takes a CHAR(2000) as parameter:

```
01 FUNCTION log_msg( msg )
02   DEFINE msg CHAR(2000)
03   CALL myLogChannel.writeLine(msg)
04 END FUNCTION
```

If you call this function with a string having 19 bytes:

```
01 CALL log_msg( "Start processing..." )
```

The runtime system copies the 19 bytes string on that stack, calls the function, and then copies the value into the the **msg** local variable. When doing this, since the values in CHAR variables must always have a length matching the variable definition size, the runtime system fills the remaining 1981 bytes with blanks. Each time you call this function, 2000 bytes are copied into a buffer.

By using a VARCHAR(2000) (or a STRING) data type in this function, you optimize the execution because no trailing blanks need to be added.

Compiler removes unused variables

If you have declared a large static array without any reference to that variable in the rest of the module, you will not see the memory grow at runtime. The compiler has removed its definition from the **42m** module.

To get the defined variable in the 42m module, you must at least use it once in the source (for example, with a LET statement). Note that memory might only be allocated when reaching the lines using the variable.

Saving memory by splitting modules

As described in dynamic module loading, **42m** modules are loaded on demand. If a program only needs some independent functions of a given module, all module resources will be allocated just to call these functions. By independent, we mean functions that do not use module objects such as variables defined outside function or SQL cursors. To avoid unnecessary resource allocation, you can extract these independent functions into another module and save a lot of memory at runtime.

Additionally, it is recommended that you create **42x** libraries with the **42m** modules that belong to the same functionality group. For example, group all accounting modules together in an accounting.42x library. By doing this, programmers using the **42x** libraries are not dependent from module re-organizations.

Saving memory by using STRING variables

The CHAR and VARCHAR data types are provided to hold string data from a database column. When you define a `CHAR` or `VARCHAR` variable with a length of 1000, the runtime system must allocate the entire size, to be able to fetch SQL data directly into the internal string buffer.

To save memory, Genero BDL introduced the STRING data type. The `STRING` type is similar to `VARCHAR`, except that you don't need to specify a maximum length and the internal string buffer is allocated dynamically as needed. Thus, by default, a `STRING` variable initially requires just a bunch of bytes, and grows during the program life time, with a limitation of 65534 bytes.

A `STRING` variable should typically be used to build SQL statements dynamically, for example from a CONSTRUCT instruction. You may also use the `STRING` type for utility function parameters, to hold file names for example.

After a large `STRING` variable is used, it should be cleared with a LET or a INITIALIZE TO NULL instruction. However, this is only needed for `STRING` variables declared as global or module variables. The variables defined in functions will be automatically destroyed when the program returns from the function.

Note that Genero also introduced the base.StringBuffer build-in class, which should be used for heavy string manipulation and modifications. String data is not copied on the stack when an object of this class is passed to a function, or when the string is modified with class methods. This can have a big impact on performance when very large strings are processed.

Saving memory by using dynamic arrays

Genero FGL supports both static arrays and dynamic arrays. For compatibility reasons, static arrays must be allocated in their entirety. This can result in huge memory usage when big structures are declared, such as:

```
01 DEFINE my_big_array ARRAY[100,50] OF RECORD
02           id CHAR(200),
02           comment1 CHAR(2000),
02           comment2 CHAR(2000)
04 END RECORD
```

If possible, replace such static arrays with dynamic arrays. However, be aware that dynamic arrays have a slightly different behavior than static arrays.

Note that after using a large dynamic array, you should clean the content by using the clear() method. This will free all the memory used by the array elements. However, this is only needed for arrays declared as global or module variables. The arrays defined in functions will be automatically cleaned and destroyed when the program returns from the function.

The Preprocessor

Summary:

- Basics
- Command line syntax
- File inclusion `&include`
- Simple macro definition `&define`
- Function macro definition `&define`
- Stringification
- Concatenation
- Predefined macros
- Un-define a macro `&undef`
- Conditional compilation `&ifdef,&ifndef,&else,&endif`

See also: Programs, Tools

Basics

The preprocessor is used to transform your sources before compilation. It allows you to include other files and to define macros that will be expanded when used in the source. It behaves similar to the C Preprocessor, with some differences.

The preprocessor transforms files as follows:

- The source file is read and split into lines.
- Continued lines are merged into one long line if it is part of a preprocessor definition.
- Comments are not removed unless they appear in a macro definition.
- Each line is split into a list of lexical tokens.

The preprocessor implements the following features :

1. File inclusion
2. Conditional compilation
3. Macro definition and expansion

There are two kind of macros:

1. Simple macros
2. Function macros

where function macros look like function calls.

Compilers command line options

Preprocessor options can be used with fglcomp and fglform compilers.

File inclusion path:

```
-I path
```

The **-I** option defines a path used to search files included by the `&include` directives.

Macro definition:

```
-D identifier
```

The **-D** option defines a macro with the value 1, so that it can be used conditional directives like `&ifdef`.

Preprocessing only:

```
-E
```

By using the **-E** option, only the pre-processing phase is done by the compilers. Result is dumped in standard output.

Preprocessing options:

```
-p [nopp|noln|fglpp]
```

When using option `-p nopp`, it disables the preprocessor phase.

By using option `-p noln` with the **-E** preprocessing-only option, you can remove line number information and un-necessary empty lines.

By default, the preprocessor expects an ampersand '**&**' as preprocessor symbol for macros. The option `-p fglpp` enables the old syntax, using the sharp '**#**' as preprocessor symbol.

Examples:

```
fglcomp -E -D DEBUG -I /usr/sources/headers program.4gl

fglcomp -E -p fglpp -I /usr/sources/headers program.4gl

fglcomp -E -p nopp -I /usr/sources/headers program.4gl
```

File Inclusion

Purpose:

The `&include` directive instructs the preprocessor to include a file. The included file will be scanned and processed before continuing with the rest of the current file.

Syntax:

```
&include "filename"
```

Notes:

1. *filename* is searched first in the directory containing the current file, then in the directories listed in the include path. (-I option). The file name can be followed by spaces and comments.

Example:

File A

```
01 First line
02 &include "B"
03 Third line
```

File B

```
01 Second line
```

Output

```
01 & 1 "A"
02 First line
03 & 1 "B"
04 Second line
05 & 3 "A"
06 Third line
```

These preprocessor directives inform the compiler of its current location with special preprocessor comments, so the compiler can provide the right error message when a syntax error occurs.

The preprocessor-generated comments use the following format:

```
& number "filename"
```

where:

- *number* is the current line in the preprocessed file
- *filename* is the current file name

Recursive inclusions

Recursive inclusions are not allowed. Doing so will fail and output an error message.

The following example is incorrect:

File A

```
01 &include "B"
```

File B

```
01 HELLO
02 &include "A"
```

Output

```
01 & 1 "A"
02 & 1 "B"
03 HELLO
04 A:0: Multiple inclusion of the file 'A'.
05   included from B:2
06   included from A:1
```

But including the same file several times is allowed:

File A

```
01 &include "B"
02 &include "B"  -- correct
```

File B

```
01 HELLO
```

Output

```
01 & 1 "A"
02 & 1 "B"
03 HELLO
04 & 2 "A"
05 & 1 "B"
06 HELLO
```

Simple macro definition

A macro is identified by its name and body. As the preprocessor scans the text, it substitutes the macro body for the name identifier.

Syntax:

```
&define identifier body
```

Notes:

1. *identifier* is the name of the macro. Any valid identifier can be used.
2. *body* is any sequence of tokens until the end of the line.

After substitution, the macro definition is replaced with blank lines.

Examples:

The following example show macro substitution with 2 simple macros:

File A

```
01 &define MAX_TEST 12
02 &define HW "Hello world"
03
04 MAIN
05   DEFINE i INTEGER
06   FOR i=1 TO MAX_TEST
07     DISPLAY HW
08   END FOR
09 END MAIN
```

Output

```
01 & 1 "A"
02
03
04
05 MAIN
06   DEFINE i INTEGER
07   FOR i=1 TO 12
08     DISPLAY "Hello world"
09   END FOR
10 END MAIN
```

The macro definition can be continued on multiple lines, but when the macro is expanded, it is joined to a single line as follows:

File A

```
01 &define TABLE_VALUES 1, \
02                       2, \
03                       3
04 DISPLAY TABLE_VALUES
```

Output

```
01 & 1 "A"
02
03
04
05 DISPLAY 1, 2, 3
```

The source file is processed sequentially, so a macro takes effect at the place it has been written:

File A

```
01 DISPLAY X
02 &define X "Hello"
03 DISPLAY X
```

Output

```
01 & 1 "A"
02 DISPLAY X
03
04 DISPLAY "Hello"
```

The macro body is expanded only when the macro is applied :

File A

```
01 &define AA BB
02 &define BB 12
03 DISPLAY AA
```

Output

```
01 & 1 "A"
02
03
04 DISPLAY 12
```

- AA is first expanded to BB.
- The text is rescanned and BB is expanded to 12.
- When the macro AA is defined, BB is not known yet; but it is known when the macro AA is used.

In order to prevent infinite recursion, a macro cannot be expanded recursively.

File A

```
01 &define A B
02 &define B A
03 &define C C
04 A C
```

Output

```
01 & 1 "A"
02
03
04
05 A C
```

- A is first expanded to B.
- B is expanded to A.
- A is not expanded again as it appears in its own expansion.

- C expands to C and can not be expanded further.

Function macro definition

Function macros are macros which can take arguments.

Syntax:

```
&define identifier( arglist ) body
```

Notes:

1. *identifier* is the name of the macro. Any valid identifier can be used.
2. *body* is any sequence of tokens until the end of the line.
3. *arglist* is a list of identifiers separated with commas and optionally whitespace.
4. There must be NO space or comment between the macro name and the opening parenthesis. Otherwise the macro is not a function macro, but a simple macro.

Example:

File A

```
01 &define function_macro(a,b) a + b
02 &define simple_macro (a,b) a + b
03 function_macro( 4 , 5 )
04 simple_macro (1,2)
```

Output

```
01 & 1 "A"
02
03
04 4 + 5
05 (a,b) a + b (1,2)
```

A function macro can have an empty argument list. In this case, parenthesis are required for the macro to be expanded. As we can see in the next example, line 03 is not expanded because it there is no '()' after *foo*. The function macro cannot be applied even if it has no arguments.

File A

```
01 &define foo() yes
02 foo()
03 foo
```

Output

```
01 & 1 "A"
02
03 yes
04 foo
```

The comma separates arguments. Macro parameters containing a comma can be used with parenthesis. In the following example Line 02 has been substituted, but line 03 produced an error, because the number of parameters is incorrect.

The error message is produced on standard error.

File A

```
01 &define one_parameter(a) a
02 one_parameter((a,b))
03 one_parameter(a,b)
```

Output

```
01 & 1 "A"
02
03 (a,b)
04 one_parameter
05 t:3: Invalid number of parameters for macro one_parameter.
```

Macro arguments are completely expanded and substituted before the function macro expansion.

A macro argument can be left empty.

File A

```
01 &define two_args(a,b) a b
02 two_args(,b)
03 two_args(,)
04 two_args()
05 two_args(,,)
```

Output

```
01 & 1 "A"
02
03 b
04
05 A:4: Invalid number of parameters for macro two_args.
06 two_args
07 A:5: Invalid number of parameters for macro two_args.
08 two_args
```

Macro arguments appearing inside strings are not expanded.

File A

```
01 &define foo(x) "x"
02 foo(toto)
```

Output

```
01 & 1 "A"
02
03 "x"
```

Stringification

The stringification transformation allows you to create a string using a macro parameter. When a macro parameter is used with a preceding '#', it is replaced by a string containing the literal text of the argument. The argument is not macro expanded before the substitution.

Example:

File A

```
01 &define test(x) IF x THEN                                    \
02                   DISPLAY "Condition "||#x||" is true."  \
03                 ELSE                                         \
04                   DISPLAY "Condition "||#x||" is false." \
05                 END IF
06 test(1=2)
```

Output

```
01 & 1 "A"
02
03
04
05
06
07 IF 1=2 THEN DISPLAY "Condition "||"1=2"||" is true."
           ELSE DISPLAY "Condition "||"1=2"||" is false."
           END IF
```

Line 07 has been split on multiple lines for readability. The preprocessor output is merged on one line.

Concatenation

The operator '##' can be used to merge two tokens while expanding a macro. The two tokens on either side of each '##' are combined to create a single token.

All tokens can not be merged. Usually these tokens are identifiers, or numbers. The concatenation result produces an identifier.

Example:

File A

```
01 &define COMMAND(NAME) #NAME, NAME ## _command
02 COMMAND(quit)
```

Output

```
01 & 1 "A"
02
03 "quit", quit_command
```

Predefined macros

The preprocessor predefines 2 macros:

1. `__LINE__` expands to the current line number. Its definition changes with each new line of the code.
2. `__FILE__` expands to the name of the current file as a string constant. Ex : "subdir/file.inc"

These macros are often used to generate error messages.

An `&include` directive changes the values of `__FILE__` and `__LINE__` to correspond to the included file.

Un-defining a macro

You are allowed to un-define a macro and then redefine it with a new body.

Syntax:

```
&undef identifier
```

Usage:

If a macro is redefined without having been undefined previously, the preprocessor issues a warning and replaces the existing definition with the new one.

Example:

File A

```
01 &define HELLO "hello"
02 DISPLAY HELLO
03 &undef HELLO
04 DISPLAY HELLO
```

Output

```
01 & 1 "A"
02
03 DISPLAY "hello"
```

```
04 DISPLAY HELLO
```

Conditional compilation

Conditional processing is supported with the `&ifdef` and `&ifndef` directives.

Syntax 1:

```
&ifdef identifier
...
[&else
...]
&endif
```

Syntax 2:

```
&ifndef identifier
...
[&else
...]
&endif
```

Notes:

1. The comment following `&endif` is not required. However, it can help to match the corresponding `&ifdef` or `&ifndef`.
2. Even if the condition is evaluated to false, the content of the `&ifdef` block is still scanned and tokenized. Therefore, it must be lexically correct.
3. Sometimes it is useful to use some code if a macro is not defined. You can use `&ifndef`, that evaluates to true if the macro is not defined.

Example:

File A

```
01 &define IS_DEFINED
02 &ifdef IS_DEFINED
03 DISPLAY "The macro is defined"
04 &endif  /* IS_DEFINE */
```

Output

```
01 & 1 "A"
02
03
04 DISPLAY "The macro is defined"
05
```

File Extensions

This page describes the file extensions used by the Genero Business Development Language.

Extension	Type	Description
.4gl	Text	BDL Source Module
.42m	Binary	Compiler BDL p-code module
.per	Text	BDL Form Specification File
.42f	XML	Compiler form specification file
.42s	Binary	Localized Strings compiled file
.4st	XML	Presentation Styles resource file
.4sm	XML	StartMenu resource file
.4tm	XML	TopMenu resource file
.4tb	XML	ToolBar resource file
.4ad	XML	Action Defaults resource file
.sch	Text	Database Schema File - column types
.str	Text	Localized Strings source file
.val	Text	Database Schema File - form field attributes
.att	Text	Database Schema File - video attributes
.42r	Binary	Compiled BDL program
.42x	Binary	Compiler BDL p-code library
.msg	Text	Message Definition source file
.iem	Binary	Compiled Message Definition file

Error Messages

This pages describes Genero system errors messages. If needed, you can customize or translate these system messages to your own language. See Localization, Runtime System Messages for more details.

Number	Description
-201	**A syntax error has occurred.** *Description:* This general error message indicates mistakes in the form of an SQL statement. *Solution:* Look for missing or extra punctuation; keywords misspelled, misused, or out of sequence, or a reserved word used as an identifier.
-235	**Character column size is too big.** *Description:* The SQL statement specifies a width for a character data type that is greater than 65,534 bytes. *Solution:* If you need a column of this size, use the TEXT data type, which allows unlimited lengths. Otherwise, inspect the statement for typographical errors.
-307	**Illegal subscript.** *Description:* The substring values (two numbers in square brackets) of a character variable are incorrect. The first is less than zero or greater than the length of the column, or the second is less than the first. *Solution:* Review all uses of square brackets in the statement to find the error. Possibly the size of a column has been altered and makes a substring fail that used to work.
-363	**CURSOR not on SELECT statement.** *Description:* The cursor named in this statement (probably an OPEN) has been associated with a prepared statement that is not a SELECT statement. *Solution:* Review the program logic, especially the DECLARE for the cursor, the statement id specified in it, and the PREPARE that set up that statement. If you intended to use a cursor with an INSERT statement, you can only do that when the INSERT statement is written as part of the DECLARE statement. If you intended to execute an SQL statement, do that directly with the EXECUTE statement, not indirectly through a cursor.
-513	**Statement not available with this database server.**
-805	**Cannot open file for load.** *Description:* The input file that is specified in this LOAD statement could not be opened. *Solution:* Check the statement. Possibly a more complete pathname is needed, the file does not exist, or your account does not have read permission for the file or a directory in which it resides.

-806	**Cannot open file for unload.** *Description:* The output file that is specified in this UNLOAD statement could not be opened. *Solution:* Check the statement. Possibly a more complete pathname is needed; the file exists, but your account does not have write permission for it; or the disk is full.
-809	**SQL Syntax error has occurred.** *Description:* The INSERT statement in this LOAD/UNLOAD statement has invalid syntax. *Solution:* Review it for punctuation and use of keywords.
-846	**Number of values in load file is not equal to number of columns.** *Description:* The LOAD processor counts the delimiters in the first line of the file to determine the number of values in the load file. One delimiter must exist for each column in the table, or for each column in the list of columns if one is specified. *Solution:* Check that you specified the file that you intended and that it uses the correct delimiter character. An empty line in the text can also cause this error. If the LOAD statement does not specify a delimiter, verify that the default delimiter matches the delimiter that is used in the file. If you are in doubt about the default delimiter, specify the delimiter in the LOAD statement.
-1102	**Field name not found in form.** *Description:* A field name listed in an INPUT, INPUT ARRAY, CONSTRUCT, SCROLL or DISPLAY statement does not appear in the form specification of the screen form that is currently displayed. *Solution:* Review the program logic to ensure that the intended window is current, the intended form is displayed in it, and all the field names in the statement are spelled correctly.
-1107	**Field subscript out of bounds.** *Description:* The subscript of a screen array in an INPUT, DISPLAY, or CONSTRUCT statement is either less than 1 or greater than the number of fields in the array. *Solution:* Review the program source in conjunction with the form specification to see where the error lies.
-1108	**Record name not in form.** *Description:* The screen record that is named in an INPUT ARRAY or DISPLAY ARRAY statement does not appear in the screen form that is now displayed. *Solution:* Review the program source in conjunction with the form specification to see if the screen record names match.
-1109	**List and record field counts differ.** *Description:* The number of program variables does not agree with the number of screen fields in a CONSTRUCT, INPUT, INPUT ARRAY, DISPLAY, or DISPLAY ARRAY statement. *Solution:* Review the statement in conjunction with the form specification to see where the error lies. Common problems include a

	change in the definition of a screen record that is not reflected in every statement that uses the record, and a change in a program record that is not reflected in the form design.
-1110	**Form file not found.** *Description:* The form file that is specified in an OPEN FORM statement was not found. *Solution:* Inspect the "form-file" parameter of the statement. It should not include the file suffix .frm. However, if the form is not in the current directory, it should include a complete path to the file.
-1112	**A form is incompatible with the current BDL version. Rebuild your form.** *Description:* The form file that is specified in an OPEN FORM statement is not acceptable. Possibly it was corrupted in some way, or it was compiled with a version of the Form Compiler that is not compatible with the version of the BDL compiler that compiled this program. *Solution:* Use a current version of the Form Compiler to recompile the form specification.
-1114	**No form has been displayed.** *Description:* The current statement requires the use of a screen form. For example, DISPLAY...TO or an INPUT statement must use the fields of a form. However, the DISPLAY FORM statement has not been executed since the current window was opened. *Solution:* Review the program logic to ensure that it opens and displays a form before it tries to use a form.
-1119	**NEXT FIELD name not found in form.** *Description:* This statement (INPUT or INPUT ARRAY) contains a NEXT FIELD clause that names a field that is not defined in the form. *Solution:* Review the form and program logic. Perhaps the form has been changed, but the program has not.
-1129	**Field (*name*) in BEFORE/AFTER clause not found in form.** *Description:* This statement includes a BEFORE FIELD clause or an AFTER FIELD clause that names a field that is not defined in the form that is currently displayed. *Solution:* Review the program to ensure that the intended form was displayed, and review this statement against the form specification to ensure that existing fields are named.
-1133	**The NEXT OPTION name is not in the menu.** *Description:* This MENU statement contains a NEXT OPTION clause that names a menu-option that is not defined in the statement. *Solution:* The string that follows NEXT OPTION must be identical to one that follows a COMMAND clause in the same MENU statement. Review the statement to ensure that these clauses agree with each other.
-1140	**NEXT OPTION is a hidden option.** *Description:* The option that is named in this NEXT OPTION

	statement has previously been hidden with the HIDE OPTION statement. Because it is not visible to the user, it cannot be highlighted as the next choice. *Solution:* Use the SHOW OPTION statement to unhide the menu option.
-1141	**Cannot close window with active INPUT, DISPLAY ARRAY, or MENU statement.** *Description:* This CLOSE WINDOW statement cannot be executed because an input operation is still active in that window. The CLOSE WINDOW statement must have been contained in, or called from within, the input statement itself. *Solution:* Review the program logic, and revise it so that the statement completes before the window is closed.
-1143	**Window is already open.** *Description:* This OPEN WINDOW statement names a window that is already open. *Solution:* Review the program logic, and see whether it should contain a CLOSE WINDOW statement, or whether it should simply use a CURRENT WINDOW statement to bring the open window to the top.
-1146	**PROMPT message is too long to fit in the window.** *Description:* Although BDL truncates the output of MESSAGE and COMMENT to fit the window dimensions, it does not do so for PROMPT and the user's response. *Solution:* Reduce the length of the prompt string, or make the window larger. You could display most of the prompting text with DISPLAY and then prompt with a single space or colon.
-1150	**Window is too small to display this menu.** *Description:* The window must be at least two rows tall, and it must be wide enough to display the menu title, the longest option name, two sets of three-dot ellipses, and six spaces. Revise the program to make the window larger or to give the menu a shorter name and shorter options. *Solution:* Review the OPEN WINDOW statement for the current window in conjunction with this MENU statement.
-1168	**Command does not appear in the menu.** *Description:* The SHOW OPTION, HIDE OPTION, or NEXT OPTION statement cannot refer to an option (command) that does not exist. *Solution:* Check the spelling of the name of the option.
-1170	**The type of your terminal is unknown to the system.** *Description:* Check the setting of your TERM environment variable and the setting of your TERMCAP or TERMINFO environment variable. *Solution:* Check with your system administrator if you need help with this action.
-1202	**An attempt was made to divide by zero.**

	Description: Zero cannot be a divisor. *Solution:* Check that the divisor is not zero. In some cases, this error arises because the divisor is a character value that does not convert properly to numeric.
-1204	**Invalid year in date.** *Description:* The year in a DATE value or literal is invalid. For example, the number 0000 is not acceptable as the year. *Solution:* Check the value of year.
-1205	**Invalid month in date.** *Description:* The month in a DATE value or literal must be a one- or two-digit number from 1 to 12. *Solution:* Check the value of month.
-1206	**Invalid day in date.** *Description:* The day number in a DATE value or literal must a one- or two-digit number from 1 to 28 (or 29 in a leap year), 30, or 31, depending on the month that accompanies it. *Solution:* Check the value of day.
-1212	**Date conversion format must contain a month, day, and year component.** *Description:* When a date value is converted between internal binary format and display or entry format, a pattern directs the conversion. When conversion is done automatically, the pattern comes from the environment variable DBDATE. When it is done with an explicit call to the rfmtdate(), rdefmtdate(), or USING functions, a pattern string is passed as a parameter. In any case, the pattern string (the format of the message) must include letters that show the location of the three parts of the date: 2 or 3 letters d; 2 or 3 letters m; and either 2 or 4 letters y. *Solution:* Check the pattern string and the value of DBDATE.
-1213	**A character to numeric conversion process failed.** *Description:* A character value is being converted to numeric form for storage in a numeric column or variable. However, the character string cannot be interpreted as a number. *Solution:* Check the character string. It must not contain characters other than white space, digits, a sign, a decimal, or the letter e. Verify the parts are in the right order. If you are using NLS, the decimal character or thousands separator might be wrong for your locale.
-1214	**Value too large to fit in a SMALLINT.** *Description:* The SMALLINT data type can accept numbers with a value range from -32,767 to +32,767. *Solution:* To store numbers that are outside this range, redefine the column or variable to use INTEGER or DECIMAL type.
-1215	**Value too large to fit in an INTEGER.** *Description:* The INTEGER data type can accept numbers with a value range from -2,147,483,647 to +2,147,483,647. *Solution:* Check the other data types available, such as DECIMAL.

-1218	**String to date conversion error.** *Description:* The data value does not properly represent a date: either it has non-digits where digits are expected, an unexpected delimiter, or numbers that are too large or are inconsistent. *Solution:* Check the value being converted.
-1226	**Decimal or money value exceeds maximum precision.** *Description:* The data value has more digits to the left of the decimal point than the declaration of the variable allows. *Solution:* Revise the program to define the variable with an appropriate precision.
-1260	**It is not possible to convert between the specified types.** *Description:* Data conversion does not make sense, or is not supported. *Solution:* Possibly you referenced the wrong variable or column. Check that you have specified the data types that you intended and that literal representations of data values are correctly formatted.
-1261	**Too many digits in the first field of datetime or interval.** *Description:* The first field of a DATETIME literal must contain 1 or 2 digits (if it is not a YEAR) or else 2 or 4 digits (if it is a YEAR). The first field of an INTERVAL literal represents a count of units and can have up to 9 digits, depending on the precision that is specified in its qualifier. *Solution:* Review the DATETIME and INTERVAL literals in this statement, and correct them.
-1262	**Non-numeric character in datetime or interval.** *Description:* A DATETIME or INTERVAL literal can contain only decimal digits and the allowed delimiters: the hyphen between year, month, and day numbers; the space between day and hour; the colon between hour, minute, and second; and the decimal point between second and fraction. Any other characters, or these characters in the wrong order, produce an error. *Solution:* Check the value of the literal.
-1263	**A field in a datetime or interval is out of range.** *Description:* At least one of the fields in a datetime or interval is incorrect. *Solution:* Inspect the DATE, DATETIME, and INTERVAL literals in this statement. In a DATE or DATETIME literal, the year might be zero, the month might be other than 1 to 12, or the day might be other than 1 to 31 or inappropriate for the month. Also in a DATETIME literal, the hour might be other than 0 to 23, the minute or second might be other than 0 to 59, or the fraction might have too many digits for the specified precision.
-1264	**Extra characters at the end of a datetime or interval.** *Description:* Only spaces can follow a DATETIME or INTERVAL literal. *Solution:* Inspect this statement for missing or incorrect punctuation.

-1265	**Overflow occurred on a datetime or interval operation.** *Description:* An arithmetic operation involving a DATETIME and/or INTERVAL produced a result that cannot fit in the target variable. *Solution:* Check if the data type can hold the result of the operation. For example, extend the INTERVAL precision by using YEAR(9) or DAY(9).
-1266	**Intervals or datetimes are incompatible for the operation.** *Description:* An arithmetic operation mixes DATETIME and/or INTERVAL values that do not match. *Solution:* Check the data types of the variable used in the operation.
-1267	**The result of a datetime computation is out of range.** *Description:* In this statement, a DATETIME computation produced a value that cannot be stored. This situation can occur, for example, if a large interval is added to a DATETIME value. This error can also occur if the resultant date does not exist, such as Feb 29, 1999. *Solution:* Review the expressions in the statement and see if you can change the sequence of operations to avoid the overflow.
-1268	**Invalid datetime or interval qualifier.** *Description:* This statement contains a DATETIME or INTERVAL qualifier that is not acceptable. These qualifiers can contain only the words YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, FRACTION, and TO. A number from 1 to 5 in parentheses can follow FRACTION. *Solution:* Inspect the statement for missing punctuation and misspelled words. A common error is adding an s, as in MINUTES.
-1301	**This value is not among the valid possibilities.** *Description:* A list or range of acceptable values has been established for this column in the form-specification file. *Solution:* You must enter a value within the acceptable range.
-1302	**The two entries were not the same -- please try again.** *Description:* To guard against typographical errors, this field has been designated VERIFY in the form-specification file. You must enter the value in this field twice, identically. *Solution:* Carefully reenter the data. Alternatively, you can cancel the form entry with the Interrupt key.
-1303	**You cannot use this editing feature because a picture exists.** *Description:* This field is defined in the form-specification file with a PICTURE attribute to specify its format. *Solution:* You cannot use certain editing keys (for example, CTRL-A, CTRL-D, and CTRL-X) while you are editing such a field. Use only printable characters and backspace to enter the value.
-1304	**Error in field.** *Description:* You entered a value in this field that cannot be stored in the program variable that is meant to receive it. *Solution:* Possibly you entered a decimal number when the application provided only an integer variable, or you entered a

	character string that is longer than the application expected.
-1305	**This field requires an entered value.** *Description:* The cursor is in a form field that has been designated REQUIRED. *Solution:* You must enter some value before the cursor can move to another field. To enter a null value, type any printable character and then backspace. Alternatively, you can cancel the form entry with the Interrupt key.
-1306	**Please type again for verification.** *Description:* The cursor is in a form field that has been designated VERIFY. This procedure helps to ensure that no typographical errors occur during data entry. *Solution:* You must enter the value twice, identically, before the cursor can move to another field. Alternatively, you can cancel the form entry with the Interrupt key.
-1307	**Cannot insert another row - the input array is full.** *Description:* You are entering data into an array of records that is represented in the program by a static array of program variables. That array is now full; no place is available to store another record. *Solution:* Press the ACCEPT key to process the records that you have entered.
-1309	**There are no more rows in the direction you are going.** *Description.* You are attempting to scroll an array of records farther than it can go, either scrolling up at the top or scrolling down at the bottom of the array. Further attempts will have the same result.
-1312	**FORMS statement error number %d.** *Description.* An error occurred in the form at runtime. *Solution*. Edit your source file: go to the specified line, correct the error, and recompile the file.
-1313	**SQL statement error number %d.** *Description:* The current SQL statement returned this error code number.
-1314	**Program stopped at '*file-name*', line number *line-number*.** *Description:* At runtime an error occurred in the specified file at the specified line. No .err file is generated. *Solution:* Edit your source file, go to the specified line, correct the error, and recompile the file.
-1318	**A parameter count mismatch has occurred between the calling function and the called function.** *Description:* Either too many or too few parameters were given in the call to the function. *Solution.* The call is probably in a different source module from the called functions. Inspect the definition of the function, and check all places where it is called to ensure that they use the number of parameters that it declares.

-1320	**A function has not returned the correct number of values expected.** *Description:* A function that returns several variables has not returned the correct number of parameters. *Solution:* Check your source code and recompile.
-1321	**A validation error has occurred as a result of the VALIDATE command.** *Description:* The VALIDATE LIKE statement tests the current value of variables against rules that are stored in the syscolval table. It has detected a mismatch. *Solution:* Ordinarily, the program would use the WHENEVER statement to trap this error and display or correct the erroneous values. Inspect the VALIDATE statement to see which variables were being tested and find out why they were wrong.
-1322	**A report output file cannot be opened.** *Description:* The file that the REPORT TO statement specifies cannot be opened. *Solution:* Check that your account has permission to write such a file, that the disk is not full, and that you have not exceeded some limit on the number of open files.
-1323	**A report output pipe cannot be opened.** *Description:* The pipe that the REPORT TO PIPE statement specifies could not be started. *Solution:* Check that all programs that are named in it exist and are accessible from your execution path. Also look for operating-system messages that might give more specific errors.
-1324	**A report output file cannot be written to.** *Description:* The file that the REPORT TO statement specifies was opened, but an error occurred while writing to it. *Solution:* Possibly the disk is full. Look for operating- system messages that might give more information.
-1326	**An array variable has been referenced outside of its specified dimensions.** *Description:* The subscript expression for an array has produced a number that is either less than one or greater than the number of elements in the array. *Solution:* Review the program logic that leads up to this statement to determine how the error was made.
-1327	**An insert statement could not be prepared for inserting rows into a temporary table used for a report.** *Description:* Within the report function, BDL generated an SQL statement to save rows into a temporary table. The dynamic preparation of the statement (see the reference material on the PREPARE statement) produced an error. *Solution:* Probably the database tables are not defined now, at execution time, as they were when the program was compiled. Either

	the database has been changed, or the program has selected a different database than the one that was current during compilation. Possibly the database administrator has revoked SELECT privilege from you for one or more of the tables that the report uses. Look for other error messages that might give more details.
-1328	**A temporary table needed for a report could not be created in the selected database.** *Description:* Within the report definition, BDL generated an SQL statement to save rows into a temporary table, but the temporary table could not be created. *Solution:* You must have permission to create tables in the selected database, and there must be sufficient disk space left in the database. You may already have a table in your current database with the same name as the temporary table that the report definition is attempting to create as a sorting table; the sorting table is named "t_*reportname*". Another possible cause with some database servers is that you have exceeded an operating-system limit on open files.
-1329	**A database index could not be created for a temporary database table needed for a report.** *Description:* Within the report definition, BDL generated SQL statements to save rows into a temporary table. However, an index could not be created on the temporary table. *Solution:* Probably an index with the same name already exists in the database. (The sorting index is named "i_reportname"; for example, "i_order_rpt".) Possibly no disk space is available in the file system or dbspace. Another possibility with some database servers is that you have exceeded an operating-system limit on open files.
-1330	**A row could not be inserted into a temporary report table.** *Description:* Within the report definition, BDL generated SQL statements that would save rows into a temporary table. However, an error occurred while rows were being inserted. *Solution:* Probably no disk space is left in the database. Look for other error messages that might give more details.
-1331	**A row could not be fetched from a temporary report table.** *Description:* Within the report definition, BDL generated SQL statements to select rows from a temporary table. The table was built successfully but now an error occurred while rows were being retrieved from it. *Solution:* Almost the only possible cause is a hardware failure or an error in the database server. Check for operating-system messages that might give more details.
-1332	**A character variable has referenced subscripts that are out of range.** *Description:* In the current statement, a variable that is used in taking a substring of a character value contains a number less than one or a number greater than the size of the variable, or the first substring expression is larger than the second.

	Solution: Review the program logic that leads up to this statement to find the cause of the error.
-1335	**A report is accepting output or being finished before it has been started.** *Description:* The program executed an OUTPUT TO REPORT or FINISH REPORT statement before it executed a START REPORT. *Solution:* Review the program logic that leads up to this statement to find the cause of the error.
-1337	**The variable *variable-name* has been redefined with a different type or length.** *Description:* The variable that is shown is defined in the GLOBALS section of two or more modules, but it is defined differently in some modules than in others. *Solutions.* Possibly modules were compiled at different times, with some change to the common GLOBALS file between. Possibly the variable is declared as a module variable in some module that does not include the GLOBALS file.
-1338	**The function '*function-name*' has not been defined in any module in the program.** *Description:* The named function is called from at least one module of the program, but it is defined in none. *Solution.* Verify that the module containing the function is a part of the program, and that the function name is correctly spelled.
-1340	**The error log has not been started.** *Description:* The program called the errorlog() function without first calling the startlog() function. *Solution:* Review the program logic to find out the cause of this error.
-1353	**Use '!' to edit TEXT and BYTE fields.** *This is a normal message text used outside an error context.*
-1355	**Cannot build temporary file.** *Description:* A TEXT or BYTE variable has been located in a temporary file using the LOCATE statement. The current statement assigns a value into that variable, so BDL attempted to create the temporary file, but an error occurred. *Solution:* Possibly no disk space is available, or your account does not have permission to create a temporary file. Look for operating-system error messages that might give more information.
-1359	**Read error on blob file '*file-name*'.** *Description:* The operating system signaled an error during output to a temporary file in which a TEXT or BYTE variable was being saved. *Solution:* Possibly the disk is full, or a hardware failure occurred. For more information, look for operating-system messages.
-1360	**No PROGRAM= clause for this field.** *Description.* No external program has been designated for this field using the PROGRAM attribute in the form-specification file (For Text

	User Interface mode on ASCII terminals only)
-1373	**The field '*field-name*' is not in the list of fields in the CONSTRUCT/INPUT statement.** *Description:* The built-in function get_fldbuf() or field_touched() has been called with the field name shown. However, input from that field was not requested in this CONSTRUCT or INPUT statement. As a result, the function cannot return any useful value. *Solution:* Review all uses of these functions, and compare them to the list of fields at the beginning of the statement.
-1374	**SQL character truncation or transaction warning.** *Description:* The program set WHENEVER WARNING STOP, and a warning condition arose. If the statement involved is a DATABASE statement, the condition is that the database that was just opened uses a transaction log. On any other statement, the condition is that a character value from the database had to be truncated to fit in its destination.
-1375	**SQL NULL value in aggregate or mode ANSI database warning.** *Description:* The program set WHENEVER WARNING STOP, and a warning condition arose. If the statement that is involved is a DATABASE statement, the condition is that the database that was just opened is ANSI compliant. On any other statement, the condition is that a null value has been used in the computation of an aggregate value.
-1376	**SQL, database server, or program variable mismatch warning.** *Description:* The program set WHENEVER WARNING STOP, and a warning condition arose. If the statement that is involved is a DATABASE or CREATE DATABASE statement, the condition is that the database server opened the database. On any other statement, the condition is that a SELECT statement returned more values than there were program variables to contain them.
-1377	**SQL float-to-decimal conversion warning.** *Description:* The program set WHENEVER WARNING STOP, and a warning condition arose. The condition is that in the database that was just opened, the database server will use the DECIMAL data type for FLOAT values.
-1378	**SQL non-ANSI extension warning.** *Description.* A database operation was performed that is not part of ANSI SQL, although the current database is ANSI compliant. This message is informational only.
-1396	**A report PRINT FILE source file cannot be opened for reading.** *Description:* The file that is named in a PRINT FILE statement cannot be opened. *Solution:* Review the file name. If it is not in the current directory, you must specify the full path. If the file exists, make sure your account has permissions to read it.

-2017	**The character data value does not convert correctly to the field type.** *Description:* You have entered a character value (a quoted string) into a field that has a different data type (for example INTEGER). However, the characters that you entered cannot be converted to the type of the field. *Solution:* Re-enter the data.
-2024	**There is already a record '*record-name*' specified.** *Description:* A screen record is automatically defined for each table that is used in the ATTRIBUTES section to define a field. If you define a record with the name of a table, it is seen as a duplicate. *Solution:* Check that the record-name of every screen record and screen array is unique in the form specification.
-2028	**The symbol '*name*' does not represent a table prefix used in this form.** *Description:* In a SCREEN RECORD statement, each component must be introduced by the name of the table as defined in the TABLES section or by the word FORMONLY. *Solution:* Review the spelling of the indicated name against the TABLES section, and check the punctuation of the rest of the statement.
-2029	**Screen record array '*record-name*' has different component sizes.** *Description:* The screen record array name has component sizes which either differ from the specified dimension of the array or differ among themselves. This error message appears when one or more of the columns appear a different number of times. *Solution.* The dimension of the screen array is written in square brackets that follow its name. Verify that the dimensions of the screen array match the screen fields.
-2039	**The attributes AUTONEXT, DEFAULT, INCLUDE, VERIFY, RIGHT and ZEROFILL are not supported for BLOB fields.** *Description:* Columns of the data type specified cannot be used in the ways that these attributes imply. *Solution:* Check that the table and column names are as you intended, and verify the current definition of the table in the database that the DATABASE statement names.
-2041	**The form '*form-name*' cannot be opened.** *Description:* The form filename cannot be opened. This is probably because it does not exist, or the user does not have read permission. *Solution:* Check the spelling of filename. Check that the form file exists in your current directory. If it is in another directory, check that the correct pathname has been provided. On a UNIX system, if these things are correct, verify that your account has read permission on the file.
-2045	**The conditional attributes of a field cannot depend on the values**

	of other fields. *Description:* The Boolean expression in a WHERE clause of a COLOR attribute can use only the name of that field and constants. *Solution:* Revise this attribute, and recompile the form.
-2100	**Field '*field-name*' has validation string error, String = *string*.** *Description:* One of the formatting or validation strings that is stored in the syscolval or syscolatt tables is improperly coded. The string is shown as is the field to which it applies. *Solution:* Update the string in the tables.
-2810	**The name '*database-name*' is not an existing database name.** *Description:* This name, which was found in the DATABASE statement at the start of the form specification, is not a database that can be found. *Solution:* Check the spelling of the database name and the database entries in the fglprofile file.
-2820	**The label name between brackets is incorrectly given or the label is missing.** *Description:* In the layout section of a form specification, the brackets should contain a simple name. Instead, they contain spaces or an invalid name. *Solution:* Check the layout section of the form for invalid form item labels.
-2830	**A left square bracket has been found on this line, with no right square bracket to match it.** *Description:* Every left square bracket field delimiter must have a right square bracket delimiter on the same line. *Solution:* Review the form definition file to make sure all fields are properly marked.
-2840	**The field label '*label-name*' was not defined in the form.** *Description:* The indicated name appears at the left of this ATTRIBUTES statement, but it does not appear within brackets in the SCREEN section. *Solution:* Review the field tags that have been defined to see why this one was omitted.
-2843	**The column '*column-name*' does not appear in the form specification.** *Description:* A name in this ATTRIBUTES statement should have been defined previously in the form specification. *Solution:* Check that all names in the statement are spelled correctly and defined properly.
-2846	**The field '*field-name*' is not a member of the table '*table-name*'.** *Description:* Something in this statement suggests that the name shown is part of this table, but that is not true in the current database. *Solution:* Review the spelling of the two names. If they are as you intended, check that the correct database is in use and that the table has not been altered.

-2859	**The column '*column-name*' is a member of more then one table -- you must specify the table name.** *Description:* Two or more tables that are named in the TABLES section have columns with the name shown. *Solution:* You must make clear which table you mean. To do this, write the table name as a prefix of the column name, as table.column, wherever this name is used in the form specification.
-2860	**There is a column/value type mismatch for '*column-name*'.** *Description:* This statement assigns a value to the field with the DEFAULT clause or uses its value with the INCLUDE clause, but it does so with data that does not agree with the data type of the field. *Solution:* Review the data type of the field (which comes from the column with which it is associated), and make sure that only compatible values are assigned.
-2862	**The table '*table-name*' cannot be found in the database.** *Description:* The indicated table does not exist in the database that is named in the form. *Solution:* Check the spelling of the table name and database name. If they are as you intended, either you are not using the version of the database that you expected, or the database has been changed.
-2863	**The column '*column-name*' does not exist among the specified tables.** *Description:* The tables that are specified in the TABLES section of the form exist, but column-name, which is named in the ATTRIBUTES section, does not. *Solution:* Check its spelling against the actual table. Possibly the table was altered, or the column was renamed.
-2864	**The table '*table-name*' is not among the specified tables.** *Description:* The indicated table is used in this statement but is not defined in the TABLES section of the form specification. *Solution:* Check its spelling; if it is as you intended, add the table in the TABLES section.
-2865	**The column '*column-name*' does not exist in the table '*table-name*'.** *Description:* Something in this statement implies that the column shown is part of the indicated table (most likely the statement refers to table-name.column). However, it is not defined in that table. *Solution:* Check the spelling of both names. If they are as you intended, then check the contents of the database; possibly the table has been altered or the column renamed.
-2892	**The column '*column-name*' appears more then once. If you wish a column to be duplicated in a form, use the same display field label.** *Description:* The same column name is listed in the ATTRIBUTES section more than once. *Solution:* The expected way to display the same column in two or

	more places is to put two or more fields in the screen layout, each with the same tag-name. Then put a single statement in the ATTRIBUTES section to associate that tag-name with the column name. The current column value will be duplicated in all fields. If you intended to display different columns that happen to have the same column-names, prefix each column with its table-name.
-2893	**The display field label '*label-name*' appears more than once in this form, but the lengths are different.** *Description:* You can put multiple copies of a field in the screen layout (all will display the same column), but all copies must be the same length. *Solution:* Review the form definition to make sure that, if you intended to have multiple copies of one field, all copies are the same.
-2975	**The display field label '*label-name*' has not been used.** *Description:* A field tag has been declared in the screen section of the form- specification file but is not defined in the attributes section. *Solution.* Check your form-specification file.
-2992	**The display label '*label-name*' has already been used.** *Description:* The forms compiler indicates that name has been defined twice. These names must be defined uniquely in the form specification. *Solution:* Review all uses of the name to see if one of them is incorrect.
-2997	**See error number %d.** *Description:* The database server returned an error that is shown. *Solution:* Look up the shown error in the database server documentation.
-4307	**The number of variables and/or constants in the display list does not match the number of form fields in the display destination.** *Description:* There must be exactly as many items in the list of values to display as there are fields listed following the TO keyword in this statement. *Solution:* Review the statement.
-4308	**The number of input variables does not match the number of form fields in the screen input list.** *Description:* Your INPUT statement must specify the same number of variables as it does fields. *Solution:* When checking this, keep in mind that when you refer to a record using an asterisk or THRU, it is the same as listing each record component individually.
-4309	**Printing cannot be done within a loop or CASE statement contained in report headers or trailers.** *Description:* BDL needs to know how many lines of space will be devoted to page headers and trailers; otherwise, it does not know how many detail rows to allow on a page. Since it cannot predict how

	many times a loop will be executed, or which branch of a CASE will be execute, it forbids the use of PRINT in these contexts within FIRST PAGE HEADER, PAGE HEADER, and PAGE TRAILER sections. *Solution:* Re-arrange the code to place the PRINT statement where it will always be executed.
-4319	**The symbol '*name*' has been defined more than once.** *Description:* The variable that is shown has appeared in at least one other DEFINE statement before this one. *Solution:* Review your code. If this DEFINE is within a function or the MAIN section, the prior one is also. If this DEFINE is outside any function, the prior one is also outside any function; however, it might be within the file included by the GLOBALS statement.
-4320	**The symbol '*name*' is not the name of a table in the specified database.** *Description:* The named table does not appear in the database. *Solution:* Review the statement. The table name may be spelled wrong in the program, or the table might have been dropped or renamed since the last time the program was compiled.
-4322	**The symbol '*name*' is not the name of a column in the specified database.** *Description:* The preceding statement suggests that the named column is part of a certain table in the specified database. The table exists, but the column does not appear in it. *Solution:* Check the spelling of the column name. If it is spelled as you intended, then either the table has been altered, or the column renamed, or you are not accessing the database you expected.
-4323	**The variable '*variable-name*' is too complex a type to be used in an assignment statement.** *Description:* The named variable is a complex variable like a record or an array, which cannot be used in a LET statement. *Solution:* You must assign groups of components to groups of components using asterisk notation.
-4324	**The variable '*variable-name*' is not a character type, and cannot be used to contain the result of concatenation.** *Description:* This statement attempts to concatenate two or more character strings (using the comma as the concatenation operator) and assign the result to the named variable. Unfortunately, it is not a character variable, and automatic conversion from characters cannot be performed in this case. *Solution:* Assign the concatenated string to a character variable; then, if you want to treat the result as numeric, assign the string as a whole to a numeric variable.
-4325	**The source and destination records in this record assignment statement are not compatible in types and/or length.** *Description:* This statement uses asterisk notation to assign all

	components of one record to the corresponding components of another. However, the components do not correspond. Note that BDL matches record components strictly by position, the first to the first, second to second, and so on; it does not match them by name. *Solution:* If the source and destination records do not have the same number and type of components, you will have to write a simple assignment statement for each component.
-4333	**The function '*function-name*' has already been called with a different number of parameters.** *Description:* Earlier in the program, there is a call to this same function or event with a different number of parameters in the parameter list. At least one of these calls must be in error. *Solution:* Examine the FUNCTION statement for the named function to find out the correct number of parameters. Then examine all calls to it, and make sure that they are written correctly.
-4334	**The variable '*variable-name*' in its current form is too complex to be used in this statement.** *Description:* The variable has too many component parts. Only simple variables (those that have a single component) can be used in this statement. *Solution:* If variable-name is an array, you must provide a subscript to select just one element. If it is a record, you must choose just one of its components. (However, if this statement permits a list of variables, as in the INITIALIZE statement, you can use asterisk or THRU notation to convert a record name into a list of components)
-4336	**The parameter '*param-name*' has not been defined within the function or report.** *Description:* The name variable-name appears in the parameter list of the FUNCTION statement for this function. However, it does not appear in a DEFINE statement within the function. All parameters must be defined in their function before use. *Solution:* Review your code. Possibly you wrote a DEFINE statement but did not spell variable-name the same way in both places.
-4338	**The symbol '*name*' has already been defined once as a parameter.** *Description:* The name that is shown appears in the parameter list of the FUNCTION statement and in at least two DEFINE statements within the function body. *Solution:* Review your code. Only one appearance in a DEFINE statement is permitted.
-4340	**The variable '*variable-name*' is too complex a type to be used in an expression.** *Description:* In an expression, only simple variables (those that have a single component) can be used. *Solution:* If the variable indicated is an array, you must provide a subscript to select just one element. If it is a record or object, you must choose just one of its components.

-4343	**Subscripting cannot be applied to the variable '*variable-name*'.** *Description:* You tried to use a [x,y] subscript expression with a variable that is neither a character data type or an array type. *Solution:* Check the variable data type and make sure it can be used with a subscript expression.
-4347	**The variable '*variable-name*' is not a record. It cannot reference record elements.** *Description:* In this statement variable-name appears followed by a dot, followed by another name. This is the way you would refer to a component of a record variable; however, variable-name is not defined as a record. *Solution:* Either you have written the name of the wrong variable, or else variable-name is not defined the way you intended.
-4363	**The report cannot skip lines while in a loop within a header or trailer.** *Description.* BDL needs to know how many lines of space will be devoted to the page header and trailer (otherwise it does not know how many detail rows to allow on the page). It cannot predict how many times a loop will be executed, so it has to forbid the use of SKIP statements in loops in the PAGE HEADER, PAGE TRAILER, and FIRST PAGE HEADER sections. *Solution:* Review the report header or trailer to avoid SKIP in loops.
-4369	**The symbol '*name*' does not represent a defined variable.** *Description:* The name shown appears where a variable would be expected, but it does not match any variable name in a DEFINE statement that applies to this context. *Solution:* Check the spelling of the name. If it is the name you intended, look back and find out why it has not yet been defined. Possibly the GLOBALS statement has been omitted from this source module, or it names an incorrect file. Possibly this code has been copied from another module or another function, but the DEFINE statement was not copied also.
-4371	**Cursors must be uniquely declared within one program module.** *Description:* In the statement DECLARE cursor-name CURSOR, the identifier cursor-name can be used in only one DECLARE statement in the source file. This is true even when the DECLARE statement appears inside a function. Although a program variable made with the DEFINE statement is local to the function, a cursor within a function is still global to the whole module *Solution:* Search for duplicated cursor names and change the name to have unique identifiers.
-4372	**The cursor '*cursor-name*' has not yet been declared in this program.** *Description:* The name shown appears where the name of a declared cursor or a prepared statement is expected; however, no cursor (or statement) of that name has been declared (or prepared) up to this point in the program.

	Solution: Check the spelling of the name. If it is the name you intended, look back in the program to see why it has not been declared. Possibly the DECLARE statement appears in a GLOBALS file that was not included.
-4374	**This type of statement can only be used within a MENU statement.** *Description:* This statement only makes sense within the context of a MENU statement. *Solution:* Review the program in this vicinity to see if an END MENU statement has been misplaced. If you intended to set up the appearance of a menu before displaying it, use a BEFORE MENU block within the scope of the MENU.
-4379	**The input file '*file-name*' cannot be opened.** *Description:* Either the file does not exist, or, on UNIX, your account does not have permission to read it. *Solution:* Possibly the filename is misspelled, or the directory path leading to the file was specified incorrectly.
-4380	**The listing file '*file-name*' cannot be created.** *Description:* The file cannot be created. *Solution:* Check that the directory path leading to the file is specified correctly and, on UNIX systems, that your account has permission to create a file in that directory. Look for other, more explicit, error messages from the operating system. Possibly the disk is full, or you have reached a limit on the number of open files.
-4383	**The elements '*name-1*' and '*name-2*' do not belong to the same record.** *Description:* The two names shown are used where two components of one record are required; however, they are not components of the same record. *Solution:* Check the spelling of both names. If they are spelled as you intended, go back to the definition of the record and see why it does not include both names as component fields.
-4402	**In this type of statement, subscripting may be applied only to array.** *Description:* The statement contains a name followed by square brackets, but the name is not that of an array variable. *Solution.* Check the punctuation of the statement and the spelling of all names. Names that are subscripted must be arrays. If you intended to use a character substring in this statement, you will have to revise the program.
-4403	**The number of dimensions for the variable '*variable-name*' does not match the number of subscripts.** *Description:* In this statement, the array whose name is shown is subscripted by a different number of dimensions than it was defined to have. *Solution:* Check the punctuation of the subscript. If it is as you

	intended, then review the DEFINE statement where variable-name is defined.
-4414	**The label *label-name* has been used but has never been defined within the above main program or function.** *Description:* A GOTO or WHENEVER statement refers to the label shown, but there is no corresponding LABEL statement in the current function or main program. *Solution:* Check the spelling of the label. If it is as you intended it, find and inspect the LABEL statement that should define it. You cannot transfer out of a program block with GOTO; labels must be defined in the same function body where they are used.
-4415	**An ORDER BY or GROUP item specified within a report must be one of the report parameters.** *Description:* The names used in a ORDER BY, AFTER GROUP OF, or BEFORE GROUP OF statement must also appear in the parameter list of the REPORT statement. It is not possible to order or group based on a global variable or other expression. *Solution:* Check the spelling of the names in the statement and compare them to the REPORT statement.
-4416	**There is an error in the validation string: '*string*'.** *Description:* The validation string in the syscolval table is not correct. *Solution:* Change the appropriate DEFAULT or INCLUDE value in the syscolval table.
-4417	**This type of statement can be used only in a report.** *Description:* Statements such as PRINT, SKIP, or NEED are meaningful only within the body of a report function, where there is an implicit report listing to receive output. *Solution:* Remove the report specific statement from the code which is not in a report body.
-4418	**The variable used in the INPUT ARRAY or DISPLAY ARRAY statement must be an array.** *Description:* The name following the words DISPLAY ARRAY or INPUT ARRAY must be that of an array of records. *Solution:* Check the spelling of the name. If it is as you intended, find and inspect the DEFINE statement to see why it is not an array. (If you want to display or input a simple variable or a single element of an array, use the DISPLAY or INPUT statement.)
-4420	**The number of lines printed in the IF part of an IF-THEN-ELSE statement of a header or trailer clause must equal the number of lines printed in the ELSE part.** *Description.* BDL needs to know how many lines will be filled in header and trailer sections (otherwise it could not know how many detail rows to put on the page). Because it cannot tell which part of an IF statement will be executed, it requires that both produce the same number of lines of output. *Solution:* Use the same number of occurrences of PRINT statements

	in each block of the IF statement.
-4440	**The field '*field-name-1*' precedes '*field-name-2*' in the record '*record-name*' and must also precede it when used with the THROUGH shorthand.** *Description:* The THROUGH or THRU shorthand requires you to give the starting and ending fields as they appear in physical sequence in the record. *Solution:* Check the spelling of the names; if they are as you intended, then refer to the VARIABLE statement where the record was defined to see why they are not in the sequence you expected.
-4447	**'*key-name*' is not a recognized key value.** *Description:* The key name used in an ON KEY clause is not known by the compiler. *Solution:* Search the documentation for possible key names (F1-F255, Control-?).
-4448	**Cannot open the file '*file-name*' for reading or writing.** *Description:* The file cannot be opened. *Solution:* Verify that the filename is correctly spelled and that your account has permission to read or write to it.
-4452	**The function (or report) '*name*' has already been defined.** *Description:* Each function (or report, which is similar to a function) must have a unique name within the program. *Solution:* Change the function or report name.
-4457	**You may have at most 4 keys in the list.** *Description:* An interactive instruction defines a ON KEY() clause with more that 4 keys. *Solution:* Remove keys from the list.
-4458	**One dimension of this array has exceeded the limit of 65535.** *Description:* The program is using a static array with a dimension that exceeds the limit. *Solution:* Use a dimension below the 65535 limit.
-4463	**The NEXT FIELD statement can only be used within an INPUT or CONSTRUCT statement.** *Description:* The NEXT FIELD statement is used outside an INPUT, INPUT ARRAY or CONSTRUCT statement. *Solution:* Remove the NEXT FIELD statement from that part of the code.
-4464	**The number of columns must match the number of values in the SET clause of an UPDATE statement.** *Description:* In an UPDATE statement, the number of values used does not match the number of columns. *Solution:* Check for the table definition, then either add or remove values or columns from the UPDATE statement.
-4476	**Record members may not be used with database column substring.**

	Description: This statement has a reference of the form name1.name2[...]. This is the form in which you would refer to a substring of a column: table.column[...]. However, the names are not a table and column in the database, so BDL presumes they refer to a field of a record. *Solution:* Inspect the statement and determine what was intended: a reference to a column or to a record. If it is a column reference, verify the names of the table and column in the database. If it is a record reference, verify that the record and component are properly defined.
-4477	**The variable '*variable-name*' is an array. You must specify one of its elements in this statement.** *Description:* You tried to use an array without element specification in a SQL statement. *Solution*. Use one of the members of the array.
-4488	**The program cannot CONTINUE or EXIT *statement-type* at this point because it is not immediately within *statement-type* statement.** *Description:* This CONTINUE or EXIT statement is not appropriate in its context. *Solution:* Review your code. Possibly the statement is misplaced, or the statement type was specified incorrectly.
-4490	**You cannot have multiple BEFORE clauses for the same field.** *Description:* You cannot specify more than one BEFORE FIELD clause for the same field. *Solution:* Review your code to eliminate multiple BEFORE FIELD clauses.
-4491	**You cannot have multiple AFTER clauses for the same field.** *Description:* You cannot specify more than one AFTER FIELD clause for the same field. *Solution:* Review your code to eliminate multiple AFTER FIELD clauses.
-4631	**Startfield of DATETIME or INTERVAL qualifiers must come earlier in the time-list than its endfield.** *Description:* The qualifier for a DATETIME or INTERVAL consists of start TO end, where the start and end are chosen from this list: YEAR MONTH DAY HOUR MINUTE SECOND FRACTION. The keyword for the start field must come earlier in the list than, or be the same as, the keyword for the end field. *Solution:* Check the order of the startfield and endfield qualifiers. For example, qualifiers of DAY TO FRACTION and MONTH TO MONTH are valid but one of MINUTE TO HOUR is not.
-4632	**Parenthetical precision of FRACTION must be between 1 and 5. No precision can be specified for other time units.** *Description:* In a DATETIME qualifier only the FRACTION field may have a precision in parentheses, and it must be a single digit from 1 to 5.

	Solution: Check the DATETIME qualifiers in the current statement; one of them violates these rules. The first field of an INTERVAL qualifier may also have a parenthesized precision from 1 to 5.
-4652	**The function '*function-name*' can only be used within an INPUT or CONSTRUCT statement.** *Description:* The function shown is being used outside of an INPUT or CONSTRUCT statement. However, it returns a result that is only meaningful in the context of INPUT or CONSTRUCT. *Solution:* Review the code to make sure that an END INPUT or END CONSTRUCT statement has not been misplaced. Review the operation and use of the function to make sure you understand it.
-4653	**No more than one BEFORE or AFTER INPUT/CONSTRUCT clause can appear in an INPUT/CONSTRUCT statement.** *Description.* There may be only one BEFORE block of statements to initialize each of these statement types. *Solution:* Make sure that the scope of all your INPUT, CONSTRUCT and MENU statements is correctly marked with END statements. Then combine all the preparation code into a single BEFORE block for each one.
-4656	**CANCEL INSERT can only be used in the BEFORE INSERT clause of an INPUT ARRAY statement.** *Description:* The CANCEL INSERT statement is being used outside of the BEFORE INSERT clause of an INPUT ARRAY. *Solution:* Review the code to make sure that CANCEL INSERT has not been used anywhere except in the BEFORE INSERT clause.
-4657	**CANCEL DELETE can only be used in the BEFORE DELETE clause of an INPUT ARRAY statement.** *Description:* The CANCEL DELETE statement is being used outside of BEFORE DELETE clause of an INPUT ARRAY. *Solution:* Review the code to make sure that CANCEL DELETE has not been used anywhere except in the BEFORE DELETE clause.
-4900	**This syntax is not supported here. Use [screenrecordname.]screenfieldname.** *Description:* The field name specification in a BEFORE FIELD or AFTER FIELD is not valid. *Solution:* Check for the field name and use [screenrecordname.]screenfieldname syntax.
-4901	**Fatal internal error: %s(%d).** *Description:* This error occurs when an incorrect field name is used in a BEFORE FIELD or AFTER FIELD statement. *Solution:* Check if the field name used in the BEFORE FIELD or AFTER FIELD clause exists.
-6001	**The license manager daemon cannot be started.** *Description:* This error occurs when a process creation fails during the start of the license manager. *Solution:* Increase the maximum number of processes allowed

	(ulimit)
-6012	**Cannot get license information. Check your environment and the license (run 'fglWrt -a info').** See error -6015.
-6013	**Time limited version: time has expired.** *Description:* The license installed is a license with time limit and time has expired. The program can not start. *Solution:* Contact your distributor or support center.
-6014	**Your serial number is not valid for this version.** *Description:* The license serial number is invalid for this version of the software. *Solution:* Contact your distributor or support center.
-6015	**Cannot get license information. Check your environment and the license (run 'fglWrt -a info').** *Description:* It is not possible for the application to check the license validity. *Solution:* • License manager: o The license may not have been installed o The license controller can not communicate with the license manager. Check that the license manager is started and check that the fglprofile entries *flm.server* and *flm.service* contain valid information. o The directory **$FLMDIR/lock** and all the files below must have read/write permission. • License controller: o The license may not have been installed. o The directory **$FGLDIR/lock** and all the files below must have read/write permission.
-6016	**Cannot get information for license (Error %s). Check your environment and the license (run 'fglWrt -a info').** *Description:* The application is unable to check the license validity. *Solution:* See error -6015.
-6017	**User limit exceeded. Cannot run this program.** *Description:* The maximum number of users allowed by the license has been reached. The program can not start. *Solution:* Contact your distributor or support center.
-6018	**Cannot access internal data file. Cannot continue this program. Please, check your environment(%s).** *Description:* When a client computer starts an application on the server, the application stores data in the **$FGLDIR/lock** directory. The client must have permission to create and delete files in this directory.

	Solution: • Do not remove or modify files contained in the directory $FGLDIR/lock • Change the permissions of the **$FGLDIR/lock**directory, or connect to the server with a user name having the correct permissions.
-6019	**This demonstration version allows one user only.** *Description:* The demonstration version is designed to run with only one user. Another user or another graphical daemon is currently active. *Solution:* Wait until the user stops the current program, or use the same graphical daemon.
-6020	**Installation: Cannot open '*file-name*'.** *Description:* A file is missing or the permissions are not set for the current user. *Solution:* Check that the file permissions are correct for the user trying to execute the application. If the file is missing, re-install the compiler package.
-6022	**Demonstration time has expired. Please, run this program again.** *Description:* The runtime demonstration version is valid only for a few minutes after you have started a program. *Solution:* Restart the program.
-6023	**C-code generation is not allowed with the demonstration program.** *Description:* Four J's Business Development Language can compile in P code and in C Code (only for the UNIX version). But with the demonstration version, C-code compilation is not available. *Solution:* Compile your program in P code.
-6025	**Demonstration time has expired. Please, contact your vendor.** *Description:* The demonstration version of Four J's Business Development Language has a time limit of 30 days. *Solution:* Either reinstall a new demonstration version, or call your Four J's Business Development Language vendor.
-6026	**Bad link for runner demonstration. Please, retry or rebuild your runner.** *Description:* The runner is corrupted.
-6027	**Cannot access license server. Please check the following:** - the license server entry in your resource file. (service port) - the license server host. - the license server program. *Description:* You have not specified a value for the environment variable [fgllic|fls|flm].server in the $FGLDIR/etc/fglprofile file. *Solution:* Check the fglprofile file for the entry point

	[fgllic|fls|flm].server and specify the name of the computer that runs the Four J's Business Development Language License Manager.
-6029	**Unknown parameter '*parameter*' for checking.** *Description:* The command line of the fglWrt or flmprg tool contains an unknown parameter. *Solution:* Check your command-line parameters and retry the command.
-6031	**Temporary license *license-number* has expired.** *Description:* Your temporary runtime license has expired. *Solution:* Call your Four J's Business Development Language distributor to get a new license.
-6032	**%s: illegal option : '%c'.** *Description:* You are not using a valid option. *Solution:* Check your command line and try the command again.
-6033	**%s: '%c' option requires an argument.** *Description:* You cannot use this option of the fglWrt tool without a parameter. *Solution:* Check your command line and try the command again.
-6034	**Warning! This is a temporary license, installation number is '*installation-number*'.** *Description:* You have installed a temporary license of 30 days. You will have to enter an installation key before the end of this period if you want to keep on running the program. *Solution:* This is only a warning message.
-6035	**Cannot read in directory** *Description:* The compiler cannot access the $FGLDIR/lock directory. The current user must have read and write permissions in this directory. *Solution:* Give the current user read and write permissions to the $FGLDIR/lock directory.
-6041	**Can not retrieve network interface information.** *Description:* An error occurred while retrieving network interface information. *Solution:* Restart your program. If this does not solve your problem, contact your distributor.
-6042	**MAC Address has changed.** *Description:* The MAC address of the host has changed since the license was first installed. *Solution:* The license must be reinstalled, or restore the old MAC address.
-6043	**The testing period is finished. You must install a new license.** *Description:* The test time license of Four J's Business Development Language has expired. *Solution:* Call your Four J's Business Development Language distributor to purchase a new license.

-6044	**IP Address has changed.** *Description:* The IP Address of the host has changed. *Solution:* Restore the IP address of the host, or reinstall the license. This is no longer checked by the latest versions of the license controller.
-6045	**Host name has changed.** *Description:* The host name has changed. *Solution:* Restore the host name or reinstall the license. This is no longer checked by the latest versions of the license controller.
-6046	**Could not get file reference number information.** *Description.* The license could not get information about the license file. *Solution:* Reinstall the license. Contact your distributor.
-6047	**The device number of the license file has changed.** *Description:* The license file has been touched. The license is no longer valid. *Solution:* Reinstall the license. Contact your distributor.
-6048	**The file reference number of the license file has changed.** *Description:* The license file has been touched. The license is no longer valid. *Solution:* Reinstall the license. Contact your distributor.
-6049	**This product is licensed for runtime only. No compilation is allowed.** *Description:* You have a runtime license installed with this package. You cannot compile BDL source code modules with this license. *Solution:* If you want to compile 4GL source code, you must purchase and install a development license. Contact your distributor.
-6050	**Temporary license *license-number* expired. Please contact your vendor.** *Description:* A license with a time limit has been installed and the license has expired. *Solution:* Install a new license to activate the product. Contact your distributor.
-6051	**Temporary license *license-number* expired. Please contact your vendor.** *Description:* A license with a time limit has been installed and the license has expired. *Solution:* Install a new license to activate the product. Contact your distributor.
-6052	**Temporary license *license-number* expired. Please contact your vendor.** *Description:* A license with a time limit has been installed and the license has expired. *Solution:* Install a new license to activate the product. Contact your

	distributor.
-6053	**Installation path has changed. It must hold the original installation path.** *Description:* The value of FGLDIR or the location of FGLDIR has been changed. *Solution:* Ask the person who installed the product for the location of the original installation directory and then set the FGLDIR environment variable.
-6054	**Cannot read a license file. Check installation path and your environment. Verify if a license is installed.** *Description:* The file that contains the license is not readable by the current user. *Solution:* • License controller: Check that the FGLDIR environment variable is correctly set and that the file $FGLDIR/etc/f4gl.sn is readable by the current user. • License manager: Check that the file $FLMDIR/etc/license/lic?????.dat is readable by the current user.
-6055	**Cannot update a license file. Check installation path and your environment. Verify if a license is installed.** *Description:* The file that contains the license cannot be overwritten by the current user. *Solution:* • License controller: Check that the FGLDIR environment variable is correctly set and that the file $FGLDIR/etc/f4gl.sn is writable by the current user. • License manager: Check that the file $FLMDIR/etc/license/lic?????.dat is writable by the current user.
-6056	**Cannot write into a license file. Please check your rights.** *Description:* The file that contains the license cannot be overwritten by the current user. *Solution:* • License controller: Check that the FGLDIR environment variable is correctly set and that the file $FGLDIR/etc/f4gl.sn is writable by the current user. • License manager: Check that the file $FLMDIR/etc/license/lic?????.dat is writable by the current user.

-6057	**Cannot read a license file. Check installation path and your environment. Verify if a license is installed.** *Description:* The file that contains the license cannot be read by the current user. *Solution:* Check that the current user can read the file $FGLDIR/etc/f4gl.sn. Also check that the FGLDIR environment variable is set correctly.
-6058	**Incorrect license file format. Verify if a license is installed.** *Description:* The file that contains the license has been corrupted. *Solution:* Reinstall the license. If you have a backup of the current installation of Genero Business Development Language, restore the files located in the $FGLDIR/etc directory.
-6059	**Incorrect license file format. Verify if a license is installed.** *Description:* The file that contains the license has been corrupted. *Solution:* Reinstall the license. If you have a backup of the current installation of Genero Business Development Language, restore the files located in the $FGLDIR/etc directory.
-6061	**License '*license-number*' not installed.** *Description:* The license shown is not installed. *Solution:* Reinstall the license.
-6062	**No installed license has been found for '*license-number*'.** *Description.* The add-user license can not be installed. No main license found to add users. *Solution:* Contact your distributor.
-6063	**License '*license-number*' is already installed.** *Description:* The license shown is already installed. *Solution:* No particular action to be taken.
-6064	**The resource 'flm.license.number' is required to use the license manager.**
-6065	**The resource 'flm.license.key' is required to use the license manager.**
-6066	**License '*license-number*' cannot be installed over '*license-number*'.** *Description:* The add-user license does not match the main license. The add-user license can not be installed. *Solution:* Contact your distributor.
-6067	**You need a installed license if you want to add users.** *Description:* The add-user license must be installed after the main license. *Solution:* Install the main license before the add-user license. If this does not solve your problem, contact your distributor.
-6068	**No license installed.** *Description:* There is no license installed for Genero Business Development Language.

	Solution: Install a license. If a license is already installed, check that the $FGLDIR environment variable is set correctly.
-6069	**Cannot uninstall the license.** *Description:* There was a problem during the uninstall of the Genero Business Development Language license. *Solution:* Check whether the FGLDIR environment variable is correctly set in your environment and the current user has permission to delete files in the $FGLDIR/etc directory.
-6070	**The license server entry must be set in your resource file in order to reach the license server.** *Description:* You are using the remote license process and you have set the value of fgllic.server, in $FGLDIR/etc/fglprofile, to localhost or to the 127.0.0.1 address. *Solution:* You must use the real IP address of the computer even if it is the local computer.
-6071	**Cannot use directory '*directory-name*'. Check installation path and verify if access rights are 'drwxrwxrwx'.** *Description:* The compiler needs to operate in the specified directory. *Solution:* Change the permission of this directory.
-6072	**Cannot create file in '*file-name*'. Check installation path and verify if access rights are 'drwxrwxrwx'.** *Description:* The compiler needs to operate in the specified directory. *Solution:* Change the permission of this directory to 777 mode.
-6073	**Cannot change mode of a file in '*file-name*'. Verify if access rights are 'drwxrwxrwx'.** *Description:* The compiler needs to operate in the specified directory. *Solution:* Change the permission of this directory to 777 mode.
-6074	**'*file-name*' does not have 'rwxrwxrwx' rights or isn't a directory. Check access rights with 'ls -ld <installation-path>/lock' or execute 'rm -r <installation-path>/lock' if no users are connected.** *Description:* The compiler needs to operate in the specified directory. *Solution:* Change the permission of this directory. The $FGLDIR/lock directory contains only data needed at runtime by BDL applications. When the application is finished, you can remove this directory. If you delete this directory while BDL applications are running, the applications will be stopped immediately.
-6075	**Cannot read from directory '*directory-name*'. Check installation path and verify if access rights are 'drwxrwxrwx'.** *Description:* The compiler needs to operate in the specified directory. *Solution:* Change the permission of this directory.
-6076	**Bad lock tree. Please check your environment.** *Description:* There is a problem accessing the $FGLDIR/lock directory. *Solution:* Check if the current user has sufficient permission to read

	and write to the $FGLDIR/lock directory. Check also if the FGLDIR environment variable is correctly set.
-6077	**Bad lock tree. Please check your environment.** *Description:* There is a problem accessing the $FGLDIR/lock directory. *Solution:* Check if the current user has sufficient permission to read and write to the $FGLDIR/lock directory. Check also if the FGLDIR environment variable is correctly set.
-6079	**Cannot get machine name or network IP address. Each graphical client must have an IP address when using a license server. FGLSERVER must hold the IP address or the host name of the client.** *Description:* You are using the remote license process and you have set the value of fgllic.server, in $FGLDIR/etc/fglprofile, to localhost or to the 127.0.0.1 address. *Solution:* You must use the real IP address of the computer even if it is the local computer. This is also true for the value used with the FGLSERVER environment variable.
-6080	**Cannot get IP address from '*host-name*' host. Check the 'flm.server' resource.** *Description:* The system cannot find the IP address of the specified host. *Solution:* This is a configuration issue regarding your system. The command ping should not reply as well. Correct your system configuration and then try to execute your program.
-6081	**Cannot reach host '*host-name*' with ping. Check license server entry in your resource file. Check your network configuration or increase 'flm.ping' value.** *Description:* The license server cannot ping the client computer, or it does not get the response in the time limit specified by the fgllic.ping entry in the $FGLDIR/etc/fglprofile file. *Solution:* Try to manually ping the specified computer. If this works, try to increase the value of the fgllic.ping entry in fglprofile. If the ping does not respond, fix the system configuration problem and then try the program again.
-6082	**SYSERROR(%d)%s: Cannot set option TCP_NODELAY on socket. Check the system error message and retry.** *Description:* There is a problem with the socket of the Windows computer. *Solution:* Check that the system is correctly configured and retry the program.
-6085	**SYSERROR(%d)%s: Cannot connect to the license server on host '*host-name*'. Check following things:** **- license server entry.** **- the license server machine.** **- the license server TCP port.**

	Description: The application cannot check the license validity. To do so, it tries to communicate with the Genero Business Development Language license service running on the computer where the product is installed. *Solution:* Check that the Genero Business Development Language License Server is running on the computer where the product is installed.
-6086	**SYSERROR(%d)%s: Cannot send data to the license server. Check the system error message and retry.** *Description:* There is a problem with the socket of the Windows computer. *Solution:* Check that the system is correctly configured and retry the program.
-6087	**SYSERROR(%d)%s: Cannot receive data from license server. Check the system error message and retry.** *Description:* There is a problem with the socket of the Windows computer. *Solution:* Check that the system is correctly configured and retry the program.
-6088	**You are not allowed to be connected for the following reason: *description*** *Description:* The program cannot connect to the license server because of the specified reason. *Solution:* Try to fix the problem described and rerun your application.
-6089	**Each graphical client must have an IP address when using a license server. FGLSERVER must hold the IP address or the host name of the client (localhost or 127.0.0.1 are not allowed).**
-6090	**SYSERROR(%d)%s: Cannot create a socket to start the license server. Check the system error message and retry.** *Description:* There is a problem with the socket of the Windows computer. *Solution:* Check that the system is correctly configured and rerun the program.
-6091	**SYSERROR(%d)%s: Cannot bind socket for the license server. Check the system error message and retry.** *Description:* There is a problem with the socket of the Windows computer. *Solution:* Check that the system is correctly configured and rerun the program.
-6092	**SYSERROR(%d)%s: Cannot listen socket for the license server.** *Description:* There is a problem with the socket of the Windows computer. *Solution:* Check that the system is correctly configured and rerun the program.
-6093	**SYSERROR(%d)%s: Cannot create a socket to search an active**

	client. *Description:* There is a problem with the socket of the Windows computer. *Solution:* Check that the system is correctly configured and rerun the program.
-6094	**SYSERROR(%d)%s: This is a WSAStartup error. Check the system error message and retry.** *Description:* There is a problem with the socket of the Windows computer. *Solution:* Check that the system is correctly configured and rerun the program.
-6095	**License problem: *reason*** *Description:* License type incompatible. You are installing an earlier version, which was not designated for use with the current license server. *Solution:* Reinstall and then contact your vendor.
-6096	**Connection refused by the license server.** *Description:* There is problem connecting the client computer to the Windows license server. *Solution:* There is a configuration problem with the license server computer. Check the configuration of the computers and of the products.
-6100	**Bad format of line sent by the license requester.**
-6101	**License number '*license-number*' does not correspond to license key '*license-key*'.**
-6102	**Verify if resource 'flm.license.number' and 'flm.license.key' correspond to a valid license.**
-6103	**License '*license-number*' is no longer available from the license server.**
-6107	**User limit exceeded. Please retry later.** *Description:* The maximum number of clients that can be run has been reached (due to the license installed). *Solution:* Retry later (when the number of current users has decreased) or install a new license that allows more users.
-6108	**Environment is incorrect.** *Description:* There is no local license, or the environment is not set correctly. *Solution:* Check your environment and your FGLDIR environment variable.
-6109	**Cannot add session #%s.** *Description:* You do not have the permissions to create the new session (the directory representing the new client). *Solution:* Check the permissions of the dedicated directories.

-6110	**Cannot add program '*program-name*' (pid=%d).** *Description:* You do not have the permissions to create the new application (the file representing the new application) for the current user . *Solution:* Check the permissions of the dedicated directories.
-6112	**Compilation is not allowed: This product is licensed for runtime only.**
-6113	**Compilation is not allowed: Invalid license.**
-6114	**Cannot start program '*program-name*' or result of process number is 0.** *Description:* When fglWrt -u is executed to find the number of users allowed on this installation, the command "ps" may be launched (only on UNIX). *Solution:* Check the permissions for ps.
-6116	**Wrong number of characters.**
-6117	**The entry must be 12 characters long.**
-6118	**Wrong checksum result for this entry.**
-6122	**You must specify entry 'flm.server' in the resource file.**
-6123	**SYSERROR(%d)%s: Cannot open socket. Check the system error message and retry.**
-6129	**License uninstalled.** *Description:* This is an information message.
-6140	**Version *version-number*** *Description:* This is an information message.
-6148	**Installation path is not known.** *Description:* You are handling licenses but the FGLDIR environment variable is not set. *Solution:* Set the FGLDIR environment variable and retry.
-6149	**Problem while installing license '*license-number*'.** *Description:* A problem occurred while licensing. *Solution:* Note the system-specific error number and contact your Technical Support.
-6150	**Temporary license not found for this version.** *Description:* While adding a definitive license key, the temporary license has not been found. *Solution:* Re-install the license.
-6151	**Wrong installation key.** *Description:* While adding a definitive license key, the installation key was not valid. *Solution:* Re-install the license.
-6152	**Problem during license installation.**

	Description: A problem occurred while installing the license. Could not write information to the disk (either own files or system files). *Solution:* Check the FGLDIR environment variable and the rights of the license files (must be able to change them).
-6153	**License installation failed.**
-6154	**License installation successful.** *Description:* This is an information message.
-6156	**Too many temporary licenses. You must reinstall a license.** *Description:* You installed a temporary license too many times. *Solution:* Contact technical support to get a valid license.
-6158	**Cannot store temporary information.** *Description:* A problem occurred while installing the license. Could not write information to the disk (either own files or system files). *Solution:* Check the FGLDIR environment variable and the rights of the license files (you must be able to change them).
-6159	**This kind of license is not permitted.**
-6160	**You do not have the permissions to be connected.**
-6161	**You do not have the permissions to compile.**
-6162	**Cannot reach the license server. Please check if 'flm.server' is correctly initialized. ('flmprg -a info up' command should answer 'ok'). The license server is running but no autocheck will be done.**
-6168	**Cannot get information from directory '*directory-name*'.**
-6170	**Old request format to license server detected. You must install a license program version 2.99 or higher.**
-6171	**A license has been installed temporarily. Only the installation key is required. You must run 'fglWrt -k <installation-key>' to install it.**
-6172	**Bad parameter: '*parameter*' hasn't the right format.**
-6173	**Invalid license number or invalid license key.**
-6174	**This option is only available for a local license. And resource 'flm.server' was found in your configuration.**
-6175	**License number '*license-number*' is invalid.**
-6176	**In license server, following problem occurs with license number '*license-number*': *problem-description***
-6177	**Following problem occurs with license number '*license-number*': *problem-description***
-6178	**Your machine is not allowed to be connected on any of your authorized licenses.**

-6179	**License validity time is reached. The users control is reactivated.**
-6180	**CPU limit exceeded. The users control is reactivated.**
-6181	**Cannot get license extension information. Check your environment, the license (run 'fglWrt -a info') and the fglWrt version ('fglWrt -V' should give *version-number* or higher).**
-6182	**Your license has '*restriction-name*' restriction. You are not allowed to run another mode.**
-6183	**Local license controller (fglWrt) may not be compatible with this runner. Check its version ('fglWrt -V' should give *version-number* or higher).**
-6184	**You are not authorized to run this version of runner.**
-6185	**Protection file is not compatible with this version of the runner. You must reinstall your license.**
-6186	**Demo version initialization.** *Description.* This is an information message.
-6196	**You are not authorized to delete sessions from the license server '*server-name*'.**
-6197	**'*extension-name*' extension is not allowed with this license type.**
-6198	**Product identifier does not correspond to the license number.**
-6200	**Module '*module-name*': The function *function-signature-1* will be called as *function-signature-2*.** *Description:* An incorrect number of parameters are used to call a BDL function. *Solution:* Check your source code and recompile your application.
-6201	**Module '*module-name*': Bad version: Recompile your sources.** *Description:* You have compiled your program with an old version. The newly compiled version of your program is not supported. *Solution:* Compile all source files and form files again.
-6202	**filename '*file-name*': Bad magic: Code can't run with this p code machine.** *Description:* You have compiled your program with an old version. The new compiled version of your program is not supported. You might also have a file with the same name as the .42r. You used the fglrun 42r-Name without specifying the extension. *Solution:* To resolve this problem, call fglrun with the .42r extension or recompile your application.
-6203	**Module '*module-name-1*': The function '*function-name*' has already been defined in module '*module-name-2*'.** *Description.* The specified function is defined for the second time in the application. The second occurrence of the function is in the specified module.

	Solution: Eliminate one of the two function definitions from your source code.
-6204	**Module '*module-name*': Unknown opcode.** *Description:* An unknown instruction was found in the compiled BDL application. *Solution:* Check that the version of the Genero Business Development Language package executing the compiled application is the same as the one that compiled the application. It is also possible that the compiled module has been corrupted. If so, you will need to recompile your application.
-6205	**INTERNAL ERROR: Alignment.** *Description:* This error is internal, which should not normally occur. *Solution:* Contact your Technical Support.
-6206	**The dynamic loader can not open module '*module-name*'.** *Description:* The module is not in the current directory or in one of the directories specified by the environment variable **FGLLDPATH**. *Solution:* Set the environment variable **FGLLDPATH**.
-6207	**The dynamic loaded module '*module-name*' does not contain the function '*function-name*'.** *Description:* A BDL module has been changed and recompiled, but the different modules of the application have not been linked afterward. *Solution:* Link the new modules together before you execute your application.
-6208	**Module '*module-name*' already loaded.** *Description:* A module was loaded twice at runtime. This can occur because one module has been concatenated with another. *Solution:* Recompile and re-link your BDL modules.
-6210	**INTERNAL ERROR: exception 2 raised before invoking the exception handler for exception 1.** *Description:* A module was loaded twice at runtime. This can occur because one module has been concatenated with another. *Solution:* Check for function names, recompile and re-link your BDL modules.
-6211	**Link has failed.** *Description:* A problem occurred while linking the BDL program. *Solution.* Check for function names, recompile and re-link your BDL modules.
-6212	**Function *function-name*: local variables size is too large - Allocation failed.** *Description:* A local function variable is too large and runtime could not allocate memory. *Solution.* Review the variable data types in the function.
-6213	**Module *module-name*: Module's variable size is too large - Allocation failed.**

	Description: A module variable is too large and runtime could not allocate memory. *Solution.* Review the variable data types in the module.
-6214	**Global variable *variable-name* size is too large - Allocation failed.** *Description:* A global variable is too large and runtime could not allocate memory. *Solution.* Review the variable data types in the globals.
-6215	**Memory allocation failed. Ending program.** *Description:* Runtime could not allocate memory. *Solution.* Check for system resources and verify if the OS user is allowed to allocate as much memory as the program needs (check for **ulimits** on UNIX systems).
-6216	**The global '*name*' has been redefined with a different constant-value.** *Description:* A global constant has been defined twice with a different value. *Solution:* A global constant may have only one value. Review your code.
-6217	**The global '*name*' has been defined as a constant and a variable.** *Description:* The same symbol was used to define a constant and a variable. *Solution:* Use a different name for the constant and the variable. Review your code.
-6218	**No runtime. You must call fgl_start() before calling fgl_call().** *Description:* This error occurs when a C extension has re-defined the main() routine, but then does not call fgl_start() to initialize the BDL runtime environment. *Solution:* Check the C extension and call fgl_start() before any other operation.
-6219	**WHENEVER ERROR CALL: The error-handler recursively calls itself.** *Description:* The exception handler calls a function which in turn calls itself recursively. *Solution:* Review the function called by the exception handler.
-6220	**Could not load C extension library '*library-name*'.\nReason: *reason*** *Description:* Runtime system could not find the shared library for the reason given. *Solution:* Check if the C extension library exists in one of the directories defined by FGLLDPATH.
-6221	**C extension initialization failed with status %d.** *Description:* C extension failed to initialize. *Solution:* Check the C extension source or documentation.
-6300	**Can not connect to GUI.**

	Description: You have run a GUI application but the environment variable FGLSERVER is not set correctly, or the Genero client (graphical front-end) is not running. *Solution:* The FGLSERVER environment variable should be set to the hostname and port of the graphical front end used by the runtime system to display the application windows. Check that the network connection is still available, make sure no firewall denies access to the workstation, and see whether the front-end is still running.
-6301	**Can not write to GUI.** *Description:* You are running a GUI application but for an unknown reason the front-end no longer responds and the runtime system could not write to the GUI socket. *Solution:* Check that the network connection is still available, make sure no firewall denies access to the workstation, and see whether the front-end is still running.
-6302	**Can not read from GUI.** *Description:* You are running a GUI application but for an unknown reason the front-end no longer responds and the runtime system could not read from the GUI socket. *Solution:* Check that the network connection is still available, make sure no firewall denies access to the workstation, and see whether the front-end is still running.
-6303	**Invalid user interface protocol.** *Description:* You are trying to execute a program with a runtime system that uses a different AUI protocol version as the front-end. *Solution:* Install either a new front-end or a new runtime environment that matches (2.0x with 2.0x, 1.3x with 1.3x).
-6304	**Invalid abstract user interface definition.** *Description:* You are trying to execute a program with a runtime system that uses a different AUI protocol version as the front-end. *Solution:* Install either a new front-end or a new runtime environment that matches (2.0x with 2.0x, 1.3x with 1.3x).
-6305	**Can not open char table file. Check your fglprofile.** *Description:* This error occurs if the conversion file defined by the gui.chartable entry, in the $FGLDIR/etc/fglprofile file, is not readable by the current user. *Solution:* Check if the gui.chartable entry is correctly set and if the specified file is readable by the current user.
-6306	**Can not open server file. Check installation.** *Description:* A file on the server side cannot be sent to the graphical interface. *Solution:* Check the permissions of the file located in the $FGLDIR/etc directory. These files must have at least read permission for the current user.
-6307	**GUI server autostart: can not identify workstation.** *Description:* GUI Server autostart configuration is wrong. Either

	DISPLAY, FGLSERVER or fglprofile settings are invalid. *Solution:* Set the required environment variables and check for fglprofile autostart entries.
-6308	**GUI server autostart: unknown workstation: check gui.server.autostart entries.** *Description:* The computer described by the X11 DISPLAY environment variable is neither the local host, nor is it listed in the fglprofile entries. *Solution:* Check if the X11 DISPLAY name is correctly set, or review the fglprofile entries.
-6309	**Not connected. Cannot write to GUI.** *Description:* For unknown reasons there was an attempt to write on the GUI socket before the connection was initiated. *Solution:* Check the program for invalid GUI operations.
-6310	**Not connected. Cannot read from GUI.** *Description:* For unknown reasons there was an attempt to read on the GUI socket before the connection was initiated. *Solution:* Check the program for invalid GUI operations.
-6311	**No current window.** *Description:* The program tries to issue a MENU instruction with no current window open. *Solution:* Review the program logic and make sure a window is open before MENU.
-6312	**The type of the user interface (FGLGUI) is invalid.** *Description:* While initiating the user interface, the runtime system did not recognize the GUI type and stopped. *Solution:* Make sure the FGLGUI environment variable has a correct value.
-6313	**The UserInterface has been destroyed.** *Description:* The error occurs when the front-end sends a DestroyEvent event, indicating some inconsistency with the starting program. This can happen, for example, when multiple StartMenus are used, or when you try to run an MDI child without a parent container, or when two MDI containers are started with the same name, etc. *Solution:* Check for inconsistency and fix it.
-6314	**Wrong connection string. Check client version.** *Description:* While starting the program, the runtime received a wrong or incorrectly constructed answer from the front-end. *Solution:* Make sure you are using a front-end that is compatible with the runtime system.
-6315	**The form is too complex for the console-ui.** *Description:* The program tries to display a form with a complex layout that can't be displayed in text mode. *Solution:* Review the form file and use a simple grid with a SCREEN

	section instead of LAYOUT.
-6316	**Error *error-number* returned from client:\n *description*** *Description:* Front end returned the specified error during GUI connection initialization. *Solution:* Check the front-end documentation for more details.
-6317	**Invalid or unsupported client protocol feature.** *Description:* The GUI protocol feature you are trying to use is not supported by the front-end. For example, you are trying to use protocol compression but the runtime is not able to compress data. *Solution:* Check the runtime system version for supported protocol features.
-6318	**The function '*function-name*' cannot be called with this version of fglrun.** *Description:* The function or class method shown in the error message cannot be used in the current context. *Solution:* Use a different function or method.
-6320	**Can't open file '*file-name*'.** *Description:* The runtime system tried to open a resource file in FGLDIR but access is denied or file no longer exists. *Solution:* Check for file permissions and existence in FGLDIR.
-6321	**No such interface capability: '*feature*'.** *Description:* The runtime system tried to use a front-end protocol capability, but is not able to use it. *Solution:* Check if the front-end is compatible with the runtime system.
-6322	**%s wrong version. Expecting %s.** *Description:* Some resource files of FGLDIR have been identified as too old for the current runtime system. *Solution:* Re-install the runtime system environment.
-6323	**Can't load factory profile '*file-name*'.** *Description:* The default fglprofile file located in FGLDIR/etc is missing or is unreadable. *Solution:* Check the permission of the file. If the file is missing, reinstall the software.
-6324	**Can't load customer profile '*file-name*'.** *Description:* The configuration file defined by the FGLPROFILE environment variable is missing or unreadable. *Solution:* Check if the FGLPROFILE environment variable is correctly set and if the file is readable by the current user.
-6325	**Can't load application resources '*file-name*'.** *Description:* The directory specified by the fglrun.default entry in FGLDIR/etc/fglprofile is missing or not readable for the current user. *Solution:* Check if the entry fglrun.default is correctly set in FGLDIR/etc/fglprofile and if the directory specified is readable by the current user.

-6326	**Can't open char map file '*file-name*'. Check your fglprofile.** *Description:* The specified char map file cannot be found or read. *Solution.* Verify that the char map file is located in FGLDIR/etc, and that the correct value is set in fglprofile (GUI.CHARTABLE entry).
-6327	**Internal error in the run time library file *library-name*.** *Description:* Something unpredictable has occurred, generating an error. *Solution:* Contact your Technical Support.
-6328	**Bad format of resource '*name*' value '*value*': you must use the syntax :** **%s='VARNAME=value'.** *Description:* The FGLPROFILE file contains an invalid environment variable definition format. *Solution:* Check the content of the profile file.
-6330	**Syntax error in profile '*filename*', line number *lineno*, near '*token*'.** *Description:* The FGLPROFILE file shown in the error message contains a syntax error. *Solution:* Check the content of the profile file.
-6331	**Front end module could not be loaded.** *Description:* A front end call failed because the module does not exist. *Solution:* The front end is probably not supporting this module.
-6332	**Front end function could not be found.** *Description:* A front end call failed because the function does not exist. *Solution:* The front end is probably not supporting this function.
-6333	**Front end function call failed.** *Description:* A front end call failed for an unknown reason. *Solution:* Call the support and report the problem.
-6334	**Front end function call stack problem.** *Description:* A front end call failed because the number of parameter or returning values does not match. *Solution:* Make sure the number of parameters and return values are correct.
-6340	**Can't open file.** *Description:* The channel object failed to open the file specified. *Solution:* Make sure the filename is correct and user has permissions to read/write to the file.
-6341	**Unsupported mode for 'open file'.** *Description:* You try to open a channel with an unsupported mode. *Solution:* See channel documentation for supported modes.
-6342	**Can't open pipe.** *Description:* The channel object failed to open a pipe to execute the

	command. *Solution:* Make sure the command you try to execute is valid.
-6343	**Unsupported mode for 'open pipe'.** *Description:* You try to open a channel with an unsupported mode. *Solution:* See channel documentation for supported modes.
-6344	**Can't write to unopened file, pipe or socket.** *Description:* You try to write to a channel object which is not open. *Solution:* First open the channel, then write.
-6345	**Channel write error.** *Description:* An unexpected error occurred while writing to the channel. *Solution:* See system error message for more details.
-6346	**Cannot read from unopened file, pipe or socket.** *Description:* You try to read from a channel object which is not open. *Solution:* First open the channel, then read.
-6360	**This runner can't execute any SQL.** *Description:* The runtime system is not ready for database connections. *Solution:* Check the configuration of FGL.
-6361	**Dynamic SQL: type unknown: *typename*.** *Description:* The database driver does not support this SQL data type. *Solution:* You cannot use this SQL data type, review the code.
-6364	**Cannot connect to sql back end.** *Description:* The runtime system could not initialize the database driver to establish a database connection. *Solution:* Make sure the database driver exists.
-6365	**Database driver not connected yet.** *Description:* There is an attempt to execute an SQL statement, but no database connect is established. *Solution:* First connect, then execute SQL statements.
-6366	**Could not load database driver *driver-name*.** *Description:* The runtime system failed to load the specified database driver. The database driver DLL or a dependent DLL could not be found. *Solution:* There is probably an environment problem, check for example the UNIX LD_LIBRARY_PATH environment variable.
-6367	**Incompatible database driver interface.** *Description:* The database driver interface does not match the interface expected by the runtime system. This can occur if you copy an old database driver into a younger FGLDIR installation. *Solution:* Call the support to get a valid database driver.
-6368	**SQL driver initialization function failed.**

	Description: The runtime system failed to initialize the database driver, program must stop because no database connection can be established. *Solution:* There is probably an environment problem (for example, INFORMIXDIR or ORACLE_HOME is not set). Check your environment and try to connect with a database vendor tool (dbaccess, sqlplus) to identify the problem.
-6369	**Invalid database connection mode.** *Description:* You try to mix DATABASE and CONNECT statements, but this is not allowed. *Solution:* Use either DATABASE or CONNECT.
-6370	**Unsupported SQL feature.** *Description:* This SQL command or statement is not supported with the current database driver. *Solution:* Review the code and use a standard SQL feature instead.
-6371	**SQL statement error number %d (%d).** *Description:* An SQL error has occurred having the specified error number. *Solution:* You can query SQLERRMESSAGE or the SQLCA record to get a description of the error.
-6372	**General SQL error, check SQLCA.SQLERRD[2].** *Description:* A general SQL error has occurred. *Solution:* You can query SQLERRMESSAGE or the SQLCA record to get a description of the error. The native SQL error code is in SQLCA.SQLERRD[2].
-6373	**Invalid database connection string.** *Description:* The database connection string that you have used is not valid. *Solution:* Verify the format of the connection string.
-6374	**Wrong database driver context.** *Description:* You try to EXECUTE, OPEN, FETCH, PUT, FLUSH, CLOSE or FREE a cursor that was declared or prepared in a different connect and driver. *Solution:* Issue a SET CONNECTION before the statement to select the same connection and driver as when the cursor was created.
-6375	**LOAD cannot get describe information for table columns.** *Description:* The LOAD instructions needs column description to allocate the automatic fetch buffers, but the database driver is not able to describe the table columns used in the INSERT statement. *Solution:* If the underlying database client API does not provide result set column description, the LOAD statement cannot be supported.
-6601	**Can not open Database dictionary '*name*'. Run database schema extraction tool.** *Description:*b>. The schema file does not exist or cannot be found. *Solution:* If the schema file exists, verify that the filename is spelled

	correctly, and that the file is in the current directory or the FGLDBPATH environment variable is set to the correct path. If the file does not exist, run the database schema extraction tool to create a schema file.
-6602	**Can not open globals file '*name*'.** *Description:* The globals file does not exist or cannot be found. *Solution:* Verify that the globals file exists. Check the spelling of the filename, and verify that the path is set correctly.
-6603	**The file '*name*' cannot be created for writing.** *Description:* The compiler failed to create the file shown in the error message for writing. *Solution:* Check for user permissions to make sure that the **.42m** file can be created.
-6604	**The function '*function-name*' can only be used within an INPUT [ARRAY], DISPLAY ARRAY or CONSTRUCT statement.** *Description:* The language provides built-in functions that can only be used within specific interactive statements. *Solution:* Review your code and make the necessary corrections.Check that the function is within the interactive statement and that appropriate END statements (END INPUT/ARRAY/DISPLAY ARRAY/CONSTRUCT) have been used.
-6605	**The module '*name*' does not contain function '*function-name*'.** *Description.* The module shown in the error message does not hold the function name as expected. *Solution:* The specified function needs to be defined in this module.
-6606	**No member function '*name*' for class '*class-name*' defined.** *Description.* The function name is mispelled or is not a method of the class for which it is called. *Solution:* Review your code and the documentation for the method you are attempting to use. If the function is an object method, make sure the referenced object in your code is of the correct class.
-6608	**Resource error:%s:parameter expected** *Description.* This is a generic error message for resource file problems.
-6609	**A grammatical error has been found at '%s' expecting: %s.** *Description:* A general syntax error message that indicates the location of the problem code and what code was expected. *Solution:* Review your code, particularly for missing END statements such as END FUNCTION or END INPUT, etc., and make the necessary corrections.
-6610	**The function '*name*' has already been called with a different number of parameters.** *Description:* Earlier in the program, there is a call to this same function or event with a different number of parameters in the parameter list.

	Solution: Check the correct number of parameters for the specified function. Then examine all calls to it, and make sure that they are written correctly.
-6611	**Function '*name*': unexpected number of returned values.** *Description:* The function shown returned a different number of values as expected. *Solution:* Check the body of the function for RETURN instructions.
-6612	**Redeclaration of function '*name*'.** *Description:* The function shown was defined multiple times. *Solution:* Change the name of conflicting functions.
-6613	**The library function '*name*' is not declared.** *Description:* The function shown was not declared. *Solution:* Change the name of the function.
-6614	**The function '*name*' may return a different number of values.** *Description:* The function shown contains multiple RETURN instructions which may return different number of values. *Solution:* Review the RETURN instructions to return the same number of values.
-6615	**The symbol '*name*' is unused.** *Description:* This is a warning indicating that the shown symbol is defined but never used. *Solution:* Useless definition can be removed.
-6616	**The symbol '*name*' does not represent a defined CONSTANT.** *Description:* The shown symbol is used as a CONSTANT, but it is not a constant. *Solution:* Review your code and check for this name.
-6617	**The symbol '*name*' is a VARIABLE.** *Description:* The symbol shown is a VARIABLE which cannot be used in the current context. *Solution:* Review your code and check for this name.
-6618	**The symbol '*name*' is a CONSTANT.** *Description:* The symbol shown is a CONSTANT which cannot be used in the current context. *Solution:* Review your code and check for this name.
-6619	**The symbol '*name*' is not an INTEGER CONSTANT.** *Description:* The symbol shown is used as if it was an INTEGER constant, but it is not. *Solution:* Review your code and check for this name.
-6620	**The symbol '*name*' is not a REPORT.** *Description:* The symbol shown is used as a REPORT, but it is not defined as a REPORT. *Solution:* Review your code and check for this name.
-6621	**The symbol '*name*' is not a FUNCTION.**

	Description: The symbol shown is used as a FUNCTION, but it is not defined as FUNCTION. *Solution:* Review your code and check for this name.
-6622	**The symbol '*name*' does not represent a valid variable type.** *Description:* The symbol shown does not . *Solution:* Review your code and check for this name.
-6623	**The method '*method-name*' can't be called without an object.** *Description:* The specified method is an object method of its class. *Solution:* Review your code. Ensure that the required object of the class has been instantiated and still exists, and that the method is called specifying the object variable as the prefix, with the period character as a separator.
-6624	**The method '*method-name*' can't be called with an object.** *Description:* The specified method is a class method and cannot be called using an object reference. No object has to be created. *Solution:* Review your code. Ensure that the method is called using the class name as the prefix, with the period character as a separator.
-6625	**The statement is not Informix compatible.** *Description:* The SQL statement is not Informix compatible. *Solution:* Change the SQL statement by using Informix SQL syntax.
-6627	**The symbol '*name*' is not a VARIABLE.** *Description:* The symbol shown is use as a variable, but is not defined as a variable. *Solution:* Review your code and check for this name.
-6628	**The GLOBALS file does not contain a GLOBALS section.** *Description:* The filename specified in a GLOBALS statement references a file that does not contain a GLOBALS section. *Solution:* Review your code to make sure that the file specified by the filename is a valid GLOBALS file, containing the required GLOBALS section.
-6629	**The type 'type-name' is too complex to be used within a C-extension.** *Description:* The type of the global variable is too complex to be used in a C extension. This error can occur when the -G option of fglcomp, to generate the C sources to share global variables with C extensions, when a global variable is defined with complex data types without a C equivalent. *Solution:* Review the definition of the global variables and use simple types instead, corresponding to a C data type. The BYTE, TEXT and STRING types are complex types.
-6630	**Memory overflow occurred during p-code generation. Simplify the module.** *Description:* A memory overflow occurred during compilation to p-

	code because the 4gl source module is too large. *Solution:* This problem can occur with very large source files. You must split the module into multiple sources.
-6802	**Can not open Database dictionary '*name*'. Run schema extraction tool.** *Description:* The schema file does not exist or cannot be found. *Solution:* If the schema file exists, verify that the filename is spelled correctly, and that the file is in the current directory or the FGLDBPATH environment variable is set to the correct path. If the file does not exist, run the database schema extraction tool to create a schema file.
-6803	**A grammatical error has been found at '*line*', expecting *token*.** *Description:* This is a generic message for errors.
-6804	**'*name*' form compilation was successful.** *Description:* This is an information message indicating that the form was compiled without problem.
-6805	**Open Form '*name*', Bad Version:%s, expecting:%s.** *Description:* You have compiled your form with a version of the form compiler that is not compatible with that used for compiling the other source code. *Solution:* Compile your form file and related source code files using the same or compatible versions of the compilers.
-6807	**The label '*name*' could not be used as column-title.** *Description:* The form file defines an invalid TABLE column title. *Solution:* Check for column titles which are not corresponding to column positions.
-6808	**The widget '*name*' can not be defined as array.** *Description:* The form file defines an item which is used as a matrix column. *Solution:* Review your form definition.
-6809	**The layout tag '*name*' is invalid, expecting: *token*.** *Description:* The form compiler detected an invalid layout tag specification. *Solution:* Review your form definition.
-6810	**The attribute '*attribute*' is invalid for item type '*name*'.** *Description:* The form compiler detected an invalid attribute definition for this item type. *Solution:* Review your form definition and check for invalid attributes.
-6811	**Syntax error near '%s', expecting %s.** *Description:* A general syntax error message that indicates the location of the problem code and what code was expected. *Solution:* Review your code and make the necessary corrections.
-6812	**Unterminated char constant.** *Description:* The form compiler detected an unterminated character

	constant. *Solution:* Review your form definition and check for missing quotes or double-quotes.
-6813	**The element '*name*' conflicts with group-box '*name*'.** *Description:* You have used the same name for an element and for a group-box. *Solution:* Review your form definition and ensure that the names used are unique.
-6814	**The screen records '*name*' must reference one table.** *Description:* The shown screen record references multiple tables in your form file. *Solution:* Review your form definition and use one unique table for a given screen record.
-6815	**Invalid indentation in between braces.** *Description:* The LAYOUT section of your form defines an invalid indentation. *Solution:* Review your form definition and check for corresponding indentations.
-6817	**TABLE container defined without a SCREEN RECORD in the INSTRUCTION section.** *Description.* The minimum value of the defined attribute must be lower than the maximum value. *Solution:* Review your code and make the necessary corrections.
-6818	**Min value must be lower that Max value.** *Description.* The minimum value of the defined attribute must be lower than the maximum value. *Solution:* Review your code and make the necessary corrections.
-6819	**Number of elements in the SCREEN RECORD must match the number of columns in TABLE container.** *Description:* The number of elements defined in the screen record is not equal to the number of columns used for the TABLE container. *Solution:* Review your form definition.
-6820	**ScrollGrid and/or Group layout tags cannot be nested.** *Description:* The form definition has nested ScrollGrid and/or Group layout tags. These tags cannot be nested. *Solution:* Review your form definition and make the necessary corrections.
-6821	**HBOX tags cannot be used for ARRAYS.** *Description:* The form definition is using an HBOX tag for an array, which is not permitted. *Solution:* Review your form definition and make the necessary corrections.
-6822	**Escaped graphical characters are not accepted in GRID sections.** *Description:* You try to use Text User Interface graphics in the new

	GRID container. *Solution:* This is not allowed, use GROUPs instead.
-6823	**Close tag does not have a matching tag above.** *Description:* The form definition has a close tag without a prior matching open tag. Open tags and close tags must match. *Solution:* Review your form definition file and make the necessary corrections.
-6824	**The table '*tablename*' is empty.** *Description:* The form layout defines a table layout tag identified by *tablename*, but nothing was found directly under this table which could be a column or a column title. *Solution:* Append columns to the table layout region.
-6825	**The tag '*tagname*' overlaps with table '*tablename*'.** *Description:* In the form layout, *tagname* overlaps the layout region of *tablename* and makes it invalid. *Solution:* Move or remove *tagname*, or redefine the layout region of *tablename*.
-6826	**Checked value must be different from unchecked value for field 'fieldname'.** *Description:* The VALUECHECKED and VALUEUNCHECKED attributes have the same value. This makes no sense because these attributes define the values corresponding to the checked and unchecked states of a checkbox. *Solution:* Use different values for these attributes.
-6826	**Duplicated item key found for field 'fieldname'.** *Description:* The ITEMS attribute of field *fieldname* defines item keys with the same value. *Solution:* Check ITEMS attribute and use unique key values. Note that '' and NULL are equivalent.
-8000	**Dom: Node not found.** *Description:* The node could not be found in the current document. *Solution:* Review your code.
-8001	**Dom: Invalid Document.** *Description:* The document passed to the DOM API is not a valid document. *Solution:* Review your code.
-8002	**Dom: Invalid usage of NULL as parameter.** *Description:* NULL cannot be used at this place. *Solution:* Review your code.
-8003	**Dom: A node is inserted somewhere it doesn't belong.** *Description:* You try to insert a node under a parent node which does not allow this type of nodes. *Solution:* Check for the possible nodes and review your code.
-8004	**Sax: Invalid hierarchy.**

	Description: The SAX handler encountered an invalid hierarchy. *Solution:* Make sure parent/child relations are respected.
-8005	**Deprecated feature: %s** *Description:* The feature you are using will be removed in a next version. *Solution:* A replacement for the feature is normally available.
-8006	**The string resource file '*name*' cannot be found.** *Description:* The string file shown could not be found. *Solution:* Check if file exists and if path is valid.
-8007	**The string resource file '*name*' cannot be read.** *Description:* The string file shown could not be read. *Solution:* Check if file exists and if user has read permissions.
-8008	**The string key '*key*' has no defined value.** *Description:* The runtime system could not find a string resource corresponding to the shown key. *Solution:* Check if the key is defined in one of the resource files.
-8009	**String resource syntax error near 'token', expecting token.** *Description:* The string file compiler detected a syntax error. *Solution:* Check for invalid syntax in the **.str** file.
-8010	**The included string file '*name*' cannot be found (*filename*:*line*) IGNORE LINE.** *Description:* The string file compiler could not find the file to be included. *Solution:* Check file name and path.
-8011	**The included string file '*name*' was already included (*filename*:*line*) IGNORE LINE.** *Description:* The string file compiler detected that the included file was already included. *Solution:* Remove the inclusion.
-8012	**Duplicate string key '*key*' (*filename*:*line*) IGNORE LINE.** *Description:* The string file compiler detected duplicated string keys. *Solution:* Review the **.str** file and remove duplicated keys.
-8013	**The string file '*name*' can not be opened for writing.** *Description:* The string file compiler could not write to the specified string file. *Solution:* Make sure the user has write permissions and file name is valid.
-8014	**The string file '*name*' can not be read.** *Description:* The runtime system could not read from the specified string file. *Solution:* Make sure the user has read permissions.
-8015	**Field (*name*) in ON CHANGE clause not found in form.** *Description:* The field used in the ON CHANGE clauses was not

	found in the form specification file. *Solution:* Make sure the field name of the ON CHANGE clause matches a valid form field.
-8016	**You cannot have multiple ON CHANGE clauses for the same field.** *Description:* It is not possible to specify multiple ON CHANGE clauses using the same field. *Solution:* Remove un-necessary ON CHANGE clauses.
-8017	**SFMT: Invalid % index used.** *Description:* The format string is not valid. *Solution:* Check for invalid % positions.
-8018	**SFMT: Format error.** *Description:* The format string is not valid. *Solution:* Check for invalid % positions.
-8019	**No more than one ON ROW CHANGE clause can appear in an INPUT ARRAY statement.** *Description:* Multiple ON ROW CHANGE clause were found in the same INPUT ARRAY. *Solution:* Remove un-necessary ON ROW CHANGE clauses.
-8020	**Multiple ON ACTION clauses with the same action name appear in the statement.** *Description:* It is not possible to specify multiple ON ACTION clauses using the same action name. *Solution:* Remove un-necessary ON ACTION clauses.
-8021	**Multiple ON KEY clauses with the same key name appear in the statement.** *Description:* It is not possible to specify multiple ON KEY clauses using the same key. *Solution:* Remove un-necessary ON KEY clauses.
-8022	**Dom: Cannot open xml-file.** *Description:* The file could not be loaded. *Solution:* Check file name and user permissions.
-8023	**Dom: The attribute '*name*' does not belong to node '*node*'.** *Description:* You try to set an attribute to a node which does not have such attribute. *Solution:* This is not allowed, review your code.
-8024	**Dom: Character data can not be created here.** *Description:* You try to create a character node under a node which does not allow such nodes. *Solution:* This is not allowed, review your code.
-8025	**Dom: Cannot set attributes of a character node.** *Description:* You try to set attributes in a character node. *Solution:* This is not allowed, review your code.

-8026	**Dom: The attribute '*name*' can not be removed: the node '*node*' belongs to the user-interface.** *Description:* You try to remove a mandatory attribute from an AUI node. *Solution:* You can only change the value of this attribute, try 'none' or an empty string.
-8027	**Sax: can not write.** *Description:* The SAX handlers could not write to the destination file. *Solution:* Make sure the file path is correct and the user has write permissions.
-8028	**Multiple ON IDLE clauses appear in the statement.** *Description:* Only one ON IDLE clause can be used inside a dialog block. *Solution:* Remove un-necessary ON IDLE clauses.
-8029	**Multiple inclusion of the source file '*name*'.** *Description:* The preprocessor detected that the specified file was included several times by the same source. *Solution:* Remove un-necessary file inclusions.
-8030	**The full path to the source file '*name*' is too long.** *Description:* The preprocessor does not support very long file names. *Solution:* Rename the file.
-8031	**The source file '*name*' cannot be read.** *Description:* The preprocessor could not read the file specified. *Solution:* Make sure the use has read permissions.
-8032	**The source file '*name*' cannot be found.** *Description:* The preprocessor could not find the file specified. *Solution:* Make sure the file exists.
-8033	**Extra token found after '*name*' directive.** *Description:* The preprocessor detected an unexpected token after the shown directive. *Solution:* Review your code and make the necessary corrections.
-8034	***feature*: This feature is not implemented.** *Description:* This preprocessor feature is not supported. *Solution:* Review your code and make the necessary corrections.
-8035	**The macro '*name*' has already been defined.** *Description:* The preprocessor found a duplicated macro definition. *Solution:* Review your code and make the necessary corrections.
-8036	**A &else directive found without corresponding &if,&ifdef or &ifndef directive.** *Description:* The preprocessor detected an unexpected **&else** directive. *Solution:* Review your code and make the necessary corrections.
-8037	**A &endif directive found without corresponding &if,&ifdef or**

	&ifndef directive. *Description:* The preprocessor detected an unexpected **&endif** directive. *Solution:* Review your code and make the necessary corrections.
-8038	**Invalid preprocessor directive &*name* found.** *Description:* The preprocessor directive shown in the error message does not exist. *Solution:* Review your code and check valid macros.
-8039	**Invalid number of parameters for macro *name*.** *Description:* The number of parameters of the preprocessor macro shown in the error message does not match de number of parameters in the definition of this macro. *Solution:* Review your code and check for the number of parameters.
-8040	**Lexical error : Unclosed string.** *Description:* The compiler detected an unclosed string and cannot continue. *Solution:* Review your code and make the necessary corrections.
-8041	**Unterminated condition &if or &else.** *Description:* The preprocessor found an un-terminated conditional directive. *Solution:* Review the definition of this directive.
-8042	**The operator '##' can only be used with identifiers and numbers. %s is not allowed.** *Description:* The preprocessor found an invalid usage of the ## string concatenation operator. *Solution:* Review the definition of this macro.
-8043	**Could not run FGLPP, command used : *command*** *Description:* The compiler could not run the preprocessor command shown in the error message. *Solution:* Make sure the preprocessor command exists.
-8044	**Lexical error : Unclosed comment.** *Description:* The compiler detected an unclosed comment and cannot continue. *Solution:* Review your code and make the necessary corrections.
-8045	**This type of statement can only be used within an INPUT, INPUT ARRAY, DISPLAY ARRAY, CONSTRUCT or MENU statement.** *Description:* This statement has not been used within a valid interactive statement, which must be terminated appropriately with END INPUT, END INPUT ARRAY, END DISPLAY ARRAY, END CONSTRUCT, or END MENU. *Solution:* Review your code and make the necessary corrections.
-8046	**This type of statement can only be used within an INPUT, INPUT ARRAY, DISPLAY ARRAY or CONSTRUCT statement.** *Description:* This statement has not been used within a valid interactive statement, which must be terminated appropriately with

	END INPUT, END INPUT ARRAY, END DISPLAY ARRAY, or END CONSTRUCT. *Solution:* Review your code and make the necessary corrections.
-8047	**Invalid use of 'dialog'. Must be used within an INPUT, INPUT ARRAY, DISPLAY ARRAY or CONSTRUCT statement.** *Description:* The predefined keyword DIALOG has not been used within a valid interactive statement, which must be terminated appropriately with END INPUT, END INPUT ARRAY, END DISPLAY ARRAY, or END CONSTRUCT. *Solution:* Review your code and make the necessary corrections.
-8048	**An error occurred while preprocessing the file '*name*'. Compilation ends.** *Description:* The FGL preprocessor could not parse the whole source file and stopped compilation. *Solution:* Review the source code and check for not well formed & preprocessor macros.
-8049	**The program cannot ACCEPT (INPUT|CONSTRUCT|DISPLAY) at this point because it is not immediately within (INPUT|INPUT ARRAY|CONSTRUCT|DISPLAY ARRAY) statement.** *Description:* ACCEPT XXX has not been used within a valid interactive statement, which must be terminated appropriately with END INPUT, END PROMPT, or END INPUT ARRAY. *Solution:* Review your code and make the necessary corrections.
-8050	**Dom: Invalid XML data found in source.** *Description:* ACCEPT DISPLAY has not been used within a valid DISPLAY ARRAY statement, which must be terminated with END DISPLAY ARRAY. *Solution:* Review your code and make the necessary corrections.
-8051	**Sax: Invalid processing instruction name.** *Description:* The om.SaxDocumentHandler.processingInstruction() does not allow invalid processing instruction names such as 'xml'. *Solution:* <?xml ..?> is not a processing instruction, it is reserved to define the XML file text declaration. You must use another name.
-8052	**Illegal input sequence. Check LANG.** *Description:* The compiler encountered an invalid character sequence. The source file uses a character sequence which does not match the locale settings (LANG). *Solution:* Check source file and locale settings.
-8053	**Unknown preprocessor directive '*name*'.** *Description:* The preprocessor directive shown in the error message is not a known directive. *Solution:* Check for typo errors and read the documentation for valid preprocessor directives.
-8054	**Unexpected preprocessor directive.** *Description:* The preprocessor encountered an unexpected directive.

	Solution: Remove the directive.
-8055	**The resource file '*name*' contains unexpected data.** *Description:* The XML resource file shown in the error message does not contain the expected nodes. For example, you try to load a ToolBar with ui.Interface.loadActionDefaults(). *Solution:* Check if the XML file contains the node types expected for this type of resource.
-8056	**XPath: Unclosed quote at position *integer*.** *Description:* The XPath parser found an unexpected quote at the given position. *Solution:* Review the XPath expression.
-8057	**XPath: Unexpected character '*character*' at position *integer*.** *Description:* The XPath parser found an unexpected character at the given position. *Solution:* Review the XPath expression.
-8058	**XPath: Unexpected token/string '*name*' at position *integer*.** *Description:* The XPath parser found an unexpected token or string at the given position. *Solution:* Review the XPath expression.
-8059	**SQL statement or language instruction with specific SQL syntax.** *Description:* The compiler found an SQL statement which is using a database specific syntax. This statement will probably not run on other database servers as the current. *Solution:* Review the SQL statement an use standard/common syntax and features.
-8060	**Spacer items are not allowed inside a SCREEN sections.** *Description:* The form contains spacer items in a SCREEN section, while these are only allowed in LAYOUT. *Solution:* Review the form specification file.
-8061	**A TABLE row should not be defined on multiple lines.** *Description:* All columns of a row in a TABLE container must be in a single line. *Solution:* Use a SCROLLGRID if you want to show row cells on multiple lines.
-8062	**DOM(ui): insert of removed node is not allowed.** *Description:* It is not possible to insert a removed node in the AUI document. *Solution:* Review the code.
-8063	**The client connection timed out, exiting program.** *Description:* The runtime system could not establish the connection with the front-end after a given time. This can for example happen during a file transfer, when the front-end takes too much time to answer to the runtime system. *Solution:* Check that your network connection is working properly.

-8064	**File transfer interrupted.** *Description:* An interruption was caught during a file transfer. *Solution:* File could not be transferred, you need to redo the operation.
-8065	**Network error during file transfer.** *Description:* An socket error was caught during a file transfer. *Solution:* Check that your network connection is working properly.
-8066	**Could not write destination file for file transfer.** *Description:* The runtime system could not write the destination file for a transfer. *Solution:* Make sure the file path is correct and check that user has write permissions.
-8067	**Could not read source file for file transfer.** *Description:* The runtime system could not read the source file to transfer. *Solution:* Make sure the file path is correct and check that user has read permissions.
-8068	**File transfer protocol error (invalid state).** *Description:* The runtime system encountered a problem during a file transfer. *Solution:* A network failure has probably raised this error.
-8069	**File transfer not available.** *Description:* File transfer feature is not supported. *Solution:* Make sure the front-end supports file transfer.
-8070	**The localized string file '*name*' is corrupted.** *Description:* The shown string resource file is invalid (probably invalid multi-byte characters corrupt the file). *Solution:* Check for locale settings (LANG), make sure the .str source uses valid characters and recompile it.
-8071	**The item '*name*' has been defined more than once.** *Description:* The form file defines several elements of the same type with the same name. *Solution:* Review the form file and use unique identifiers.
-8072	**Statement must terminate with ';'.** *Description:* An ESQL/C preprocessor directive is not terminated with a semi-colon. *Solution:* Add a semi-colon to the end of the directive.
-8073	**Invalid 'include' directive file name.** *Description:* An include preprocessor directive is using an invalid file name. *Solution:* Check the file name.
-8074	**A &elif directive found without corresponding &if,&ifdef or &ifndef directive.** *Description:* The preprocessor found an **&elif** directive with no

	corresponding **&if**. *Solution:* Add the **&if** directive before the **&elif**, or remove the **&elif**.
-8075	**The compiler plugin *name* could not be loaded.** *Description:* fglcomp could not load the plugin because it was not found. *Solution:* Make sure the plugin exists and can be loaded.
-8076	**The compiler plugin *name* does not implement the required interface.** *Description:* fglcomp could not load the plugin because the interface is invalid. *Solution:* Check if the plugin corresponds to the version of the compiler.
-8077	**The attribute '*name*' has been defined more than once.** *Description:* The variable attribute shown in the error message was defined multiple times. *Solution:* Review the variable definition and remove duplicated attributes.
-8078	**The attribute '*name*' is not allowed.** *Description:* The variable attribute shown in the error message is not allowed for this type of variable. *Solution:* Review the possible variable attributes.
-8079	**An error occurred while parsing the XML file.** *Description:* The runtime system could not parse an XML file, which is probably not using a valid XML format. *Solution:* Check for XML format typos and if possible, validate the XML file with a DTD.
-8080	**Could not open xml file.** *Description:* The specified XML file cannot be opened. *Solution:* Make sure the file exists and has access permissions for the current user.
-8081	**Invalid multibyte character has been encountered.** *Description:* A compiler found an invalid multi-byte character in the source and cannot compile the form or module. *Solution:* Check locale settings (LANG) and verify if there are no invalid characters in your sources.
-8082	**The item '*name*' is used in an invalid layout context.** *Description:* The form item name is used in a layout part which does not support this type of form item. This error occurs for example when you try to define a BUTTON as a TABLE column. *Solution:* Review your form definition file and use correct item types.
-8083	**NULL pointer exception.** *Description:* The program is using calling a method thru an object variable which is NULL. *Solution:* You must assign an object reference to the variable before calling a method.

-8084	**Can't open socket.** *Description:* The channel object failed to open a client socket. *Solution:* Make sure the IP address and port are correct.
-8085	**Unsupported mode for 'open socket'.** *Description:*You try to open a channel with an unsupported mode. *Solution:* See channel documentation for supported modes.
-8086	**The socket connection timed out.** *Description:* Socket connect could not be established and timeout expired. *Solution:* Check all network layers and try again.
-8087	**File error in BYTE or TEXT readFile or writeFile.** *Description:* File I/O error occured while reading from or writing to a file. *Solution:* Verify the file name, content and access permissions.
-8088	**The dialog attribute 'attribute-name' is not supported.** *Description:* A dialog instruction was declared with an ATTRIBUTES clause containing an unsupported option. *Solution:* Review the ATTRIBUTES clause and remove unsupported option.
-8089	**Action 'action-name' not found in dialog.** *Description:* You try to use and action name that does not exist in the current dialog. *Solution:* Verify if name of the action is defined by an ON ACTION clause.
-8090	**Field 'field-name' already used in this DIALOG.** *Description:* The DIALOG instruction binds the same field-name or screen-record multiple times. *Solution:* Review all sub-dialog blocks and check the field-names / screen-records.
-8091	**Clause 'clause-name' already used.** *Description:* You have defined the same dialog control block multiple times. For example, AFTER ROW was defined twice. *Solution:* Remove the un-necessary control blocks.

General Terms used in this documentation

This documentation uses several terms that must be clarified for a good understanding. Here is a short description for all these terms:

Product

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

End User

The *End User* is the person that uses the Product; that person works on hardware called the Workstation.

Programs

The *Programs* are the software components that are developed and distributed by the supplier of the Product. *Programs* typically implement business rules and processing, usually called Business Logic. *Programs* are executed by the Runtime System on the Application Server machine. These components are typically p-code modules, forms and additional files.

Developer

The *Developer* is the person in charge of the conception and implementation of the Product components.

Application Data

Application Data defines the data manipulated by the Product. It is typically managed by one or more Database Systems. The *Application Data* has a volatile state when loaded in the Runtime System, and it has a static state when stored in the Database System.

Database

The *Database* is a logical entity regrouping the Application Data. It is managed by the Database System.

Database System

The *Database System* is the software that manages data storage and searching; it is usually installed on the Database Server machine and is supported by a tier software vendor. It is the software managing the Data in the Three-Tier C/S model.

Development Database

The *Development Database* is the Database used in the application development environment.

Production Database

The *Production Database* is the Database used on production sites.

Front End

The *Front End* is the software that manages the display of the User Interface on the Workstation machine. This component is historically called "The Client", in a thin Client/Server context. It is the software managing the Presentation in the Three-Tier C/S model.

Runtime System

The *Runtime System* is the software that manages the execution of the Programs, where the Business Logic is processed. It is typically implemented by the *Dynamic Virtual Machine* (DVM) and historically called "The Runner". It is the software managing the Processing in the Three-Tier C/S model.

User Interface

The *User Interface* defines the parts of the Programs that interact with the end user, including interactive elements like windows, screens, input fields, buttons and menus. It is displayed on the Workstation. This can typically be implemented by different kinds of Front Ends, based on ASCII terminals, graphical platforms (MS Windows, X11) or even through web protocols like HTML over HTTP.

Graphical User Interface

The *Graphical User Interface* (GUI) mode identifies the user interface displayed on a remote machine via a Front End.

Text User Interface

The *Text User Interface* (TUI) mode identifies the user interface displayed on ASCII terminals (TTY on UNIX or Console Window on MS Windows).

Workstation

The *Workstation* identifies the hardware used by the End User to interact with the Product. It can be an ASCII Terminal, a PC, a diskless station or even a cellular phone, as long as a Front End is available on that hardware.

Genero BDL Tutorial Summary

If you are a developer new to **Genero** and the **Business Development Language**, this tutorial is designed for you, to explain concepts and provide code examples for some of the common business-related tasks. The only prerequisite knowledge is familiarity with relational databases and SQL.

The chapters contain a series of programs that range in complexity from simply displaying a database row to more advanced topics, such as handling arrays and master/detail relationships. Each chapter has a general discussion of the features and programming techniques used in the example programs, with annotated code samples. The examples in later chapters build on concepts and functions explained in earlier chapters. These programs have the BDL keywords in uppercase letters; this is a convention only. For ease in reading, the BDL keywords are colored green. The line numbers in the programs are for reference only; they are not a part of the BDL code.

If you wish to run the example programs or try out the programming techniques described in this tutorial, See Testing the Example Programs for the requirements.

For an overview of Genero BDL, see Introduction: BDL Concepts.

Tutorial Chapters

Chapter	Description
1 - Overview	This chapter provides an overview of the Tutorial and a description of the database schema and sample data used for the example programs.
2 - Using BDL	This chapter illustrates the structure of a BDL program and some of the BDL statements that perform some common tasks - display a text message to the screen, connect to a database and retrieve data, define variables, and pass variables between functions.
3 - Displaying Data(Windows/Forms)	This chapter illustrates opening a window that contains a form to display information to the user. An SQL statement is used to retrieve the data from a database table. A form specification file is defined to display the values retrieved. The actions that are available to the user are defined in the source code, tied to buttons that display on the form.
4 - Searching the Database(Query by Example)	The program in this chapter allows the user to search a database by entering criteria in a form. The search criteria is used to build an SQL SELECT statement to retrieve the desired database rows. A cursor is defined in the program, to allow the user to scroll back and forth between the rows of the result set. Testing the success of the SQL statements and handling errors is illustrated.
5 - Enhancing the Form	Program forms can be displayed in a variety of ways. This chapter illustrates adding a Toolbar or a Topmenu (pulldown menu) by modifying the form specification file, changing the window's appearance, and disabling/enabling actions. The example programs in this chapter use some of the action defaults defined by Genero BDL to standardize the presentation of common actions to the user.
6 - Modifying Data (Insert/Update/Delete)	This program allows the user to insert/update/delete rows in the customer database table. Embedded SQL statements (UPDATE/INSERT/DELETE) are used to update the table, based on the values stored in the program record. SQL transactions, concurrency, and consistency are discussed. A dialog window is displayed to prompt the user to verify the deletion of a row.

7 - Displaying an Array of Data	Unlike the previous programs, the example in this chapter displays multiple customer records at once. The program defines a program array to hold the records, and displays the records in a form containing a table and a screen array. The example program is then modified to dynamically fill the array as needed. This program illustrates a library function - the example is written so it can be used in multiple programs, maximizing code re-use.
8 - Modifying an Array	The program in this chapter allows the user to view and change a list of records displayed on a form. As each record in the program array is added, updated, or deleted, the program logic makes corresponding changes in the rows of the corresponding database table.
9 - Creating Reports	This program generates a simple report of the data in the customer database table. The two parts of a report, the report driver logic and the report definition are illustrated. A technique to allow a user to interrupt a long-running report is shown.
10 -Using Localization	Localization support and localized strings allow you to internationalize your application using different languages, and to customize it for specific industry markets in your user population. This chapter illustrates the use of localized strings in your programs.
11 - Managing Master/Detail Forms	The form used by the program in this chapter contains fields from both the **orders** and **items** tables in the **custdemo** database, illustrating a master-detail relationship. Since there are multiple items associated with a single order, the rows from the items table are displayed in a table on the form. This chapter focuses on the master/detail form and the unique features of the corresponding program.
12 - Changing the User Interface Dynamically	This chapter focuses on using the classes and methods in the **ui** package of built-in classes to modify the user interface at runtime. Among the techniques illustrated are hiding or disabling form items; changing the text, style or image associated with a form item; loading a ComboBox from a database table; and adding Toolbars and Topmenus dynamically.

Testing the Example Programs

Before you can create the **custdemo** database and test the example programs, the following requirements must be met:

- You must have access to one of the supported relational database systems, and it must be up and running.
- Genero BDL must be installed, and the appropriate environment must be set to allow you to use it.
- The front-end client specific to your system (Genero Desktop Client, for example) must be installed.
- A text-editor to view and edit the program files must be available.

The example database is designed to be as generic as possible, so it can be implemented on various relational database systems.

The source code of example programs are provided in the **Tutorial** directory of the HTML documentation.

To set up the environment in order to run the example programs:

1. Using your database system software, create an empty database named **custdemo** with logging enabled.
2. Make a copy on your system of the **Tutorial** directory provided in the documentation.
3. From your copy of the Tutorial files:
    - execute the SQL statements in the file **custdemo.sql** to create tables in the **custdemo** database.
    - Execute the SQL statements in the file **loadcust.sql** to insert rows into the tables of the **custdemo** database.
4. Configure FGLPROFILE with **dbi.*** entries to connect to the database. See Connections for more details.
5. Set the FGLDBPATH environment variable to "**..**", in order to let compilers access the **custdemo.sch** database schema file from example sub-directories.

Tutorial Chapter 1: Overview

Summary:

- Overview
- The BDL Language
- The BDL Tutorial
- The Example Database (custdemo)
- The Sample data

See also: Introduction: BDL Concepts

Overview

Especially well-suited for large-scale, database-intensive business applications, Genero Business Development Language (**BDL)** is a **reliable, easy-to-learn high-level programming language** that allows application developers to:

- express business logic in a clear yet powerful syntax
- use SQL statements for database access to any of the supported databases
- localize your application to follow a specific language or cultural rules
- define user interfaces in an abstract, platform-independent manner
- define Presentation Styles to customize and standardize the appearance of the interface
- manipulate the user interface at runtime, as a tree of objects

The separation of business logic, user interface, and deployment provides maximum flexibility.

- The business logic is written in text files (.4gl source code modules) that interact with separate form files defining the user interface.
- Actions defined in the business logic are tied to action views (buttons, menu items, toolbar icons) in the form definition files, and respond to user interaction statements in the source code.
- Compiling a form definition file translates it into XML, which is used to display the user interface to various Genero clients running on different platforms.

You can **write once, deploy anywhere** - one production release supports all major versions of Unix, Linux, Windows, and Mac OS X.

Compiling, linking, and deploying BDL applications, and additional resources for developers, are discussed in Introduction: BDL Concepts.

The BDL Language

The Genero Business Development Language includes:

- Program flow control
- Conditional logic
- SQL statement support
- Connection management
- Error handling
- Localized strings

Dynamic SQL management allows you to execute any SQL statement that is valid for your database version, in addition to those that are included as part of the language. The statement can be hard coded or created at runtime, with or without SQL parameters, returning or not returning a result set.

High-level **BDL user interaction statements** substitute for the many lines of code necessary to implement common business tasks, mediating between the user and the user interface in order to:

- Provide a selection of actions to the user (MENU)
- Allow the user to enter database search criteria on a form (CONSTRUCT)
- Display information from database tables (DISPLAY, DISPLAY ARRAY)
- Allow the user to modify the contents of database tables (INPUT, INPUT ARRAY)

Multiple dialogs allow a Genero program to handle interactive statements isuch as the above in parallel.

In addition, built-in classes and methods, and built-in functions are provided to assist you in your program development.

The BDL Tutorial

The chapters in this tutorial describe the basic functionality of BDL. Annotated code examples in each chapter guide you through the steps to implement the features discussed above. In addition, complete source code programs for the examples are included in the Tutorial directory of the BDL documentation. See the Tutorial Summary for a description of each chapter.

The example programs interact with an demo database, the **custdemo** database, containing store and order information for a fictional retail chain.

If you wish to test the example programs on your own system, see Testing the Programs for information about the software and sample data that must be installed and configured.

The Example Database (custdemo)

The following SQL statements create the tables for the custdemo database; these statements are in the file **custdemo.sql** in the Tutorial subdirectory of the documentation.

```
create table customer(
    store_num     integer not null,
    store_name    char(20) not null,
    addr          char(20),
    addr2         char(20),
    city          char(15),
    state         char(2),
    zipcode       char(5),
    contact_name  char(30),
    phone         char(18),
    primary key (store_num)
);
create table orders(
    order_num     integer not null,
    order_date    date not null,
    store_num     integer not null,
    fac_code      char(3),
    ship_instr    char(10),
    promo         char(1) not null,
    primary key (order_num)
);
create table factory(
    fac_code      char(3) not null,
    fac_name      char(15) not null,
    primary key (fac_code)
);
create table stock(
    stock_num     integer not null,
    fac_code      char(3) not null,
    description   char(15) not null,
    reg_price     decimal(8,2) not null,
    promo_price   decimal(8,2),
    price_updated date,
    unit          char(4) not null,
    primary key (stock_num, fac_code)
);
create table items(
    order_num     integer not null,
    stock_num     integer not null,
    quantity      smallint not null,
    price         decimal(8,2) not null,
    primary key (order_num, stock_num)
);
create table state(
    state_code char(2) not null,
    state_name char(15) not null,
    primary key (state_code)
```

```
);
```

The Sample Data

The following sample data for the custdemo database is contained in the file **loadcust.sql** in the Tutorial subdirectory of the documentation.

Customer table

101|Bandy's Hardware|110 Main| |Chicago|IL|60068|Bob Bandy|630-221-9055|
102|The FIX-IT Shop|65W Elm Street Sqr.| |Madison|WI|65454| |630-34343434|
103|Hill's Hobby Shop|553 Central Parkway| |Eau Claire|WI|54354|Janice Hilstrom|666-4564564|
104|Illinois Hardware|123 Main Street| |Peoria|IL|63434|Ramon Aguirra|630-3434334|
105|Tools and Stuff|645W Center Street| |Dubuque|IA|54654|Lavonne Robinson|630-4533456|
106|TrueTest Hardware|6123 N. Michigan Ave| |Chicago|IL|60104|Michael Mazukelli|640-3453456|
202|Fourth Ill Hardware|6123 N. Michigan Ave| |Chicago|IL|60104|Michael Mazukelli|640-3453456|
203|2nd Hobby Shop|553 Central Parkway| |Eau Claire|WI|54354|Janice Hilstrom|666-4564564|
204|2nd Hardware|123 Main Street| |Peoria|IL|63434|Ramon Aguirra|630-3434334|
205|2nd Stuff|645W Center Street| |Dubuque|IA|54654|Lavonne Robinson|630-4533456|
206|2ndTest Hardware|6123 N. Michigan Ave| |Chicago|IL|60104|Michael Mazukelli|640-3453456|
302|Third FIX-IT Shop|65W Elm Street Sqr.| |Madison|WI|65454| |630-34343434|
303|Third Hobby Shop|553 Central Parkway| |Eau Claire|WI|54354|Janice Hilstrom|666-4564564|
304|Third IL Hardware|123 Main Street| |Peoria|IL|63434|Ramon Aguirra|630-3434334|
305|Third and Stuff|645W Center Street| |Dubuque|IA|54654|Lavonne Robinson|630-4533456|
306|Third Hardware|6123 N. Michigan Ave| |Chicago|IL|60104|Michael Mazukelli|640-3453456|

Orders table

1|04/04/2003|101|ASC|FEDEX|N|
2|06/06/2006|102|ASC|FEDEX|Y|
3|06/10/2006|103|PHL|FEDEX|Y|
4|06/10/2006|104|ASC|FEDEX|Y|
5|07/06/2006|101|ASC|FEDEX|Y|
6|07/16/2006|105|ASC|FEDEX|Y|
7|08/04/2006|104|PHL|FEDEX|Y|
8|08/16/2006|101|ASC|FEDEX|Y|
9|08/23/2006|101|ASC|FEDEX|Y|
10|09/06/2006|106|PHL|FEDEX|Y|

Items table

1|456|10|5.55|
1|310|5|12.85|
1|744|60|250.95|
2|456|15|5.55|
2|310|2|12.85|
3|323|2|0.95|
4|744|60|250.95|
4|456|15|5.55|
5|456|12|5.55|
5|310|15|12.85|
5|744|6|250.95|
6|456|15|5.55|
6|310|2|12.85|
7|323|10|0.95|
8|456|10|5.55|
8|310|15|12.85|
9|744|20|250.95|
10|323|200|0.95|

Stock table

456|ASC|lightbulbs|5.55|5.0|01/16/2006|ctn|
310|ASC|sink stoppers|12.85|11.57|06/16/2006|grss|
323|PHL|bolts|0.95|0.86|01/16/2006|20/b|
744|ASC|faucets|250.95|225.86|01/16/2006|6/bx|

Factory table

ASC|Assoc. Std. Co.|
PHL|Phelps Lighting|

State table

IL|Illinois|
IA|Iowa|
WI|Wisconsin|

Tutorial Chapter 2: Using BDL

Summary:

- A simple BDL program
- Writing BDL programs
- Compiling and Executing the program
- Debugging BDL programs
- The "Connect to database" program
  - Connecting to a database
  - Variable Definition
  - Variable Scope
  - Passing Variables
  - Retrieving data from a database table
  - Example: connectdb.4gl

A simple BDL program

This simple example displays a text message to the screen, illustrating the structure of a BDL program.

Because Genero BDL is a structured programming language as well as a 4th generation language, executable statements can appear only within logical sections of the source code called program blocks. This can be the MAIN statement, a FUNCTION statement, or a REPORT statement. (Reports are discussed in Chapter 9.)

Execution of any program begins with the special, required program block MAIN**,** delimited by the keywords MAIN and END MAIN. The source module that contains MAIN is called the main module.

The FUNCTION statement is a unit of executable code, delimited by FUNCTION and END FUNCTION, that can be called by name. In a small program, you can write all the functions used in the program in a single file. As programs grow larger, you will usually want to group related functions into separate files, or source modules. Functions are available on a global basis. In other words, you can reference any function in any source module of your program.

The following example is a small but complete Genero BDL program, **simple.4gl**

Program simple.4gl

```
01 -- simple.4gl
02
03 MAIN
04     CALL sayIt()
05 END MAIN
06
07 FUNCTION sayIt()
```

```
08    DISPLAY "Hello, world!"
09 END FUNCTION  -- sayIt
```

Notes:

- Line 01 simply lists the filename as a comment, which will be ignored by BDL.
- Line 03 indicates the start of the MAIN program block.
- Line 04 Within the MAIN program block, the CALL statement is used to invoke the function named sayIt. Although no arguments are passed to the function sayIt, the empty parentheses are required. Nothing is returned by the function.
- Line 05 defines the end of the MAIN program block. When all the statements within the program block have been executed the program will terminate automatically.
- Line 07 indicates the start of the FUNCTION **sayIt**.
- Line 08 uses the DISPLAY statement to display a text message, enclosed within double quotes, to the user. Because the program has not opened a window or form, the message is displayed on the command line.
- Line 09 indicates the end of the FUNCTION. The comment (-- sayIt) is optional. After the message is displayed, control in the program is returned to the MAIN function, to line 05, the line immediately following the statement invoking the function. As there are no additional statements to be executed (END MAIN has been reached), the program terminates.

Writing BDL Programs

- Genero BDL source code is written as text in a source module (a file with a filename extension of **.4gl**).
- BDL statements do not require a statement terminator.
- You can begin a **comment** that terminates at the end of the current line with a pair of minus signs (--) or #. Curly braces { } can be used to delimit comments that occupy multiple lines.
- All white space in a source code module is treated as a single space, so you are free to use indentations and white space for clarity.
- Although the language keywords in this example and throughout the tutorial are in all-capitals, this is just a convention used in these documents. You may write keywords in lowercase, or any combination of capitals and lowercase you prefer.
- The line numbers shown in all the code examples are not a part of the code; they simply link the notes for the programs with the correct program lines.

Compiling and Executing the Program

The following tools can be used to compile and execute the **simple** program from the command line.

1. Create the database schema files if they have not already been created:

   ```
   fgldbsch -db custdemo
   ```

2. Compile the single module program:

   ```
   fglcomp simple.4gl
   ```

3. Execute the program:

   ```
   fglrun simple.42m
   ```

Tip:

1. You can compile and run the program without specifying the file extensions:

   ```
   fglcomp simple
   fglrun simple
   ```

2. You can do this in one command line, adding the -M option for errors:

   ```
   fglcomp -M simple && fglrun simple
   ```

Debugging a BDL Program

You can use the command line debugger to search for programming errors. The command line debugger is integrated in the runtime system. You typically start a program in debug mode by passing the **-d** option to **fglrun**.

The following lines illustrate a debug session with the previous program sample:

```
fglrun -d simple

(fgldb) break main
Breakpoint 1 at 0x00000000: file simple.4gl, line 2.
(fgldb) run
Breakpoint 1, main() at simple.4gl:2
2         CALL sayIt()
(fgldb) step
sayit() at simple.4gl:6
6         DISPLAY "Hello, world!"
(fgldb) next
Hello, world!
7     END FUNCTION -- sayIt
(fgldb) continue
Program existed normally.
(fgldb) quit
```

For more details, see the Debugger reference.

The "Connect to database" Program

This program illustrates connecting to a database and retrieving data, defining variables, and passing variables between functions. A row from the **customer** table of the **custdemo** example database is retrieved by an SQL statement and displayed to the user.

Connecting to the Database

A Database Connection is a session of work, opened by the program to communicate with a specific database server, in order to execute SQL statements as a specific user. To connect to a database server, most database engines require a name to identify the server, a name to identify the database entity, a user name and a password.

Connecting through the Open Database Interface, the database can be specified directly, and the specification will be used as the data source. Or, you can define the database connection parameters indirectly in the FGLPROFILE configuration file, and the database specification will be used as a key to read the connection information from the file. This technique is flexible; for example, you can develop your application with the database name "custdemo" and connect to the real database "custdemo1" in a production environment.

The CONNECT instruction opens a session in multi-session mode, allowing you to open other connections with subsequent CONNECT instructions (to other databases, for example). If you have multiple connections open, you can use the SET CONNECTION instruction to switch to a specific session; this suspends other opened connections. The DISCONNECT instruction can be used to disconnect from specific sessions, or from all sessions. The end of a program disconnects all sessions automatically.

The *user name* and *password* can be specified in the CONNECT instruction, or defaults can be defined in FGLPROFILE. Otherwise, the user name and password provided to your operating system will generally be used for authentication.

```
CONNECT TO "custdemo"
```

Variable Definition

A Variable contains volatile information of a specific BDL data type. Variables must be declared before you use them in your program, using the DEFINE statement. After definition, variables have default values based on the data type.

```
DEFINE cont_ok INTEGER
```

You can use the LIKE keyword to declare a variable that has the same data type as a specified column in a database schema. The column data type defined by the database

schema must be supported by the language. A SCHEMA statement must define the database name, identifying the database schema files to be used. The column data types are read from the schema file during compilation, not at runtime. Make sure that your schema files correspond exactly to the production database.

```
DEFINE store_name LIKE customer.store_name
```

Genero BDL allows you to define structured variables as records or arrays. Examples of this are included in later chapters.

Variable Scope

Variables defined in a FUNCTION, REPORT or MAIN program block have *local scope* (are known only within the program block). The DEFINE statement declares the variables and causes memory to be allocated for them. DEFINE must precede any executable statements within the same program block.

A Variable defined with *modular scope* can have its value set and can be used in any function within a single source-code module. The DEFINE statement must appear at the top of the module, before any program blocks. Memory for module variables is allocated when the program starts, and is released when the program ends.

A Variable with *local* scope can have its value set and can be used only within the function in which it is defined. The DEFINE statement must be the first statement in the function. Memory for the variable is allocated when the function is called by a program, and is released when the function ends.

A compile-time error occurs if you declare the same name for two variables that have the same scope. You can, however, declare the same name for variables that differ in their scope. For example, you can use the same identifier to reference different local variables in different program blocks. If a local variable has the same name as a module variable, then the local variable takes precedence inside the program block in which it is declared. Elsewhere in the same source-code module, the name references the module variable.

Passing Variables

Functions can be invoked explicitly using the CALL statement. Variables can be passed as arguments to a function when it is invoked. The parameters can be variables, literals, constants, or any valid expressions. Arguments are separated by a comma. If the function returns any values, the RETURNING clause of the CALL statement assigns the returned values to variables in the calling routine. The number of input and output parameters is static.

The function that is invoked must have a RETURN instruction to transfer the control back to the calling function and pass the return values. The number of returned values must correspond to the number of variables listed in the RETURNING clause of the CALL statement invoking this function. If the function returns only one unique value, it can be used as a scalar function in an expression.

```
CALL myfunc()

CALL newfunc(var1) RETURNING var2, var3

LET var2 = anotherfunc(var1)

IF testfunc1(var1) == testfunc2(var1) THEN
```

Retrieving data from a database

Using Static SQL, an embedded SQL SELECT statement can be used to retrieve data from a database table into program variables. If the SELECT statement returns only one row of data, you can write it directly as a procedural instruction, using the INTO clause to provide the list of variables where the column values will be fetched. If the SELECT statement returns more than one row of data, you must declare a database cursor to process the result set.

Example: connectdb.4gl

Note: The line numbers shown in the examples in this tutorial are not part of the BDL code; they are used here so specific lines can be easily referenced. The BDL keywords are shown in uppercase, as a convention only. The keywords also appear in green in this documentation.

Program connectdb.4gl

```
01 -- connectdb.4gl
02 SCHEMA custdemo
03
04 MAIN
05   DEFINE
06    m_store_name LIKE customer.store_name
07
08   CONNECT TO "custdemo"
09
10   CALL select_name(101)
11       RETURNING m_store_name
12   DISPLAY m_store_name
13
14   DISCONNECT CURRENT
15
16 END MAIN
17
18 FUNCTION select_name(f_store_num)
19   DEFINE
20    f_store_num  LIKE customer.store_num,
21    f_store_name LIKE customer.store_name
22
23   SELECT store_name INTO f_store_name
24      FROM customer
25     WHERE store_num = f_store_num
```

```
26
27   RETURN f_store_name
28
29 END FUNCTION  -- select_name
```

Notes:

- Line 02 The SCHEMA statement is used to define the database schema files to be used as **custdemo.** The LIKE syntax has been used to define variables in the module.
- Lines 05 and 06 Using DEFINE the local variable **m_store_name** is declared as being LIKE the store_name column; that is, it has the same data type definition as the column in the **customer** table of the custdemo database.
- Line 08 A connection in multi-session mode is opened to the custdemo database, with connection parameters defined in FGLPROFILE. Once connected to the database server, a current database session is started. Any subsequent SQL statement is executed in the context of the current database session.
- Line 10 The **select_name** function is called, passing the literal value **101** as an argument. The function returns a value to be stored in the local variable **m_store_name**.
- Line 12 The value of **m_store_name** is displayed to the user on the standard output.
- Line 14 The DISCONNECT instruction disconnects you from the current session. As there are no additional lines in the program block, the program terminates.
- Line 18 Beginning of the definition of the function **select_name**. The value 101 that is passed to the function will be stored in the local variable **f_store_num**.
- Lines 19 thru 21 Defines multiple local variables used in the function, separating the variables listed with a comma. Notice that a variable must be declared with the same name and data type as the parameter listed within the parenthesis in the function statement, to accept the passed value.
- Lines 23 thru 25 Contains the embedded SELECT ... INTO SQL statement to retrieve the store name for store #101. The store name that is retrieved will be stored in the **f_store_name** local variable. Since the store number is unique, the WHERE clause ensures that only a single row will be returned.
- Line 27 The RETURN statement causes the function to terminate, returning the value of the local variable **f_store_name**. The number of variables returned matches the number declared in the RETURNING clause of the CALL statement invoking the function. Execution of the program continues with line 12.

Compiling and Executing the Program

1. Create the database schema files if they have not already been created:

   ```
   fgldbsch -db custdemo
   ```

2. Compile the single module program:

```
fglcomp connectdb.4gl
```

3. Execute the program:

```
fglrun connectdb.42m
```

Tutorial Chapter 3: Displaying Data (Windows/Forms)

Summary:

- Application Overview
- The .4gl File- Program Logic
  - Opening Windows and Forms
  - Interacting with the User
    - Defining Actions - the MENU statement
    - Displaying Messages and Errors
    - Example: dispcust.4gl (MAIN)
  - Retrieving and Displaying Data
    - Defining a Record
    - Using SQL to Retrieve Data
    - Displaying a Record
    - Example: dispcust.4gl (query_cust function)
- The Form Specification File
  - Overview
  - SCHEMA section (optional)
  - The ACTION DEFAULTS, TOPMENU, and TOOLBAR sections (optional)
  - LAYOUT section
  - TABLES section (optional)
  - ATTRIBUTES section
  - INSTRUCTIONS section (optional)
  - Example: custform.per
- Compiling the program and form

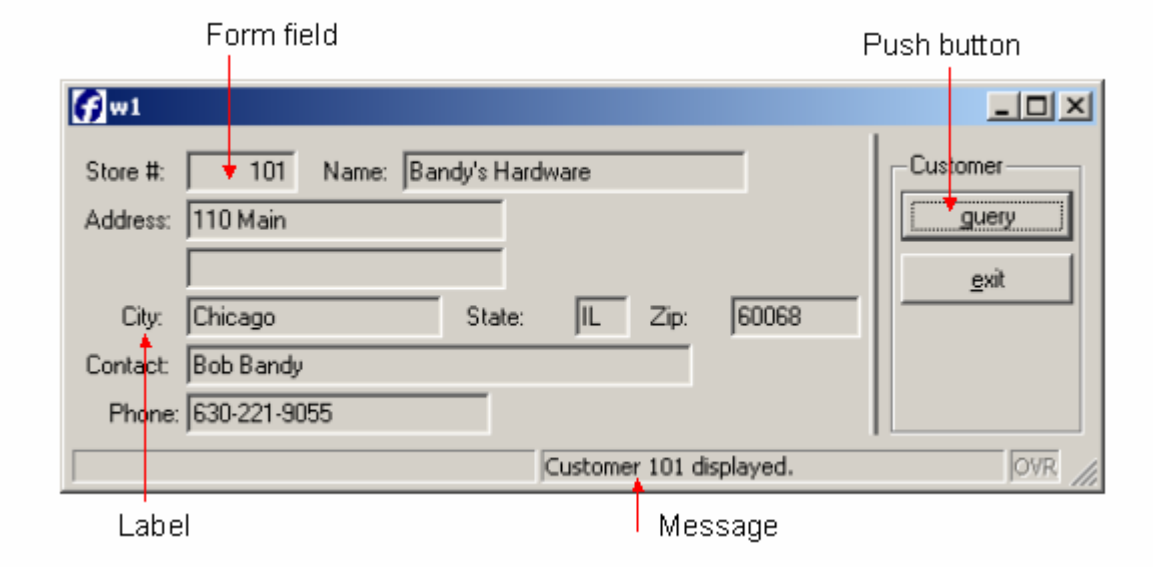
Application Overview

This example program opens a WINDOW containing a FORM to display information to the user. The appearance of the form is defined in a separate form definition file. The program logic to display information on the form is written in the .4gl program module. The same form file can be used with different applications. This separation of user interface and business logic provides maximum flexibility.

The options to retrieve data or exit are defined as *actions* in a MENU statement in the .4gl file. By default, push buttons are displayed on the form corresponding to the actions listed in the MENU statement. When the user presses the "query" button, the code listed for the action statement is executed - in this case, an SQL SELECT statement retrieves a single row from the customer table and displays it on the form.

A FORM can contain form fields for entering and displaying data; explanatory text (labels); and other form objects such as Buttons, Topmenus (dropdown menus), toolbar icons, folders, tables, and CheckBoxes. Form objects that are associated with an action

are called *action views*. Messages providing information to the user can be displayed on the form.



Display on Windows platforms

The .4gl File - Opening Windows and Forms

A program creates a window with the OPEN WINDOW instruction, and destroys a window with the CLOSE WINDOW instruction. The OPEN WINDOW ... WITH FORM instruction can be used to automatically open a window containing a specified form:

```
OPEN WINDOW custwin WITH FORM "custform"
```

When you are using a graphical front end, windows are created as independent resizable windows. By default windows are displayed as normal application windows, but you can specify a Presentation Style. The standard window styles are defined in the default Presentation Style file (FGLDIR/lib/default.4st):

If the WITH FORM option is used in opening a window, the CLOSE WINDOW statement closes both the window and the form.

```
CLOSE WINDOW custwin
```

When the runtime system starts a program, it creates a default window named SCREEN. This default window can be used as another window, but it can be closed if not needed.

```
CLOSE WINDOW SCREEN
```

Note: The appropriate Genero Front-end Client must be running for the program to display the window and form.

The .4gl File - Interacting with the User

Defining Actions - the MENU statement

Your form can display options to the user using action views - buttons, dropdown menus (top menus), toolbars, and other items on the window. See Form Specification Files for a complete list of form items.

An action defined in the .4gl module, which identifies the program routine to be executed, can be associated with each action view shown on the form.. You define the program logic to be executed for each action in the .4gl module.

- In this BDL program, the MENU statement supplies the list of actions and the statements to be executed for each action. The actions are specified with ON ACTION clauses:

```
ON ACTION query
   CALL query_cust()
```

- The ON ACTION clause defines the action name and the statements to be executed for the action. The presentation attributes - title, font, comment, etc. - for the graphical object that serves as the action view are defined in a separate action defaults file, or in the Action Defaults section of the form file. This allows you to standardize the appearance of the views for common actions. Action Defaults are illustrated in chapter 5.

  You can also use ON ACTION clauses with some other interactive BDL statements, such as INPUT, INPUT ARRAY, DIALOG, and DISPLAY ARRAY.

- When the MENU statement in your program is executed, the action views for the actions (**query,** in the example) that are listed in the interactive MENU statement are enabled. Only the action views for the actions in the specific MENU statement are enabled, so you must be sure to include a means of exiting the MENU statement. If there is no action view defined in your form specification file for a listed action, a simple push button action view is automatically displayed in the window. Control is turned over to the user, and the program waits until the user responds by selecting one of enabled action views or exiting the form. Once an action view is selected, the corresponding program routine (action) is executed.

.See MENUs for a complete discussion of the statement and all its options.

Displaying Messages and Errors

The MESSAGE and ERROR statements are used to display text containing a message to the user. The text is displayed in a specific area, depending on the front end configuration and window style. The MESSAGE text is displayed until it is replaced by another MESSAGE statement or field comment. You can specify any combination of variables and strings for the text. BDL generates the message to display by replacing any variables with their values and concatenating the strings:

```
MESSAGE "Customer " || l_custrec.store_num , || " retrieved."
```

The Localized Strings feature can be used to customize the messages for specific user communities. This is discussed in Chapter 10.

Example: dispcust.4gl

This portion of the **dispcust.4gl** program connects to a database, opens a window and displays a form and a menu.

Program dispcust.4gl

```
01 -- dispcust.4gl
02 SCHEMA custdemo
03
04 MAIN
05
06   CONNECT TO "custdemo"
07
08   CLOSE WINDOW SCREEN
09   OPEN WINDOW custwin WITH FORM "custform"
10   MESSAGE "Program retrieves customer 101"
11
12   MENU "Customer"
13     ON ACTION query
14       CALL query_cust()
15     ON ACTION exit
16       EXIT MENU
17   END MENU
18
19   CLOSE WINDOW custwin
20
21   DISCONNECT CURRENT
22
23 END MAIN
```

Notes:

- Line 02 The **SCHEMA** statement is required since variables are defined as LIKE a database table in the function query_cust.
- Line 06 opens the connection to the **custdemo** database.
- Line 08 closes the default window named **SCREEN,** which is opened each time the runtime system starts a program containing interactive statements

- Line 09 uses the WITH FORM syntax to open a window having the identifier **custwin** containing the form identified as **custform.** The window name must be unique among all windows defined in the program. Its scope is the entire program. You can use the window's name to reference any open window in other modules with other statements. Although there can be multiple open windows, only one window may be current at a given time.
  By default, the window that opens will be a normal application window.
  The form identifier is the name of the compiled **.42f file** (**custform.42f**). The form identifier must be unique among form names in the program. Its scope of reference is the entire program.
- Line 10 displays a string as a MESSAGE to the user. The message will be displayed until it is replaced by a different string.
- Lines 12 through 17 contain the interactive MENU statement. By default, the menu options **query** and **exit** are displayed as buttons in the window, with C**ustomer** as the menu title. When the MENU statement is executed, the buttons are enabled, and control is turned over to the user.
  If the user selects the **query** button, the function **query_cust** will be executed. Following execution of the function, the action views (buttons in this case) are re-enabled and the program waits for the user to select an action again.
  If the user selects the **exit** button, the MENU statement is terminated, and the program continues with line 19.
- Line 19 The window **custwin** is closed**,** which automatically closes the form, removing both objects from the application's memory.
- Line 21 The program disconnects from the database; as there are no more statements in MAIN, the program terminates.

The .4gl File - Retrieving and Displaying Data

Defining a Record

In addition to defining individual variables, the DEFINE statement can define a record**,** a collection of variables each having its own data type and name. You put the variables in a record so you can treat them as a group. Then, you can access any member of a record by writing the name of the record, a dot (known as dot notation), and the name of the member.

```
DEFINE custrec RECORD
        store_num  LIKE customer.store_num
        store_name LIKE customer.store_name
      END RECORD
DISPLAY custrec.store_num
```

Your record can contain variables for the columns of a database table. At its simplest, you write `RECORD LIKE` *`tablename.*`* to define a record that includes members that match in data type all the columns in a database table. However, if your database schema changes often, it's best to list each member individually, so that a change in the structure of the database table won't break your code. Your record can also contain members that are not defined in terms of a database table.

Using SQL to Retrieve the Data

A subset of SQL, known as Static SQL, is provided as part of the BDL language and can be embedded in the program. At runtime, these SQL statements are automatically prepared and executed by the Runtime System.

```
SELECT store_num, store_name INTO custrec.* FROM customer
```

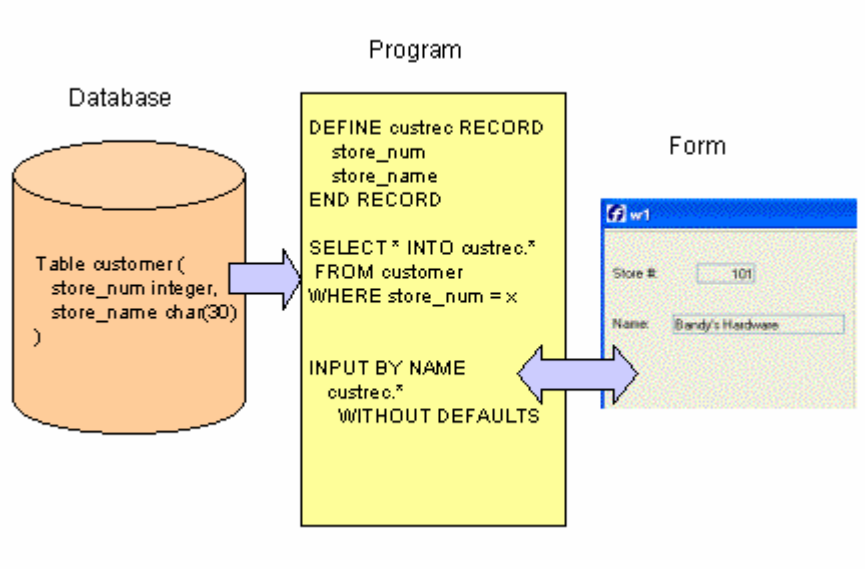
Only a limited number of SQL instructions are supported this way. However, Dynamic SQL Management allows you to execute any kind of SQL statement.

Displaying a Record: DISPLAY BY NAME

A common technique is to use the names of database columns as the names of both the members of a program record and the fields in a form. Then, the DISPLAY BY NAME statement can be used to display the program variables. By default, a screen record consisting of the form fields associated with each database table column is automatically created. BDL will match the variable name to the name of the form field, ignoring any record name prefix:

```
DISPLAY BY NAME custrec.*
```

The program variables serve as the intermediary between the database and the form that is displayed to the user. Values from a row in the database table are retrieved into the program variables by an SQL SELECT statement, and are then displayed on the form. In Chapter 6 you will see how the user can change the values in the form, resulting in changes to the program variables, which could then be used in SQL statements to modify the data in the database.



Example: dispcust.4gl (function query_cust)

This function retrieves a row from the customer table and displays it in a form.

Function query_cust

```
01 FUNCTION query_cust()      -- displays one row
02   DEFINE  l_custrec RECORD
03      store_num    LIKE customer.store_num,
04      store_name   LIKE customer.store_name,
05      addr         LIKE customer.addr,
06      addr2        LIKE customer.addr2,
07      city         LIKE customer.city,
08      state        LIKE customer.state,
09      zipcode      LIKE customer.zipcode,
10      contact_name LIKE customer.contact_name,
11      phone        LIKE customer.phone
12     END RECORD
13
14   SELECT store_num,
15          store_name,
16          addr,
17          addr2,
18          city,
19          state,
20          zipcode,
21          contact_name,
22          phone
23      INTO l_custrec.*
24      FROM customer
25      WHERE store_num = 101
26
27   DISPLAY BY NAME l_custrec.*
28   MESSAGE "Customer " || l_custrec.store_num ||
29           " displayed."
30 END FUNCTION
```

Notes:

- Line `01` is the beginning of the function **query_cust**. No variables are passed to the function.
- Lines `02` thru `12` DEFINE a record **l_custrec** as LIKE columns in the **customer** database table, listing each variable separately.
- Line `14` thru `25` SELECT **..** INTO can be used, since the statement will retrieve only one row from the database. The SELECT statement lists each column name to be retrieved, rather than using `SELECT *`. This allows for the possibility that additional columns might be added to a table at a future date. Since the SELECT list retrieves values for all the variables in the program record, in the order listed in the DEFINE statement, the shorthand `INTO l_custrec.*` can be used.
- Line `27` The names in the program record **l_custrec** match the names of screen fields on the form, so DISPLAY BY NAME can be used. `l_custrec.*` indicates that all of the members of the program record are to be displayed.

- Lines 28 and 29 A string for the MESSAGE statement is concatenated together using the double pipe (||) operator and displayed. The message consists of the string "**Customer** ", the value of **l_custrec.store_num**, and the string " **displayed**".

There are no additional statements in the function, so the program returns to the MENU statement, awaiting the user's next action.

The Form Specification File

Overview

You can specify the layout of a form in a form specification file, which is compiled separately from your program. The form specification file defines the initial settings for the form, which can be changed programmatically at runtime.

Form specification files have a file extension of **.per** . The structure of the form is independent of the use of the form. For example, one function can use a form to display a database row, another can let the user enter a new database row, and still another can let the user enter criteria for selecting database rows.

A Form can contain the following types of items:

- Container - groups other form items. Every form item must be in a container. A GRID is the basic container, frequently used to display a single row of database data. TABLE containers can provide record-list presentation in columns and rows. Other containers, such as a FOLDER or GROUP, provide additional options for organizing the data that is displayed.
- FormField - defines an area where the user can view and edit data. The data is stored in variables defined in the .4gl source code file. The EDIT formfield provides a simple line-edit field. Other form items, such as a COMBOBOX or RADIOGROUP, provide a user-friendly interface to the data stored in the underlying formfield. The data type of a formfield can be defined by a database table column, or it can be FORMONLY - defined specifically in the form.
- Action view - allows the user to trigger actions specified in the .4gl file. An Action view can be a BUTTON, Toolbar icon, or Topmenu option, for example.
- Other - items that enhance the display or provide read-only information (an IMAGE or LABEL, for example).

Each form and form item has attributes that control its appearance and behavior. See Form Specification Files, Form Specification File Attributes, and The Interaction Model for additional information about form items.

Styles from a Presentation Styles file can be applied to the form and form items.

A basic form specification consists of the following sections:

The SCHEMA section (optional)

This specifies the database schema file to be used when the form is compiled. It is required if any form items are defined as data types based on a column of a database table.

```
SCHEMA custdemo
```

The ACTION DEFAULTS, TOPMENU, and TOOLBAR sections (optional)

These sections are provided to allow you to define the decoration for action views (action defaults), as well as to define Topmenus and Toolbars for the form. In this case, the definitions are specific to the form. If your definitions are in external XML files instead, they can be applied to any form.

This is discussed in chapter 5.

The LAYOUT section

This section defines the appearance of a form using a layout tree of containers, which can hold other containers or can define a screen area. Some of the available containers are GRID, VBOX, HBOX, GROUP, FOLDER, and PAGE.

The simplest layout tree could have only a GRID container defining the dimensions and the position of the logical elements of a screen:

```
LAYOUT
  GRID
    grid-area
  END
END
```

The END keyword is mandatory to define the end of a container block.

The *grid-area* is delimited by curly braces. Within this area, you can specify the position of form items or interactive objects such as BUTTON, COMBOBOX, CHECKBOX, RADIOGROUP, PROGRESSBAR**,** etc.

Simple form fields, delimited by square brackets ([ ]), are form items used to display data and take input. Generally, the number of characters in the space between the brackets defines the width of the region to be used by the item. For example, in the grid-area, the following field could be defined:

```
[f01           ]
```

This form field has an item tag of **f01**, which will be used to link the field to its definition in the ATTRIBUTES section of the form specification.

Interactive form items, such as COMBOBOX, CHECKBOX, and RADIOGROUP, can be used instead of simple form fields to represent the values in the underlying formfield. Special width calculations are done for some of these form items, such as COMBOBOX, BUTTONEDIT, and DATEEDIT. If the default width generated by the form compiler does not fit, the - dash symbol can be used to define the real width of the item.

Text in the grid-area that is outside brackets is display-only text, as in the word **Company** below:

```
Company [f01            ]
```

The TABLES section (optional)

If a database table or database view is referenced elsewhere in the form specification file, in the ATTRIBUTES section for example, the table or view must be listed in the TABLES section:

```
TABLES
  customer
END
```

A *default screen record* is automatically created for the form fields associated with each table listed in this section.

The ATTRIBUTES section

The ATTRIBUTES section defines properties of the items used in the form.

Form Fields

For form fields (items that can be used to display data or take input) the definition is:

```
<item-type> <item-tag> = <item-name>, <attribute-list> ;
```

- The *item-type* defines the kind of graphical object which must be used to display the form element.
- The *item-tag* identifies the form item in the display area.
- The *item-name* provides the name of the form item.
- The optional *attribute-list* defines the aspect and behavior of the form item.

Examples:

```
EDIT f01 = customer.cust_num, REQUIRED;
COMBOBOX f03 = customer.state;
CHECKBOX f04 = formonly.propcheck;
```

The most commonly used item-type, EDIT, defines a simple line edit box for data input or display. This example uses an EDIT item-type for the form field f01.The COMBOBOX

and CHECKBOX item types present the data contained in the form fields f03 and f04 in a user-friendly way.

The *item-name* must specify a database column as the name of the display field, or must be FORMONLY (fields defined as FORMONLY are discussed in chapter 11.) Fields are associated with database columns only during the compilation of the form specification file, to identify the data type for the form field based on the database schema. After the form compiler identifies the data types, the association between fields and database columns is broken, and the item-name is associated with the screen record.

Form field and form item definitions can optionally include an *attribute-list* to specify the appearance and behavior of the item. For example, you can define acceptable input values, on-screen comments, and default values for fields; you can insure that a value is entered in the field during the input of a new row (REQUIRED); columns in a table can be specified as sortable or non-sortable; numbers and dates can be formatted for display; data entry patterns can be defined and input data can be upshifted or downshifted.

A form field can be an EDIT, BUTTONEDIT, CHECKBOX, COMBOBOX, DATEEDIT, IMAGE, LABEL, PROGRESSBAR, RADIOGROUP, or TEXTEDIT.

Other form items

For form items that are not form fields (BUTTON, CANVAS, GROUP, static IMAGE, static LABEL, SCROLLGRID, and TABLE) the definition is:

```
<item-type> <item-tag> : <item-name> , <attribute-list> ;
```

Examples:

```
BUTTON btn1: print, TEXT = "Print Report";
LABEL lab1 : label1, TEXT ="Customer";
```

The INSTRUCTIONS section (optional)

The INSTRUCTIONS section is used to define explicit screen records or screen arrays. This is discussed in Chapter 7.

Example: Form Specification File custform.per

This form specification file is used with the **dispcust.4gl** program to display program variables to the user. This form uses a layout with a simple GRID to define the display area.

custform.per

```
01 SCHEMA custdemo
02
```

```
03 LAYOUT
04   GRID
05   {
06    Store #:[f01  ] Name:[f02                ]
07    Address:[f03                ]
08            [f04                ]
09       City:[f05            ]State:[f6]Zip:[f07   ]
10    Contact:[f08                           ]
11      Phone:[f09               ]
12
13   }
14   END   --grid
15 END  -- layout
16
17 TABLES
18     customer
19 END
20
21 ATTRIBUTES
22 EDIT f01 = customer.store_num, REQUIRED;
23 EDIT f02 = customer.store_name, COMMENT="Customer name";
24 EDIT f03 = customer.addr;
25 EDIT f04 = customer.addr2;
26 EDIT f05 = customer.city;
27 EDIT f6  = customer.state;
28 EDIT f07 = customer.zipcode;
29 EDIT f08 = customer.contact_name;
30 EDIT f09 = customer.phone;
31 END
```

Notes:

- Line 01 lists the database schema file from which the form field data types will be obtained.
- Lines 03 through 15 delimit the LAYOUT section of the form.
- Lines 04 thru 14 delimit the GRID area, indicating what will be displayed to the user between the curly brackets on lines 05 and 13.
- Line 17 The TABLES statement is required since the field descriptions reference the columns of the database table **customer**.
- Within the grid area, the form fields have item tags linking them to descriptions in the ATTRIBUTES section, in lines 20 thru 28. As an example, f01 is the display area for a program variable having the same data type definition as the **store_num** column in the **customer** table of the **custdemo** database.
- Line 22 All of the item-tags in the form layout section are listed in the ATTRIBUTES section. For example, the item-tag f01 is listed as having an item-type of **EDIT**. This field will be used for display only in this program, but the same form will be used for input in a later program. An additional attribute, REQUIRED, indicates that when this form is used for input, an entry in the field f01 must be made. This prevents the user from trying to add a row with a NULL **store_num** to the **customer** table, which would result in an error message from the database.

- Line 23 The second field is defined with the attribute COMMENT, which specifies text to be displayed when this field gets the focus, or as a tooltip when the mouse goes over the field.

Compiling the Program and Form

When this form is compiled (translated) using the **fglform** tool, an XML file is generated that has a file extension of **.42f**. The runtime system uses this file along with your programs to define the Abstract User Interface**.**

Compile the form:

```
fglform custform.per
```

Compile the single module program:

```
fglcomp dispcust.4gl
```

Execute the program:

```
fglrun dispcust.42m
```

Tutorial Chapter 4: Query by Example

Summary:

- Implementing Query-by-Example
  - CONSTRUCT and STRING variables
  - PREPARE Statement
- Allowing the User to Cancel the Query
  - Pre-defined Actions (accept/cancel)
  - DEFER INTERRUPT and INT_FLAG
  - Conditional Logic: IF and CASE
  - The Query Program
  - Example: custmain.4gl
  - Example: custquery..4gl (function query_cust)
  - Example: custquery.4gl (function get_cust_cnt)
- Retrieving Data from the Database
  - Use of Cursors
  - The SQLCA.SQLCODE
  - Example: custquery.4gl (function cust_select)
  - Example: custquery.4gl (function fetch_cust)
  - Example: custquery.4gl (function fetch_rel_cust)
  - Example: custquery.4gl (function display_cust)
- Compiling and Linking a Multiple-module Program
- Modifying the Program to Handle Errors
  - WHENEVER ERROR statement
  - Negative SQLCA.SQLCODE
  - Using SQLERRMESSAGE
  - Example: custquery.4gl (function cleanup)
  - Error if Cursor is Not Open:

This program implements query-by-example, using the CONSTRUCT statement to allow the user to enter search criteria in a form. The criteria is used to build an SQL SELECT statement which will retrieve rows from the customer database table. A SCROLL CURSOR is defined in the program, to allow the user to scroll back and forth between the rows of the result set. The SQLCA.SQLCODE is used to test the success of the SQL statements. Handling errors, and allowing the user to cancel the query, is illustrated.

Display on Windows platforms

Implementing Query-by-Example

Query-by-Example allows users to enter a value or a range of values for one or several form fields. Then your program looks up the database rows that satisfy the requirements. The BDL statement that makes this possible is CONSTRUCT.

Steps:

1. Define fields linked to database columns in a form specification file.
2. Define a STRING variable in your program to hold the query criteria.
3. Open a window and display the form.
4. Activate the form with the interactive dialog statement CONSTRUCT, for entry of the query criteria. Control is turned over to the user to enter his criteria.
5. The user enters his criteria in the fields specified in the CONSTRUCT statement. The CONSTRUCT statement accepts logical operators in any of the fields to indicate ranges, comparisons, sets, and partial matches. Using the form in this program, for example, the user can enter a specific value, such as **IL** in the **state** field, to retrieve all the rows from the **customer** table where the **state** column = **IL**. Or he can enter relational tests, such as **> 103**, in the **Store #** field, to retrieve only those rows where the **store_num** column is greater than **103**.
6. After entering his criteria, the user selects OK, to instruct your program to continue with the query, or Cancel to terminate the dialog.
   In this program, the action views for accept (OK) and cancel are displayed as buttons on the screen.
7. If the user accepts the dialog, the CONSTRUCT statement creates a Boolean expression by generating a logical expression for each field with a value and then applying unions (and relations) to the field statements. This expression is stored in the character string that you specified in the CONSTRUCT statement.
8. You can then use the Boolean expression to create a STRING variable containing a complete SELECT statement. You must supply the WHERE keyword to convert the Boolean expression into a WHERE clause. Make sure that you supply the spaces required to separate the constructed Boolean expression from the other parts of the SELECT statement.
9. Execute the statement to retrieve the row(s) from the database table, after preparing it or declaring a cursor for SELECT statements that might retrieve more than one row.

Using CONSTRUCT and STRING variables

A basic CONSTRUCT statement has the following format:

```
CONSTRUCT <variable-name> ON <column-list> FROM <field-list
```

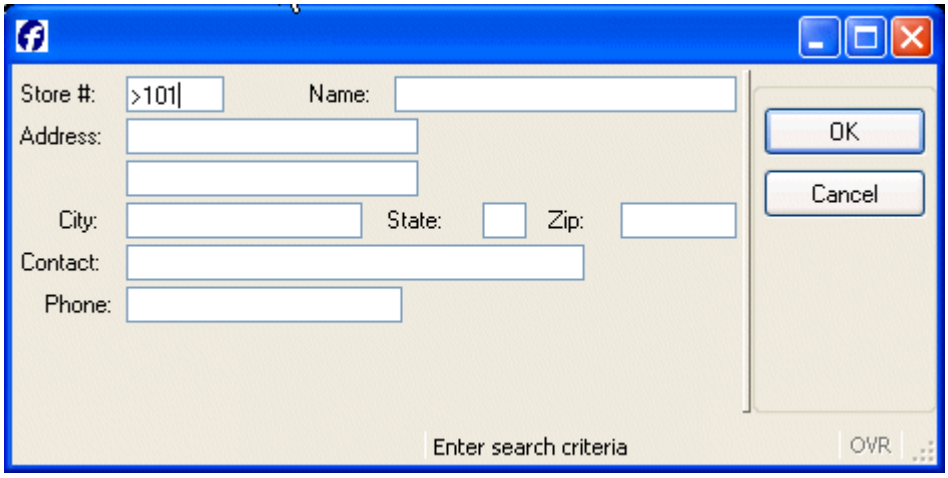
This statement temporarily binds the specified form fields to database columns. It allows you to identify database columns for which the user can enter search criteria. Each field and CONSTRUCT corresponding column must be the same or compatible data types. You can use the BY NAME clause when the fields on the screen form have the same names as the corresponding columns in the ON clause. The user can query only the screen fields implied in the BY NAME clause.

```
CONSTRUCT BY NAME <variable-name> ON <column-list>
```

The runtime system converts the entered criteria into a Boolean SQL condition that can appear in the WHERE clause of a SELECT statement. The variable to hold the query condition can be defined as a STRING data type. Strings are a variable length, dynamically allocated character string data type, without a size limitation. The STRING

variable can be concatenated, using the double pipe operator (||), with the text required to form a complete SQL SELECT statement. The LET statement can be used to assign a value to the variable. For example:

```
DEFINE where_clause, sqltext STRING
CONSTRUCT BY NAME where_clause ON customer.*
LET sql_text = "SELECT COUNT(*) FROM customer WHERE " ||
where_clause
```



Display on Windows Platform

In this example the user has entered the criteria **> 101** in the **store_num** field. The **where_clause** would be generated as

```
"store_num > 101"
```

and the complete **sql_text** would be

```
"SELECT COUNT(*) FROM customer WHERE store_num > 101"
```

Preparing the SQL Statement

The STRING created in the example is not valid for execution. The PREPARE instruction sends the text of the string to the database server for parsing, validation, and to generate the execution plan. The scope of a prepared SQL statement is the module in which it is declared.

```
PREPARE cust_cnt_stmt FROM sql_text
```

A prepared SQL statement can be executed with the EXECUTE instruction.

```
EXECUTE cust_cnt_stmt INTO cust_cnt
```

Since the SQL statement will only return one row (containing the count) the **INTO** syntax of the EXECUTE instruction can be used to store the count in the local variable

cust_cnt. (The function **cust_select** illustrates the use of database cursors with SQL SELECT statements.)

When a prepared statement is no longer needed, the FREE instruction will release the resources associated with the statement.

```
FREE cust_cnt_stmt
```

Allowing the User to Cancel the Query Operation

Predefined Actions (accept/cancel)

The language pre-defines some actions and associated names for common operations, such as **accept** or **cancel**, used during interactive dialogs with the user such as CONSTRUCT. You do not have to define these actions in the interactive instruction block, the runtime system interprets predefined actions. For example, when the **accept** action is caught, the dialog is validated.

You can define action views (such as buttons, toolbar icons, menu items) in your form using these pre-defined names; the corresponding action will automatically be attached to the view. If you do not define any action views for the actions, default buttons (such as OK/Cancel) for these actions will be displayed on the form as appropriate when interactive dialog statements are executed.

When the CONSTRUCT statement executes, buttons representing **accept** and **cancel** actions will be displayed by default, allowing the user to validate or cancel the interactive dialog statement. If the user selects Cancel, the INT_FLAG is automatically set to TRUE. Once INT_FLAG is set to TRUE, your program must re-set it to FALSE to detect a new cancellation. You typically set INT_FLAG to FALSE before you start a dialog instruction, and you test it just after (or in the AFTER CONSTRUCT / AFTER INPUT block) to detect if the dialog was canceled:

```
LET INT_FLAG = FALSE
CONSTRUCT BY NAME where_part
   ...
END CONSTRUCT
IF INT_FLAG = TRUE THEN
   ...
END IF
```

DEFER INTERRUPT and the INT_FLAG

The statement DEFER INTERRUPT in your MAIN program block will prevent your program from terminating abruptly if a SIGINT signal is received. When using a GUI interface, the user can generate an interrupt signal if you have an action view named 'interrupt' (the predefined **interrupt** action). If an interrupt event is received, TRUE is assigned to the built-in global integer variable INT_FLAG.

It is up to the programmer to manage the interruption event (stop or continue with the program), by testing the value of INT_FLAG variable.

Interruption handling is discussed in the report example, in chapter 9.

Conditional Logic

Once the CONSTRUCT statement is completed, you must test whether the INT_FLAG was set to TRUE (whether the user cancelled the dialog). Genero BDL provides the conditional logic statements IF or CASE to test a set of conditions.

The IF statement

```
IF <condition> THEN
  ....
ELSE
  ....
END IF
```

IF statements can be nested. The `ELSE` clause may be omitted.

If *condition* is TRUE, the runtime system executes the block of statements following `THEN`, until it reaches either the `ELSE` keyword or the `END IF` keywords. Your program resumes execution after `END IF`. If *condition* is FALSE, the runtime system executes the block of statements between `ELSE` and `END IF`.

```
IF (INT_FLAG = TRUE) THEN
      LET INT_FLAG = FALSE
      LET cont_ok = FALSE
    ELSE
      LET cont_ok = TRUE
    END IF
```

The CASE statement

The CASE statement specifies statement blocks to be executed conditionally, depending on the value of an expression. Unlike IF statements, `CASE` does not restrict the logical flow of control to only two branches. Particularly if you have a series of nested `IF` statements, the `CASE` statement may be more readable. In the previous example, the `CASE` statement could have been substituted for the `IF` statement:

```
CASE
WHEN (INT_FLAG = TRUE)
  LET INT_FLAG = FALSE
  LET cont_ok = FALSE
OTHERWISE
  LET cont_ok = TRUE
END CASE
```

Usually, there would be several conditions to check. The following statement uses an alternative syntax, since all the conditions check the value of **var1**:

```
CASE var1
WHEN 100
  CALL routine_100()
WHEN 200
  CALL routine_200()
OTHERWISE
  CALL error_routine()
END CASE
```

The first `WHEN` condition in the CASE statement will be evaluated. If the condition is true(**var1=100**), the statement block is executed and the `CASE` statement is exited. If the condition is not true, the next `WHEN` condition will be evaluated, and so on through subsequent `WHEN` statements until a condition is found to be true, or `OTHERWISE` or `END CASE` is encountered. The `OTHERWISE` clause of the `CASE` statement can be used as a catch-all for unanticipated cases.

See Flow Control for other examples of IF and CASE syntax and the additional conditional statement `WHILE`.

The Query program

The Query program consists of two modules. The **custmain.4gl** module must be linked with the **custquery.4gl** module in order for the program to be run. The line numbers shown in the code are for reference only, and are not a part of the code.

Example: Module custmain.4gl

This module contains the MAIN program block for the query program, and the MENU that drives the query actions.

Module custmain.4gl

```
01 MAIN
02
03   DEFER INTERRUPT
04
05   CONNECT TO "custdemo"
06   CLOSE WINDOW SCREEN
07   OPEN WINDOW w1 WITH FORM "custform"
08
09   MENU "Customer"
10     ON ACTION query
11       CALL query_cust()
12     ON ACTION next
```

```
13         CALL fetch_rel_cust(1)
14      ON ACTION previous
15         CALL fetch_rel_cust(-1)
16      ON ACTION exit
17        EXIT MENU
18    END MENU
19
20    CLOSE WINDOW w1
21
22    DISCONNECT CURRENT
23
24 END MAIN
```

Notes:

- Line 01 Beginning of the MAIN block. The SCHEMA statement is not needed since this module does not define any program variables in terms of a database table.
- Line 03 uses the DEFER INTERRUPT statement to prevent the user from terminating the program prematurely by pressing the INTERRUPT key.
- Line 07 opens a window with the same form that was used in the Chapter 3 example.
- Lines 09 thru 18 contains the MENU for the query program. Four actions - **query**, **next**, **previous**, and **quit** - will be displayed as buttons on the form. The pre-defined actions accept (OK button) and cancel will automatically be displayed as buttons when the CONSTRUCT statement is executed.
- Line 11 calls the function **query_cust** in the **cust_query.4gl** module.
- Line 13 calls the function **fetch_rel_cust** in the **cust.query.4gl** module. The literal value **1** is passed to the function, indicating that the cursor should move forward to the next row.
- Line 15 calls the function **fetch_rel_cust** also, but passes the literal value **-1**, indicating that the cursor should move backwards to retrieve the previous row in the results set.
- Line 17 exits the MENU statement.
- Line 20 closes the window that was opened.
- Line 22 disconnects from the database.

There are no further statements so the Query program terminates.

Example: Module custquery.4gl

This module of the Query program contains the logic for querying the database and displaying the data retrieved. The function **query_cust** is called by the "query" option of the MENU in **custmain.4gl**.

Module custquery.4gl (and function query_cust)

```
01 -- custquery.4gl
02
```

```
03 SCHEMA custdemo
04
05 DEFINE  mr_custrec RECORD
06  store_num    LIKE customer.store_num,
07  store_name   LIKE customer.store_name,
08  addr         LIKE customer.addr,
09  addr2        LIKE customer.addr2,
10  city         LIKE customer.city,
11  state        LIKE customer.state,
12  zipcode      LIKE customer.zipcode,
13  contact_name LIKE customer.contact_name,
14  phone          LIKE customer.phone
15 END RECORD
16
17 FUNCTION query_cust()
18   DEFINE cont_ok      SMALLINT,
19          cust_cnt     SMALLINT,
20          where_clause STRING
21   MESSAGE "Enter search criteria"
22   LET cont_ok = FALSE
23
24   LET INT_FLAG = FALSE
25   CONSTRUCT BY NAME where_clause
26      ON customer.store_num,
27         customer.store_name,
28         customer.city,
29         customer.state,
30         customer.zipcode,
31         customer.contact_name,
32         customer.phone
33
34   IF (INT_FLAG = TRUE) THEN
35     LET INT_FLAG = FALSE
36     CLEAR FORM
37     LET cont_ok = FALSE
38     MESSAGE "Canceled by user."
39   ELSE
40     CALL get_cust_cnt(where_clause)
41       RETURNING cust_cnt
42     IF (cust_cnt > 0) THEN
43       MESSAGE cust_cnt USING "<<<<",
44                   " rows found."
45        CALL cust_select(where_clause)
46          RETURNING cont_ok
47     ELSE
48       MESSAGE "No rows found."
49       LET cont_ok = FALSE
50     END IF
51   END IF
52
53   IF (cont_ok = TRUE) THEN
54     CALL display_cust()
55   END IF
56
57 END FUNCTION
```

Notes:

- Line 03 is required to identify the database schema file to be used when compiling the module.
- Lines 05 thru 15 define a RECORD, **mr_custrec**, that is modular in scope, since it is at the top of the module and outside any function. The values of this record will be available to, and can be set by, any function in this module.
- Line 17: **Function query_cust**. This is the beginning of the function **query_cust**.
- Line 18 defines **cont_ok**, a local variable of data type SMALLINT, to be used as a flag to indicate whether the query should be continued. The keywords TRUE and FALSE are used to set the value of the variable (0=FALSE, <>0=TRUE).
- Line 19 defines another local SMALLINT variable, **cust_cnt**, to hold the number of rows returned by the SELECT statement.
- Line 20 defines **where_clause** as a local STRING variable to hold the boolean condition resulting from the CONSTRUCT statement.
- Line 21 displays a message to the user that will remain until it is replaced by another MESSAGE statement.
- Line 22 sets cont_ok to FALSE, prior to executing the statements of the function.
- Line 24 sets INT_FLAG to FALSE. It is common to set this global flag to FALSE immediately prior to the execution of an interactive dialog, so your program can test whether the user attempted to cancel the dialog.
- Lines 25 thru 32: The CONSTRUCT statement lists the database columns for which the user may enter search criteria. The program does not permit the user to enter search criteria for the address columns. The BY NAME syntax matches the database columns to form fields having the same name.
- Line 34 is the beginning of an IF statement testing the value of INT_FLAG. This test appears immediately after the CONSTRUCT statement, to test whether the user terminated the CONSTRUCT statement (INT_FLAG would be set by the runtime system to TRUE).
- Lines 35 thru 38 are executed only if the value of INT_FLAG is TRUE. The INT_FLAG is immediately re-set to FALSE, since it is a global variable which other parts of your program will test. The form is cleared of any criteria that the user has entered, the **cont_ok** flag is set to FALSE, and a message is displayed to the user. The program will continue with the statements after the END IF on line 49.
- Lines 40 thru 50: contain the logic to be executed if INT_FLAG was not set to TRUE (the user did not cancel the query).
    - In lines 40 and 41, the **get_cust_cnt** function is called, to retrieve the number of rows that would be returned by the query criteria. The **where_clause** variable is passed to the function, and the value returned will be stored in the **cust_cnt** variable.
    - Lines 42 is the beginning of a nested IF statement, testing the value of **cust_cnt**.
    - Lines 43 thru 46 are executed if the value of **cust_cnt** is greater than zero; a message with the number of rows returned is displayed to the user, and the function **cust_select** is called. The **where_clause** is passed to this function, and the returned value is stored in **cont_ok**. Execution continues with the statement after the END IF on line 51.

    - Lines 48 and 49 are executed if the value is zero (no rows found); a message is displayed to the user, and **cont_ok** is set to FALSE. Execution continues after the END IF on line 51.
- Line 49 is the end of the IF statement beginning on line 33.
- Lines 53 thru 55 test the value of **cont_ok**, which will have been set during the preceding IF statements and in the function **cust_select**. If **cont_ok** is TRUE, the function **display_cust** is called.
- Line 57 is the end of the **query_cust** function.

Example: custquery.4gl (Function get_cust_cnt)

This function is called by the function **query_cust** to return the count of rows that would be retrieved by the SELECT statement. The criteria previously entered by the user and stored in the variable **where_clause** is used.

Function get_cust_cnt

```
01 FUNCTION get_cust_cnt(p_where_clause)
02   DEFINE p_where_clause STRING,
03          sql_text STRING,
04          cust_cnt SMALLINT
05
06   LET sql_text =
07    "SELECT COUNT(*) FROM customer" ||
08    " WHERE " || p_where_clause
09
10   PREPARE cust_cnt_stmt FROM sql_text
11   EXECUTE cust_cnt_stmt INTO cust_cnt
12   FREE cust_cnt_stmt
13
14   RETURN cust_cnt
15
16 END FUNCTION
```

Notes:

- Line 01 The function accepts as a parameter the value of **where_clause**, stored in the local variable **p_where_clause** defined on Line 60.
- Line 02 defines a local STRING variable, **sql_txt**, to hold the complete text of the SQL SELECT statement.
- Line 04 defines a local variable **cust_cnt** to hold the count returned by the SELECT statement.
- Lines 06 thru 08 create the string containing the complete SQL SELECT statement, concatenating **p_where_clause** at the end using the || operator. Notice that the word WHERE must be provided in the string.
- Line 10 uses the PREPARE statement to convert the STRING into an executable SQL statement, parsing the statement and storing it in memory. The prepared statement is modular in scope. The prepared statement has the identifier **cust_cnt_stmt**, which does not have to be defined.

- Line 11 executes the SQL SELECT statement contained in **cust_cnt_stmt**, using the EXECUTE ... INTO syntax to store the value returned by the statement in the variable **cust_cnt**. This syntax can be used if the SQL statement returns a single row of values.
- Line 12 The FREE statement releases the memory associated with the PREPAREd statement, since this statement is no longer needed.
- Line 14 returns the value of **cust_cnt** to the calling function, **query_cust**.
- Line 16 is the end of the **get_cust_cnt** function.

Retrieving data from the Database

Using Cursors

When an SQL SELECT statement in your application will retrieve more than one row, a cursor must be used to pass the selected data to the program one row at a time. The cursor is a data structure that represents a specific location within the active set of rows that the SELECT statement retrieved.

- Sequential cursor - reads through the active set only once each time it is opened, by moving the cursor forward one row each time a row is requested.
- Scroll cursor - fetches the rows of the active set in any sequence. To implement a scroll cursor, the database server creates a temporary table to hold the active set.

The scope of a cursor is the module in which it is declared. Cursor names must be unique within a module.

The general sequence of program statements when using a SELECT cursor for Query-by-Example is:

- DECLARE - the program declares a cursor for the STRING that contains the SQL SELECT statement. This allocates storage to hold the cursor. Note that the string does not have to be prepared using the PREPARE statement.
- OPEN - the program opens the cursor. The active set associated with the cursor is identified, and the cursor is positioned before the first row of the set.
- FETCH - the program fetches a row of data into host variables and processes it. The syntax `FETCH NEXT <cursor-identifier> INTO <variable-names>` can be used with a SCROLL CURSOR to fetch the next row relative to the current position of the cursor in the SQL result set. Using `FETCH PREVIOUS ...` moves the cursor back one row in the SQL result set.
- CLOSE - the program closes the cursor after the last row desired is fetched. This releases the active result set associated with the cursor. The cursor can be re-opened.
- FREE - when the cursor is no longer needed, the program frees the cursor to release the storage area holding the cursor. Once a cursor has been freed, it must be declared again before it can be re-opened.

The cursor program statements must appear physically within the module in the order listed.

The SQLCA.SQLCODE

The "SQLCA" name stands for "SQL Communication Area". The SQLCA variable is a predefined record containing information on the execution of an SQL statement. The SQLCA record is filled after any SQL statement execution. The SQLCODE member of this record contains the SQL execution code:

Execution Code	Description
0	SQL statement executed successfully.
100	No rows were found.
<0	An SQL error occurred.

The NOTFOUND constant is a predefined integer value that evaluates to 100. This constant is typically used to test the execution status of an SQL statement returning a result set, to check if rows have been found.

Example custquery.4gl (function cust_select)

This function is called by the function **query_cust**, if the row count returned by the function **get_cust_cnt** indicates that the criteria previously entered by the user and stored in the variable **where_clause** would produce an SQL SELECT result set.

Function cust_select

```
01 FUNCTION cust_select(p_where_clause)
02   DEFINE p_where_clause STRING,
03          sql_text STRING,
04          fetch_ok SMALLINT
05
06   LET sql_text = "SELECT store_num, " ||
07    " store_name, addr, addr2, city, " ||
08    " state, zipcode, contact_name, phone " ||
09    " FROM customer WHERE " || p_where_clause ||
10    " ORDER BY store_num"
11
12   DECLARE cust_curs SCROLL CURSOR FROM sql_text
13   OPEN cust_curs
14   CALL fetch_cust(1)   -- fetch the first row
15      RETURNING fetch_ok
16   IF NOT (fetch_ok) THEN
17      MESSAGE "no rows in table."
```

```
18   END IF
19
20   RETURN fetch_ok
21
22 END FUNCTION
```

Notes:

- Line 01 The function **cust_select** accepts as a parameter the **where_clause**, storing it in the local variable **p_where_clause**.
- Lines 06 thru 10 concatenate the entire text of the SQL statement into the local STRING variable **sql_txt**.
- Line 12 declares a SCROLL CURSOR with the identifier **cust_curs**, for the STRING variable **sql_text**.
- Line 13 opens the cursor, positioning before the first row of the result set. Note that these statements are physically in the correct order within the module.
- Lines 14 and 15 call the function **fetch_cust**, passing as a parameter the literal value **1**, and returning a value stored in the local variable **fetch_ok**. Passing the value **1** to **fetch_cust** will result in the NEXT row of the result set being fetched (see the logic in the function **fetch_cust**), which is this case would be the first row.
- Line 16 Since **fetch_ok** is defined as a SMALLINT, it can be used as a flag containing the values `TRUE` or `FALSE`. The value returned from the function **fetch_cust** indicates whether the fetch was successful.
- Line 17 displays a message to the user if the FETCH was not successful. Since this is the fetch of the first row in the result set, another user must have deleted the rows after the program selected the count.
- Line 20 returns the value of **fetch_ok** to the calling function. This determines whether the function **display_cust** is called.
- Line 22 is the end of the function **cust_select**.

Tips:

1. Lines 15 and 16 could be combined to shorten the code:

   ```
   IF NOT fetch_cust(1) THEN ...
   ```

   This syntax would call the function **fetch_cust** implicitly, passing the parameter **1**; the function returns TRUE or FALSE, which would be tested by the IF statement.

Example: custquery.4gl (function fetch_cust)

This function is designed so that it can be re-used each time a row is to be fetched from the **customer** database table; a variable is passed to indicate whether the cursor should move forward one row or backward one row.

Function fetch_cust

```
01 FUNCTION fetch_cust(p_fetch_flag)
02    DEFINE p_fetch_flag SMALLINT,
03           fetch_ok SMALLINT
04
05   LET fetch_ok = FALSE
06   IF (p_fetch_flag = 1) THEN
07     FETCH NEXT cust_curs
08       INTO mr_custrec.*
09   ELSE
10     FETCH PREVIOUS cust_curs
11       INTO mr_custrec.*
12   END IF
13
14   IF (SQLCA.SQLCODE = NOTFOUND) THEN
15     LET fetch_ok = FALSE
16   ELSE
17     LET fetch_ok = TRUE
18   END IF
19
20   RETURN fetch_ok
21
22 END FUNCTION
```

Notes:

- Line 01 The function **fetch_cust** accepts a parameter and stores it in the local variable **p_fetch_flag**.
- Line 03 defines a variable, **fetch_ok,** to serve as an indicator whether the FETCH was successful.
- Lines 06 thru 12 tests the value of **p_fetch_flag**, moving the cursor forward with `FETCH NEXT` if the value is **1**, and backward with `FETCH PREVIOUS` if the value is **-1**. The values of the row in the **customer** database table are fetched into the program variables of the **mr_custrec** record. The `INTO mr_custrec.*` syntax requires that the program variables in the record **mr_custrec** are in the same order as the columns are listed in the SELECT statement.
- Lines 14 thru 15 tests **SQLCA.SQLCODE** and sets the value of **fetch_ok** to `FALSE` if the fetch did not return a row. If the FETCH was successful, **fetch_ok** is set to `TRUE`.
- Line 20 returns the value of **fetch_ok** to the calling function.
- Line 22 is the end of the function **fetch_cust.**

Example: querycust.4gl (function fetch_rel_cust)

This function is called by the MENU options "next" and "previous" in the **custmain.4gl** module.

Function fetch_rel_cust

```
01 FUNCTION fetch_rel_cust(p_fetch_flag)
```

```
02   DEFINE p_fetch_flag SMALLINT,
03          fetch_ok SMALLINT
04
05   MESSAGE " "
06   CALL fetch_cust(p_fetch_flag)
07     RETURNING fetch_ok
08
09   IF (fetch_ok) THEN
10     CALL display_cust()
11   ELSE
12     IF (p_fetch_flag = 1) THEN
13       MESSAGE "End of list"
14     ELSE
15       MESSAGE "Beginning of list"
16     END IF
17   END IF
18
19 END FUNCTION
```

Notes:

- Line 01 The parameter passed to it, **p_fetch_flag** will be **1** or **-1**, depending on the direction in which the SCROLL CURSOR is to move.
- Line 05 re-sets the MESSAGE display to blanks.
- Line 06 calls the function **fetch_cust**, passing it the value of **p_fetch_flag**. The function **fetch_cust** uses the SCROLL CURSOR to retrieve the next row in the direction indicated, returning `FALSE` if there was no row found.
- Lines 09 and 10 If a row was found (the **fetch_cust** function returned `TRUE`) the **display_cust** function is called to display the row in the form.
- Line 13 If no rows were found and the direction is forward, indicated by **p_fetch_flag** of **1**, the cursor is past the end of the result set.
- Line 15 If no rows were found and the direction is backward, indicated by **p_fetch_flag** of **-1**, the cursor is prior to the beginning of the result set.
- Line 19 is the end of the function **fetch_rel_cust**.

Example: custquery.4gl (function display_cust)

This function displays the contents of the **mr_custrec** record in the form. It is called by the functions **query_cust** and **fetch_rel_cust**.

Function display_cust

```
01 FUNCTION display_cust()
02   DISPLAY BY NAME mr_custrec.*
03 END FUNCTION
```

Notes:

- Line 02 uses the DISPLAY BY NAME syntax to display the contents of the program record **mr_custrec** to the form fields having the same name.

Compiling and Linking the Program

The two example modules must be compiled and then linked into a single program.

From the command line:

```
fglcomp custmain.4gl
fglcomp custquery.4gl
```

This produces the object modules **custmain.42m** and **custquery.42m**, which must be linked to produce the program **cust.42r**:

```
fgllink -o cust.42r custmain.42m custquery.42m
```

Or, compile both modules and link at the same time:

```
fgl2p -o cust.42r custmain.4gl custquery.4gl
```

Modifying the Program to Handle Errors

The WHENEVER ERROR statement

Since program statements that access the database may be expected to fail occasionally (the row is locked, etc.) the WHENEVER ERROR statement can be used to handle this type of error.

By default, when a runtime error occurs the program will stop. To prevent this happening when SQL statements that access the database fail, surround the SQL statement with WHENEVER ERROR statements, as in the following example based on the **fetch_cust** function in the **custquery.4gl** program module:

```
01 IF (p_fetch_flag = 1) THEN
02    WHENEVER ERROR CONTINUE
03    FETCH NEXT cust_curs
04       INTO mr_custrec.*
05    WHENEVER ERROR STOP
06 ...
```

WHENEVER ERROR statements are modular in scope, and generate additional code for exception handling when the module is compiled. This exception handling is valid until the end of the module or until a new WHENEVER ERROR instruction is encountered by the compiler.

When the example code is compiled, WHENEVER ERROR CONTINUE will generate code to prevent the program from stopping if the FETCH statement fails. Immediately after the FETCH statement, the WHENEVER ERROR STOP instruction will generate the code to re-set the default behavior for the rest of the module.

You can write your own error function to handle SQL errors, and use the WHENEVER ERROR CALL <*function-name*> syntax to activate it. Run-time errors may be logged to an error log.

Negative SQLCA.SQLCODE

The database server returns an execution code whenever an SQL statement is executed, available in SQLCA.SQLCODE. If the code is a negative number, an SQL error has occurred. Just as we checked the SQLCA.SQLCODE for the NOTFOUND condition, we can also check the code for database errors (negative SQLCODE). The SQLCA.SQLCODE should be checked immediately after each SQL statement that may fail, including DECLARE, OPEN, FETCH, etc. For simplicity of the examples, the error handling in these programs is minimal.

SQLERRMESSAGE

If an SQL error occurs, the SQLERRMESSAGE operator returns the error message associated with the error code. This is a character string that can be displayed to the user with the ERROR instruction.

```
    ERROR SQLERRMESSAGE
```

Changes to function fetch_cust (custquery.4gl)

```
01 FUNCTION fetch_cust (p_fetch_flag)
02    DEFINE p_fetch_flag SMALLINT,
03           fetch_ok     SMALLINT
04
05   LET fetch_ok = FALSE
06   IF (p_fetch_flag = 1) THEN
07      WHENEVER ERROR CONTINUE
08     FETCH NEXT cust_curs
09       INTO mr_custrec.*
10     WHENEVER ERROR STOP
11   ELSE
12      WHENEVER ERROR CONTINUE
13     FETCH PREVIOUS cust_curs
14       INTO mr_custrec.*
15     WHENEVER ERROR STOP
16   END IF
17
18   CASE
19   WHEN (SQLCA.SQLCODE = 0)
20      LET fetch_ok = TRUE
21   WHEN (SQLCA.SQLCODE = NOTFOUND)
22      LET fetch_ok = FALSE
23   WHEN (SQLCA.SQLCODE < 0)
24      LET fetch_ok = FALSE
25     ERROR SQLERRMESSAGE
```

```
26   END CASE
27
28   RETURN fetch_ok
29
30 END FUNCTION
```

Notes:

- Lines 08, 09, 13, 14 The SQL statements are surrounded by WHENEVER ERROR statements. If an error occurs during the SQL statements, the program will continue. The error handling is re-set to the default (STOP) immediately after each SQL statement so that failures of other program statements will not be ignored.
- Lines 18 to 26 Immediately after the WHENEVER ERROR STOP statement, the SQLCA.SQLCODE is checked, to see whether the SQL statement succeeded. A CASE statement is used, since there are more than two conditions to be checked.

Close and Free the Cursor

Closing and freeing the cursor when you no longer need it is good practice, especially if the modules are part of a larger program. This function must be placed in the same module as the DECLARE/OPEN/FETCH statements and in sequence, so this is the last function in the **query_cust** module. However, the function can be called from **cust_main**, as a final "cleanup" routine.

Function cleanup (custquery.4gl)

```
01 FUNCTION cleanup()
02   WHENEVER ERROR CONTINUE
03   CLOSE cust_curs
04   FREE cust_curs
05   WHENEVER ERROR STOP
06 END FUNCTION
```

Notes:

- Line 03 Closes the cursor used to retrieve the database rows.
- Line 04 Frees the memory associated with the cursor.
- Lines 02 and 05 The WHENEVER ERROR statements prevent a program error if the user exited the program without querying, and the cursor was never created).

Error if Cursor is not Open

In the example program in this chapter, if the user selects the Next or Previous action from the MENU before he has queried, the program returns an error ("Program stopped at line .... Fetch attempted on unopened cursor."). One way to prevent this error would

be to add a variable to the program to indicate whether the user has queried for a result set, and to prevent him from executing the actions associated with Next or Previous until he has done so.

Changes to function query_cust (custquery.4gl)

```
01 FUNCTION query_cust()
02   DEFINE cont_ok      SMALLINT,
03          cust_cnt     SMALLINT,
04          where_clause STRING
05   MESSAGE "Enter search criteria"
06   LET cont_ok = FALSE
07
...
08
09   IF (cont_ok = TRUE) THEN
10     CALL display_cust()
11   END IF
12
13   RETURN cont_ok
14
15 END FUNCTION
```

Notes:

- Line 13 A single line is added to the **query_cust** function to return the value of **cont_ok,** which indicates whether the query was successful**,** to the calling function in **custmain.4gl**.

Changes to module custmain.4gl

```
01 MAIN
02   DEFINE query_ok SMALLINT
03
04   DEFER INTERRUPT
05
06   CONNECT TO "custdemo"
07   CLOSE WINDOW SCREEN
08   OPEN WINDOW w1 WITH FORM "custform"
09   LET query_ok = FALSE
10
11   MENU "Customer"
12     ON ACTION query
13       CALL query_cust() RETURNING query_ok
14     ON ACTION next
15       IF (query_ok) THEN
16         CALL fetch_rel_cust(1)
17       ELSE
18         MESSAGE "You must query first."
19       END IF
20     ON ACTION previous
21       IF (query_ok) THEN
22         CALL fetch_rel_cust(-1)
23       ELSE
24         MESSAGE "You must query first."
```

```
25       END IF
26     ON ACTION quit
27       EXIT MENU
28   END MENU
29
30   CLOSE WINDOW w1
31   CALL cleanup()
32   DISCONNECT CURRENT
33
34 END MAIN
```

Notes:

- Line 03 defines the variable **query_ok**, which will be used to indicate whether the user has queried.
- Line 09 sets the initial value of **query_ok** to FALSE.
- Line 13 the function **query_cust** now returns a value for **query_ok**.
- Lines 15 thru 19 and Lines 21 thru 25: these sections test the value of **query_ok** when Next or Previous has been selected. If **query_ok** is TRUE, the function **fetch_rel_cust** is called; otherwise, a message is displayed to the user.
- Line 31 calls the **cleanup** function to close the cursor used to fetch the database rows.

Tutorial Chapter 5: Enhancing the Form

Summary:

- Adding a Toolbar
- Adding a Topmenu
- Adding a ComboBox form item
- Changing the Window Appearance
- Examples
- Managing actions
  - Disable/enable actions
  - The close action
- Example: custmain.4gl
- Action Defaults
- MENU/Action Defaults Interaction

You can change the way that program options are displayed in a form in a variety of ways. This example program illustrates some of the simple changes that can be made:

- By changing the form specification file, you can provide the user with a valid list of abbreviations for the state field and add a Toolbar or pulldown menu (Topmenu). The program business logic in the BDL program need not change. Once you recompile the form file, it can be used by the program with no additional changes required.
- You can change the appearance of the application window, adding a custom title and icon.
- You can disable and enable actions dynamically to control the options available to the user.

The program also illustrates some of the Genero BDL action defaults that standardize the presentation of common actions.

Adding a Toolbar

Display on Windows platforms

The TOOLBAR section of a form specification file defines a Toolbar with buttons that are bound to actions. A Toolbar definition can contain the following elements:

- an ITEM - specifies the action that is bound to the Toolbar button
- a SEPARATOR - a vertical line

Values can be assigned to TEXT, COMMENT, and IMAGE attributes for each item in the Toolbar.

The TOOLBAR commands are enabled by actionsdefined by the current interactive BDL instruction, which in our example is the MENU statement in the **custquery.4gl** module. When a Toolbar button is selected by the user, the program triggers the actionto which the Toolbar button is bound.

Example: (in custform.per)

This TOOLBAR will display buttons for **find**, **next, previous**, and **quit** actions.

Form (custform.per)

```
01 SCHEMA custdemo
02
03 TOOLBAR
04   ITEM find
05   ITEM previous
06   ITEM next
07   SEPARATOR
08   ITEM quit (TEXT="Quit", COMMENT="Exit the program", IMAGE="exit")
09 END
10
...
```

Notes:

- Line 04 The ITEM command-identifier **find** will be bound to the MENU statement action **find** on line 14 in the **custmain.4gl** file shown below. The word **find** must be identical in both the TOOLBAR ITEM and the MENU statement action, and must always be in lower-case. The other command-identifiers are similarly bound.
- Line 08 Although attributes such as TEXT or COMMENT are defined for the ITEM **quit**, the ITEMS **find**, **previous**, and **next** do not have any attributes defined in the form specification file. These actions are common actions that have default attributes defined in the action defaults file.

Adding a Topmenu

The same options that were displayed to the user as a TOOLBAR can also be defined as buttons on a pull-down menu (a TOPMENU). To change the presentation of the menu options to the user, simply modify and recompile the form specification file.

Display on Windows platforms

The TOPMENU section of the form specification allows you to design the pull-down menu. The TOPMENU section must appear after SCHEMA, and must contain a tree of GROUP elements that define the pull-down menu. The GROUP TEXT value is the title for the pull-down menu group.

A GROUP can contain the following elements:

- a COMMAND - specifies the action the menu option must be bound to
- a SEPARATOR - a horizontal line
- GROUP children - a subgroup within a group.

Values can be assigned to attributes such as TEXT, COMMENT, and IMAGE. for each item in the TOPMENU.

As in a Toolbar, the TOPMENU commands are enabled by actions defined by the current interactive BDL instruction (dialog), which in our example is the MENU statement in the **custquery.4gl** module. When a TOPMENU option is selected by the user, the program triggers the action to which the TOPMENU command is bound.

Example (in custform.per):

Form custform.per

```
01 SCHEMA custdemo
02
03 TOPMENU
04   GROUP form (TEXT="Form")
05     COMMAND quit (TEXT="Quit", COMMENT="Exit the program",
IMAGE="exit")
06   END
07   GROUP stores (TEXT="Stores")
08     COMMAND find
09     SEPARATOR
13     COMMAND next
14     COMMAND previous
15  END
16 END
17
...
```

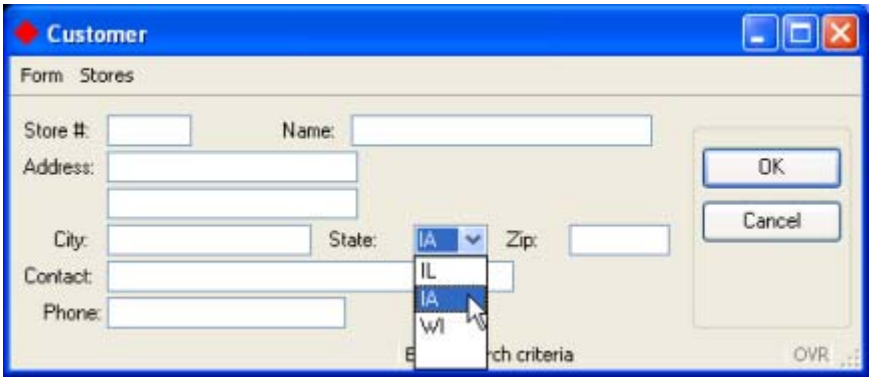
Notes:

- Lines 04 and 07 This example TOPMENU will consist of two groups on the menu bar of the form. The TEXT displayed on the menu bar for the first group will be **Form**, and the second group will be **Stores**.
- Line 08 to 14: Under the menu bar item **Stores**, the command-identifier **find** on line 05 will be bound to the MENU statement action **find** on line 14 in the **custmain.4gl** file shown below. The word **find** must be identical (including case) in both the TOPMENU command and the MENU statement action. The other command-identifiers are similarly bound.

The revised form specification file must be re-compiled before it can be used in the program.

Adding a COMBOBOX form item

In this example application the only valid values for the **state** column of the database table **customer** are IL, IA, and WI. The form item used to display the **state** field can be changed to a COMBOBOX displaying a dropdown list of valid state values. The COMBOBOX is active during an INPUT, INPUT ARRAY, or CONSTRUCT statement, allowing the user to select a value for the state field.



Display on Windows platforms

The values of the list are defined by the ITEMS attribute:

```
COMBOBOX f6=customer.state, ITEMS = ("IL", "IA", "WI");
```

In this example, the value displayed on the form and the real value (the value to be stored in the program variable corresponding to the form field) are the same. You can choose to define different display and real values; in the following example, the values Paris, Madrid, and London would be displayed to the user, but the value stored in the corresponding program variable would be 1, 2, or 3:

```
COMBOBOX f9 = formonly.cities, ITEMS =
((1,"Paris"),(2,"Madrid"),(3,"London"));
```

Although the list of values for the COMBOBOX is contained in the form specification file in this example program, you could also set the INITIALIZER attribute to define a function that will provide the values. The initialization function would be invoked at runtime when the form is loaded, to fill the COMBOBOX item list dynamically with database records, for example.

See form file item-types for a complete list of the item types that can be used on a form.

Changing the Window Appearance

Genero provides attributes that can be used to customize the appearance of windows, forms, and form objects in your application. In addition, you can create Presentation Styles to standardize the appearance of window and form objects across applications.

Some of the simple changes that you can make are:

Title

The default title for a window is the name of the object in the OPEN WINDOW statement. For example, in the programs we've seen so far, the title of the window is w1:

```
OPEN WINDOW w1 WITH FORM "custform"
```

In the form specification file, the attribute TEXT of the LAYOUT section can be used to change the title of the parent window:

```
LAYOUT (TEXT="Customer")
```

Icon

The Genero runtime system provides built-in classes, or object templates, which contain methods, or functions, that you can call from your programs. The classes are grouped together into packages. One package, **ui**, contains the "Interface" class, allowing you to manipulate the user interface. For example, the **setImage** method can be used to set the default icon for the windows of your program. You may simply call the method, prefixing it with the package name and class name; you do not need to create an Interface object.

```
CALL ui.Interface.setImage("imagename")
```

Window Style

By default windows are displayed as normal application windows, but you can choose a specific style using the WINDOWSTYLE attribute of the LAYOUT section of the form file. The default window styles are defined as a set of attributes in an external file (**default.4st)**.

```
LAYOUT (WINDOWSTYLE="dialog")
```

Example: (in custform.per)

Form custform.per

```
...
18 LAYOUT (TEXT="Customer")
19  GRID
20  {
21   Store #:[f01  ]       Name:[f02                ]
22   Address:[f03                ]
23           [f04                ]
24      City:[f05            ]State:[f6  ]Zip:[f07   ]
25   Contact:[f08                          ]
```

```
26     Phone:[f09                ]
27  }
28  END
29 END
30 TABLES
31   customer
32 END
33 ATTRIBUTES
34 EDIT f01=customer.store_num,
35   REQUIRED, COMMENT="This is the co-op store number";
36 EDIT f02=customer.store_name;
37 EDIT f03=customer.addr;
38 EDIT f04=customer.addr2;
39 EDIT f05=customer.city;
40 COMBOBOX f6=customer.state,
41   REQUIRED, ITEMS = ("IL", "IA", "WI");
41 EDIT f07=customer.zipcode;
42 EDIT f08=customer.contact_name;
43 EDIT f09=customer.phone;
43 END
```

Notes:

- Line 18, the title of the window is set to **Customer**. Since this is a normal application window, the default window style is used.
- Line 40, a COMBOBOX is substituted for a simple Edit form field.
- Line 35 and 41 The REQUIRED attribute forces the user to enter or select a value for this field when a new record is being added. See the attributes list for a complete list of the attributes that can be defined for a form field.

Example: (in custmain.4gl)

Changing the icon for the application windows:

Module custmain.4gl

```
...
04 MAIN
05   DEFINE query_ok SMALLINT
06
07   DEFER INTERRUPT
08
09   CONNECT TO "custdemo"
10   CLOSE WINDOW SCREEN
11   CALL ui.Interface.setImage("smiley")
12   OPEN WINDOW w1 WITH FORM "custform"
13
...
```

Notes:

- Line 11 For convenience, the image used is the **smiley** image from the **pics** directory, which is the default image directory of the Genero Desktop Client.

Managing Actions

Disable/Enable Actions

In the example in the previous lesson, if the user clicks the Next or Previous buttons on the application form without first querying successfully, a message displays and no action is taken. You can disable and enable the actions instead, providing visual cues to the user when the actions are not available. The ui.Dialog built-in class provides an interface to the BDL interactive dialog statements, such as CONSTRUCT and MENU. The method **setActionActive** enables and disables actions. To call a method of this class, use the pre-defined **DIALOG** object within the interactive instruction block.

For example:

```
MENU
   BEFORE MENU
    CALL DIALOG.setActionActive("actionname" , state)
  ...
END MENU
```

where *actionname* is the name of the action, *state* is an integer, **0** (disable) or **1** (enable).

You must be within an interactive instruction in order to use the DIALOG object in your program, but you can pass the object to a function. Using this technique, you could create a function that enables/disables an action, and call the function from the MENU statement, for example. See ui.Dialog for further information.

The Close Action

In Genero applications, when the user clicks the X button in the upper-right corner of the application window, a predefined **close** action is sent to the program. What happens next depends on the interactive dialog statement:

- When the program is in a MENU dialog statement, the **close** action is converted to an INTERRUPT key press. If there is a COMMAND KEY (INTERRUPT) block in the MENU statement, the statements in that control block are executed. Otherwise, no action is taken.
- When the program is in an INPUT, INPUT ARRAY, CONSTRUCT or DISPLAY ARRAY statement, the **close** action cancels the dialog, and the **int_flag** is set to TRUE. Your program can check the value of **int_flag** and take appropriate action.

You can change this default behavior by overwriting the **close** action within the interactive statement. For example, to exit the MENU statement when the user clicks this button:

```
MENU
```

```
    ...
    ON ACTION close
       EXIT MENU
END MENU
```

By default the action view for the **close** action is hidden and does not display on the form.

Example: (custmain.4gl)

Module custmain.4gl

```
01
02 MAIN
03 DEFINE query_ok SMALLINT
04
05 DEFER INTERRUPT
06 CONNECT TO "custdemo"
07 CLOSE WINDOW SCREEN
08 CALL ui.Interface.setImage("smiley")
09 OPEN WINDOW w1 WITH FORM "custform"
10
11 LET query_ok = FALSE
12
13 MENU
14   BEFORE MENU
15     CALL DIALOG.setActionActive("next",0)
16     CALL DIALOG.setActionActive("previous",0)
17   ON ACTION find
18     CALL DIALOG.setActionActive("next",0)
19     CALL DIALOG.setActionActive("previous",0)
20     CALL query_cust() RETURNING query_ok
21     IF (query_ok) THEN
22       CALL DIALOG.setActionActive("next",1)
23       CALL DIALOG.setActionActive("previous",1)
24     END IF
25   ON ACTION next
26      CALL fetch_rel_cust(1)
27   ON ACTION previous
28     CALL fetch_rel_cust(-1)
29   ON ACTION quit
30     EXIT MENU
31   ON ACTION close
32     EXIT MENU
33 END MENU
34
35 CLOSE WINDOW w1
36
37 DISCONNECT CURRENT
38
39 END MAIN
```

Notes:

- Line 08 The icon for the application windows is set to the "exit" image.
- Lines 15, 16 Before the menu is first displayed, the **next** and **previous** actions are disabled.
- Lines 18, 19 Before the **query_cust** function is executed the **next** and **previous** actions are disabled
- Lines 21 thru 24 If the query was successful the **next** and **previous** actions are enabled.
- Line 31 The **close** action is included in the menu, although an action view won't display on the form. If the user clicks the X button in the top right of the window, the action on line 32, EXIT MENU, will be taken.

Action Defaults

The Genero BDL runtime system includes an XML file, **default.4ad,** in the **lib** subdirectory of the installation directory FGLDIR, that defines presentation attributes for some commonly used actions. If you match the action names used in this file exactly when you define your action views (TOOLBAR or TOPMENU items, buttons, etc.) in the form specification file, the presentation attributes defined for this action will be used. All action names must be in lower-case.

For example, the following line in the **default.4ad** file:

```
<ActionDefault name="find" text="Find"
               image="find" comment="Search" />
```

defines presentation attributes for a **find** action- the text to be displayed on the action view **find** defined in the form, the image file to be used as the icon for the action view, and the comment to be associated with the action view. The attribute values are case-sensitive,so the action name in the form specification file must be "find", not "Find".

The following line in the **default.4ad** file defines presentation attributes for the pre-defined action **cancel**. An accelerator key is assigned as an alternate means of invoking the action:

```
<ActionDefault name="cancel" text="Cancel"
               acceleratorName="Escape" />
```

You can override a default presentation attribute in your program. For example, by specifying a TEXT attribute for the action **find** in the form specification file, the default TEXT value of "Find " will be replaced with the value "Looking".

```
03 TOPMENU
04
...
07   GROUP stores (TEXT="Stores")
08     COMMAND find (TEXT="Looking")
```

You can create your own **.4ad** file to standardize the presentation attributes for all the common actions used by your application. See Action Defaults for additional details.

MENU/Action Defaults Interaction

The attributes of the action views for the MENU actions in the **custmain.4gl** example will be determined as shown in the table below. Attributes defined in the form specification file override attributes defined in the **.4ad** file.

Action	From the form specification file	From the default.4ad file	From the MENU statement in the .4gl file
find	No attributes listed	TEXT="Find" IMAGE="find" COMMENT="Search"	Over-ridden by default.4ad
next	No attributes listed	TEXT="Next" IMAGE="goforw" COMMENT="Next record"	Over-ridden by default.4ad
previous	No attributes listed	TEXT="Previous" IMAGE="gorev" COMMENT="Previous record"	Over-ridden by default.4ad
close	Not listed in the form file	attributes are listed in default.4ad but the action view is not displayed on form by default	Over-ridden by default.4ad (pre-defined action)
quit	For both TOPMENU and TOOLBAR, the action view has the attributes TEXT="Quit", COMMENT="Exit the program", IMAGE="exit".	Action is not listed in the file	Over-ridden by the form specification file.
***accept**	Not listed in the form file.	TEXT="OK" AcceleratorName="Return" AcceleratorName2="Enter"	This action is not defined in a MENU instruction (pre-defined action.)
***cancel**	Not listed in the form file.	TEXT="Cancel" AcceleratorName="Escape"	This action is not defined in a MENU instruction (pre-defined action.)

* The pre-defined actions **accept** and **cancel** do not have action views defined in the form specification file; by default, they appear on this form as buttons in the righthand section of the form when the CONSTRUCT statement is active. Their attributes are taken from the **default.4ad** file.

Images

The image files specified in these definitions are among the files provided with the **Genero Desktop Client**, in the **pics** subdirectory.

Tutorial Chapter 6: Add/Update/Delete

Summary:

- Entering data on a form (INPUT statement)
  - INPUT attribute (UNBUFFERED)
  - INPUT attribute (WITHOUT DEFAULTS)
- Updating database tables
  - SQL transactions
  - Concurrency and Consistency
- Adding a new row
  - INPUT statement control blocks
  - Example: Add a row to the customer table
- Updating an existing row
  - Using a work record
  - SELECT ... FOR UPDATE
  - CURSOR WITH HOLD
  - Example: Update a row in the customer table
- Deleting a row
  - Using a modal Menu to prompt for validation
  - Example: Deleting a row in the customer table

This program allows the user to insert/update/delete rows in the customer table. Embedded SQL statements (UPDATE/INSERT/DELETE) are used to update the table, based on the values stored in the program record. SQL transactions, and concurrency and consistency issues are discussed. Prior to deleting a row, a dialog window is displayed to prompt the user to verify the deletion.

Entering data on a form: INPUT statement

The INPUT statement allows the user to enter or change the values in a program record, which can then be used as the data for new rows in a database table, or to update existing rows. In the INPUT statement you list:

- The program variables that are to receive data from the form
- The corresponding form fields that the user will use to supply the data

```
INPUT <program-variables> FROM <form-fields>
```

The FROM clause explicitly binds the fields in the screen record to the program variables, so the INPUT instruction can manipulate values that the user enters in the screen record. The number of record members must equal the number of fields listed in the FROM clause. Each variable must be of the same (or a compatible) data type as the corresponding screen field. When the user enters data, the runtime system checks the entered value against the data type of the variable, not the data type of the screen field.

When invoked, the INPUT statement enables the specified fields of the form in the current BDL window, and waits for the user to supply data for the fields. The user moves the cursor from field to field and types new values. Each time the cursor leaves a field, the value typed into that field is deposited into the corresponding program variable. You can write blocks of code as clauses in the INPUT statement that will be called automatically during input, so that you can monitor and control the actions of your user within this statement.

The INPUT statement ends when the user selects the **accept** or **cancel** actions.

INPUT supports the same shortcuts for naming records as the DISPLAY statement. You can ask for input to all members of a record, from all fields of a screen record, and you can ask for input BY NAME from fields that have the same names as the program variables.

```
INPUT BY NAME <programrecord>.*
```

UNBUFFERED attribute

By default, field values are buffered. The UNBUFFERED attribute makes the INPUT dialog "sensitive", allowing you to easily change some form field values programmatically during INPUT execution. When you assign a value to a program variable, the runtime system will automatically display that value in the form; when you input values in a form field, the runtime system will automatically store that value in the corresponding program variable. Using the UNBUFFERED attribute is **strongly** recommended.

WITHOUT DEFAULTS attribute

The same INPUT statement can be used, with the WITHOUT DEFAULTS attribute, to allow the user to make changes to an existing program record representing a row in the database. This attribute prevents BDL from automatically displaying any default values that have been defined for the form fields when INPUT is invoked, allowing you to display the existing database values on the screen before the user begins editing the data. In this case, when the INPUT statement is used to allow the user to add a new row, any existing values in the program record must first be nulled out.

Updating Database Tables

The values of the program variables that have been input through the form can be used in SQL statements that update tables in a database.

SQL transactions

The embedded SQL statements INSERT, UPDATE, and DELETE can be used to make changes to the contents of a database table. If your database has transaction logging,

you can use the BEGIN WORK and COMMIT WORK commands to delimit a transaction block, usually consisting of multiple SQL statements. If you do not issue a BEGIN WORK statement to start a transaction, each statement executes within its own transaction. These single-statement transactions do not require either a BEGIN WORK statement or a COMMIT WORK statement. At runtime, the Genero database driver generates the appropriate SQL commands to be used with the target database server.

To eliminate concurrency problems, keep transactions as short as possible.

Concurrency and Consistency

While your program is modifying data, another program may also be reading or modifying the same data. To prevent errors, database servers use a system of locks. When another program requests the data, the database server either makes the program wait or turns it back with an error. BDL provides a combination of statements to control the effect that locks have on your data access:

- ```
  SET LOCK MODE TO {WAIT [n]| NOT WAIT }
  ```

  This defines the timeout for lock acquisition for the current connection. The timeout period can be specified in seconds (**n**). If no period is specified, the timeout is infinite. If the LOCK MODE is set to NOT WAIT, an exception is returned immediately if a lock cannot be acquired.

  Warning: This feature is not supported by all databases. When possible, the database driver sets the corresponding connection parameter to define the timeout. If the database server does not support setting the lock timeout parameter, the runtime system generates an exception.

- ```
  SET ISOLATION LEVEL TO { DIRTY READ
                         | COMMITTED READ
                         | CURSOR STABILITY
                         | REPEATABLE READ }
  ```

  This defines the ISOLATION LEVEL for the current connection. When possible, the database driver executes the native SQL statement that corresponds to the specified isolation level.

For portable database programming, the following is recommended:

- Transactions must be enabled in your database.
- The ISOLATION LEVEL must be at least COMMITTED READ. On most database servers, this is usually the default isolation level and need not be changed.

- The LOCK MODE must be set to WAIT or WAIT <*timeperiod*>**,** if this is supported by your database server.

See Transactions in the BDL Reference Manual for a more complete discussion. The ODI Adaptation Guides provide detailed information about the behavior of specific database servers.

Adding a new row

INPUT Statement Control blocks

Genero BDL provides some optional control blocks for the INPUT statement that are called automatically as the user moves the cursor through the fields of a form. This allows your program to initialize field contents when adding a new row, for example, or to validate the user's input.

For example:

- BEFORE FIELD control blocks are executed immediately prior to the focus moving to the specified field. The example program uses this control block to prevent the user from changing the store number during an Update, by immediately moving the focus to the store name field (the NEXT FIELD instruction).
- An ON CHANGE is used to verify the uniqueness of the store number that was entered, and to make sure that the store name is not left blank. The user receives notification of a problem with the value of a field as soon as the field is exited. Validating these values as they are completed is less disruptive than notifying the user of several problems after the entire record has been entered.

See the INPUT statement for a complete list of control blocks.

Example: add a new row to the customer table

Module custmain.4gl

The MENU statement in the module **custmain.4gl** is modified to call functions for adding, updating, and deleting the rows in the customer table.

The MAIN block (custmain.4gl)

```
01 -- custmain.4gl
02
03 MAIN
04   DEFINE query_ok INTEGER
05
```

```
06   DEFER INTERRUPT
07   CONNECT TO "custdemo"
08   SET LOCK MODE TO WAIT 6
09   CLOSE WINDOW SCREEN
10   OPEN WINDOW w1 WITH FORM "custform"
11
12   MENU
13    ON ACTION find
14      LET query_ok = query_cust()
15    ON ACTION next
16      IF (query_ok) THEN
17        CALL fetch_rel_cust(1)
18      ELSE
19        MESSAGE "You must query first."
20      END IF
21    ON ACTION previous
22      IF (query_ok) THEN
23        CALL fetch_rel_cust(-1)
24      ELSE
25        MESSAGE "You must query first."
26      END IF
27    COMMAND "Add"
28      IF (inpupd_cust("A")) THEN
29        CALL insert_cust()
30      END IF
31    COMMAND "Delete"
32      IF (delete_check()) THEN
33        CALL delete_cust()
34     END IF
35    COMMAND "Modify"
36      IF inpupd_cust("U") THEN
37        CALL update_cust()
38     END IF
39    ON ACTION quit
40      EXIT MENU
41  END MENU
42
43  CLOSE WINDOW w1
44
45  DISCONNECT CURRENT
46
47 END MAIN
```

Notes:

- Line `08` sets the lock timeout period to 6 seconds.
- Lines `12` thru `41` define the main menu of the program.
- Lines `27` thru `30` The MENU option "Add" now calls an **inpupd_cust** function. Since this same function will also be used for updates, the value "A", indicating an Add of a new row, is passed. If **inpupd_cust** returns `TRUE`, the **insert_cust** function is called.
- Lines `31` thru `34` The MENU option "Delete" now calls a **delete_check** function. If **delete_check** returns `TRUE`, the **delete_cust** function is called.

- Lines 35 thru 38 are added to the MENU statement for the "Modify" option, calling the **inpud_cust** function. The value "U", for an Update of a new row, is passed as a parameter. If **inpupd_cust** returns TRUE, the **update_cust** function is called.

Module custquery.4gl (function inpupd_cust)

A new function, **inpupd_cust,** is added to the **custquery.4gl** module, allowing the user to insert values for a new customer row into the form.

Function inpupd_cust (custquery.4gl)

```
01 FUNCTION inpupd_cust(au_flag)
02   DEFINE au_flag CHAR(1),
03          cont_ok SMALLINT
04
05   LET cont_ok = TRUE
07
08   IF (au_flag = "A") THEN
09     MESSAGE "Add a new customer"
10     INITIALIZE mr_custrec.* TO NULL
12   END IF
13
14   LET INT_FLAG = FALSE
15
16   INPUT BY NAME mr_custrec.*
17         WITHOUT DEFAULTS ATTRIBUTES(UNBUFFERED)
18
19    ON CHANGE store_num
20     IF (au_flag = "A") THEN
21      SELECT store_name,
22             addr,
23             addr2,
24             city,
25             state,
26             zipcode,
27             contact_name,
28             phone
29        INTO mr_custrec.*
30       FROM customer
31       WHERE store_num = mr_custrec.store_num
32      IF (SQLCA.SQLCODE = 0)THEN
33       ERROR "Store number already exists."
34         LET cont_ok = FALSE
35         CALL display_cust()
36        EXIT INPUT
37      END IF
38     END IF
39
40   AFTER FIELD store_name
41     IF (mr_custrec.store_name IS NULL) THEN
42       ERROR "You must enter a company name."
43       NEXT FIELD store_name
44     END IF
45
```

```
46  END INPUT
47
48  IF (INT_FLAG) THEN
49    LET INT_FLAG = FALSE
50    LET cont_ok = FALSE
51    MESSAGE "Operation cancelled by user"
52    INITIALIZE mr_custrec.* TO NULL
53  END IF
54
55  RETURN cont_ok
56
57 END FUNCTION
```

Notes:

- Line 01 The function accepts a parameter defined as CHAR(1). In order to use the same function for both the input of a new record and the update of an existing one, the CALL to this function in the MENU statement in **main.4gl** will pass a value "**A**" for add, and "**U**" for update.
- Line 06 The variable **cont_ok** is a flag to indicate whether the update operation should continue; set initially to TRUE.
- Lines 08 thru 12 test the value of the parameter **au_flag**. If the value of **au_flag** is "**A**" the operation is an Add of a new record, and a MESSAGE is displayed. Since this is an Add, the modular program record values are initialized to NULL prior to calling the INPUT statement, so the user will have empty form fields in which to enter data.
- Line 14 sets the INT_FLAG global variable to FALSE prior to the INPUT statement, so the program can determine if the user cancels the dialog.
- Line 17 The UNBUFFERED and WITHOUT DEFAULTS clauses of the INPUT statement are used. The WITHOUT DEFAULTS clause is required since this statement will also be used for Updates, to prevent the existing values displayed on the form from being erased or replaced with default values.
- Lines 19 thru 38 Each time the value in **store_num** changes, the **customer** table is searched to see if that **store_num** already exists. If so, the values in the **mr_custrec** record are displayed in the form, the variable **cont_ok** is set to FALSE, and the INPUT statement is immediately terminated.
- Lines 40 thru 44 The AFTER FIELD control block verifies that **store_name** was not left blank. If so, the NEXT FIELD statement returns the focus to the **store_name** field so the user may enter a value.
- Line 46 END INPUT is required when any of the optional control blocks of the INPUT statement are used.
- Lines 48 thru 53 The INT_FLAG is checked to see if the user has cancelled the input. If so, the variable **cont_ok** is set to FALSE, and the program record **mr_custrec** is NULLED out. The UNBUFFERED attribute of the INPUT statement assures that the NULL values in the program record are automatically displayed on the form.
- Line 55 returns the value of **cont_ok**, indicating whether the input was successful.

Module custquery.4gl (function insert_cust)

A new function, **insert_cust**, in the **custquery.4gl** module, contains the logic to add the new row to the customer table.

Function insert_cust

```
01 FUNCTION insert_cust()
02
03  WHENEVER ERROR CONTINUE
04  INSERT INTO customer (
05     store_num,
06     store_name,
07     addr,
08     addr2,
09     city,
10     state,
11     zipcode,
12     contact_name,
13     phone
14     ) VALUES (mr_custrec.*)
15  WHENEVER ERROR STOP
16
17  IF (SQLCA.SQLCODE = 0) THEN
18     MESSAGE "Row added"
19  ELSE
20     ERROR SQLERRMESSAGE
21  END IF
22
23 END FUNCTION
```

Notes:

- Lines 04 thru 14 contain an embedded SQL statement to insert the values in the program record **mr_custrec** into the **customer** table. This syntax can be used when the order in which the members of the program record were defined matches the order of the columns listed in the SELECT statement. Otherwise, the individual members of the program record must be listed separately. Since there is no BEGIN WORK/COMMIT WORK syntax used here, this statement will be treated as a singleton transaction and the database driver will automatically send the appropriate COMMIT statement. The INSERT statement is surrounded by WHENEVER ERROR statements.
- Lines 17 thru 21 test the SQLCA.SQLCODE that was returned from the INSERT statement. If the INSERT was not successful, the corresponding error message is displayed to the user.

Updating an existing Row

Updating an existing row in a database table provides more opportunity for concurrency and consistency errors that inserting a new row. Using the following techniques can help to minimize these errors.

Using a work record

A work record and a local record, both identical to the program record, are defined to allow the program to compare the values.

1. A SCROLL CURSOR is used to allow the user to scroll through a result set generated by a query. The scroll cursor is declared WITH HOLD so it will not be closed when a COMMIT WORK or ROLLBACK WORK is executed.
2. When the user chooses Update, the values in the current program record are copied to the work record.
3. The INPUT statement accepts the user's input and stores it in the program record. The WITHOUT DEFAULTS keywords are used to insure that the original values retrieved from the database were not replaced with default values.
4. If the user accepts the input, a transaction is started with BEGIN WORK.
5. The primary key stored in the program record is used to SELECT the same row into the local record. FOR UPDATE locks the row.
6. The SQLCA.SQLCODE is checked, in case the database row was deleted after the initial query.
7. The work record and the local record are compared, in case the database row was changed after the initial query.
8. If the work and local records are identical, the database row is updated using the new program record values input by the user.
9. If the UPDATE is successful, a COMMIT WORK is issued. Otherwise, a ROLLBACK WORK is issued.
10. The SCROLL CURSOR has remained open, allowing the user to continue to scroll through the query result set.

SELECT ... FOR UPDATE

To explicitly lock a database row prior to updating, a SELECT ... FOR UPDATE statement may be used to instruct the database server to lock the row that was selected. SELECT ... FOR UPDATE cannot be used outside of an explicit transaction. The locks are held until the end of the transaction.

SCROLL CURSOR WITH HOLD

Like many programs that perform database maintenance, the Query program uses a SCROLL CURSOR to move through an SQL result set, updating or deleting the rows as needed. BDL cursors are automatically closed by the database interface when a COMMIT WORK or ROLLBACK WORK statement is performed. To allow the user to continue to scroll through the result set, the SCROLL CURSOR can be declared WITH HOLD, keeping it open across multiple transactions.

Example: Updating a Row in the customer table

Module custquery.4gl

The module has been modified to define a **work_custrec** record that can be used as working storage when a row is being updated.

Module custquery.4gl

```
01
02 SCHEMA custdemo
03
04 DEFINE  mr_custrec, work_custrec RECORD
05     store_num    LIKE customer.store_num,
06     store_name   LIKE customer.store_name,
07     addr         LIKE customer.addr,
08     addr2        LIKE customer.addr2,
09     city         LIKE customer.city,
10     state        LIKE customer.state,
11     zipcode      LIKE customer.zipcode,
12     contact_name LIKE customer.contact_name,
13     phone        LIKE customer.phone
14    END RECORD
...
```

Notes:

- Lines 04 thru 15 define a **work_custrec** record that is modular in scope and contains the identical structure as the **mr_custrec** program record.

The function **inpupd_cust** in the **custquery.4gl** module has been modified so it can also be used to obtain values for the Update of existing rows in the **customer** table.

Function inpupd_cust (custquery.4gl)

```
01 FUNCTION inpupd_cust(au_flag)
02   DEFINE au_flag  CHAR(1),
03          cont_ok  SMALLINT
04
05   INITIALIZE work_custrec.* TO NULL
06   LET cont_ok = TRUE
07
08   IF (au_flag = "A") THEN
09     MESSAGE "Add a new customer"
10     LET mr_custrec.* = work_custrec.*
11   ELSE
12     MESSAGE "Update customer"
13     LET work_custrec.* = mr_custrec.*
14   END IF
```

```
15
16   LET INT_FLAG = FALSE
17
18   INPUT BY NAME mr_custrec.*
19      WITHOUT DEFAULTS ATTRIBUTES(UNBUFFERED)
20
21     BEFORE FIELD store_num
22      IF (au_flag = "U") THEN
23        NEXT FIELD store_name
24      END IF
25
26     ON CHANGE store_num
27      IF (au_flag = "A") THEN
...
28     AFTER FIELD store_name
29      IF (mr_custrec.store_name IS NULL) THEN
...
30
31   END INPUT
```

Notes:

- Line 05 sets the **work_custrec** program record to NULL.
- Line 10 For an Add, the **mr_custrec** program record is set equal to the **work_custrec** record, in effect setting **mr_custrec** to NULL. The LET statement uses less resources than INITIALIZE.
- Line 13 For an Update, the values in the **mr_custrec** program record are copied into **work_custrec**, saving them for comparison later.
- Lines 21 thru 24 A BEFORE FIELD **store_num** clause has been added to the INPUT statement. If this is an Update, the user should not be allowed to change **store_num**, and the NEXT FIELD instruction moves the focus to the **store_name** field.
- Line 26 The ON CHANGE **store_num** control block, which will only execute if the au_flag is set to "A" (the operation is an Add) remains the same.
- Line 28 The AFTER FIELD **store_name** control block remains the same, and will execute if the operation is an Add or an Update.

A new function **update_cust** in the **custquery.4gl** module updates the row in the customer table.

Function update_cust (custquery.4gl)

```
01 FUNCTION update_cust()
02   DEFINE l_custrec RECORD
03     store_num    LIKE customer.store_num,
04     store_name   LIKE customer.store_name,
05     addr         LIKE customer.addr,
06     addr2        LIKE customer.addr2,
07     city         LIKE customer.city,
08     state        LIKE customer.state,
09     zipcode      LIKE customer.zipcode,
```

```
    contact_name LIKE customer.contact_name,
    phone        LIKE customer.phone
  END RECORD,
  cont_ok INTEGER

 LET cont_ok = FALSE

 BEGIN WORK

 SELECT store_num,
     store_name,
     addr,
     addr2,
     city,
     state,
     zipcode,
     contact_name,
     phone
   INTO l_custrec.* FROM customer
   WHERE store_num = mr_custrec.store_num
   FOR UPDATE

 IF (SQLCA.SQLCODE = NOTFOUND) THEN
   ERROR "Store has been deleted"
   LET cont_ok = FALSE
 ELSE
   IF (l_custrec.* = work_custrec.*) THEN
    WHENEVER ERROR CONTINUE
    UPDATE customer SET
       store_name = mr_custrec.store_name,
      addr = mr_custrec.addr,
      addr2 = mr_custrec.addr2,
      city = mr_custrec.city,
      state = mr_custrec.state,
      zipcode = mr_custrec.zipcode,
      contact_name = mr_custrec.contact_name,
      phone = mr_custrec.phone
     WHERE store_num = mr_custrec.store_num
    WHENEVER ERROR STOP
    IF (SQLCA.SQLCODE = 0) THEN
      LET cont_ok = TRUE
      MESSAGE "Row updated"
    ELSE
      LET cont_ok = FALSE
      ERROR SQLERRMESSAGE
    END IF
  ELSE
    LET cont_ok = FALSE
    LET mr_custrec.* = l_custrec.*
    MESSAGE "Row updated by another user."
  END IF
END IF

IF (cont_ok = TRUE) THEN
  COMMIT WORK
ELSE
```

```
66    ROLLBACK WORK
67  END IF
68
69 END FUNCTION
```

Notes:

- Lines 02 thru 12 define a local record, **l_custrec** with the same structure as the modular program records **mr_custrec** and **work_custrec**.
- Line 15 The **cont_ok** variable will be used as a flag to determine whether the Update should be committed or rolled back.
- Line 17 Since this will be a multiple-statement transaction, the BEGIN WORK statement is used to start the transaction.
- Lines 19 thru 30 use the **store_num** value in the program record to re-select the row. FOR UPDATE locks the database row until the transaction ends.
- Lines 32 thru 34 check SQLCA.SQLCODE to make sure the record has not been deleted by another user. If so, an error message is displayed, and the variable **cont_ok** is set to FALSE.
- Lines 36 thru 60 are to be executed if the database row was found.
- Line 36 compares the values in the **l_custrec** local record with the **work_custrec** record that contains the original values of the database row. All the values must match for the condition to be TRUE.
- Lines 37 thru 55 are executed if the values matched. An embedded SQL statement is used to UPDATE the row in the customer table using the values which the user has previously entered in the **mr_custrec** program record. The SQL UPDATE statement is surrounded by WHENEVER ERROR statements. The SQLCA.SQLCODE is checked after the UPDATE, and if it indicates the update was not successful the variable **cont_ok** is set to FALSE and an error message is displayed.
- Lines 57 through 59 are executed if the values in **l_custrec** and **work_custrec** did not match. The variable **cont_ok** is set to FALSE. The values in the **mr_custrec** program record are set to the values in the **l_custrec** record (the current values in the database row, retrieved by the SELECT .. FOR UPDATE statement.) The UNBUFFERED attribute of the INPUT statement assures that the values will be automatically displayed in the form. A message is displayed indicating the row had been changed by another user.
- Lines 63 thru 67 If the variable **cont_ok** is TRUE (the update was successful) the program issues a COMMIT WORK to end the transaction begun on Line 278. If not, a ROLLBACK WORK is issued. All locks placed on the database row are automatically released.

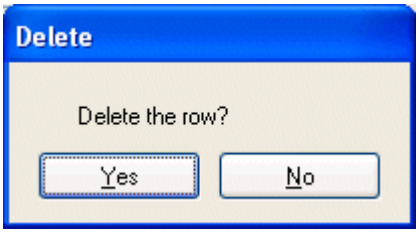
Deleting a Row

The SQL DELETE statement can be used to delete rows from the database table. The primary key of the row to be deleted can be obtained from the values in the program record.

Using a dialog Menu to prompt for validation

The MENU statement has an optional STYLE attribute that can be set to 'dialog', automatically opening a temporary modal window. You can also define a message and icon with the COMMENT and IMAGE attributes. This provides a simple way to prompt the user to confirm some action or operation that has been selected.

The menu options appear as buttons at the bottom of the window. Unlike standard menus, the dialog menu is automatically exited after any action clause such as ON ACTION, COMMAND or ON IDLE. You do not need an EXIT MENU statement.



Example: Deleting a Row

Function **delete_check** is added to the **custquery.4gl** module to check whether a store has any orders in the database before allowing the user to delete the store from the customer table. If there are no existing orders, a dialog MENU is used to prompt the user for confirmation.

Function delete_check (custquery.4gl)

```
01 FUNCTION delete_check()
02   DEFINE del_ok SMALLINT,
03          ord_count SMALLINT
04
05   LET del_ok = FALSE
06
07   SELECT COUNT(*) INTO ord_count
08     FROM orders
09     WHERE orders.store_num =
10        mr_custrec.store_num
11
12   IF ord_count > 0 THEN
13    MESSAGE "Store has existing orders"
14   ELSE
15    MENU "Delete" ATTRIBUTES (STYLE="dialog",
16      COMMENT="Delete the row?")
17    COMMAND "Yes"
18      LET del_ok = TRUE
19    COMMAND "No"
20      MESSAGE "Delete canceled"
21    END MENU
22  END IF
```

```
23
24  RETURN del_ok
25
26 END FUNCTION
```

Notes:

- Line 02 defines a variable **del_ok** to be used as a flag to determine if the Delete should continue.
- Line 05 sets **del_ok** to FALSE.
- Lines 07 thru 10 use the **store_num** value in the **mr_custrec** program record in an SQL statement to determine whether there are orders in the database for that **store_num.** The variable **ord_count** is used to store the value returned by the SELECT statement.
- Lines 12 thru 13 If the count is greater than zero, there are existing rows in the **orders** table for the **store_num**. A message is displayed to the user. del_ok remains set to FALSE.
- Lines 15 thru 21 If the count is zero, the Delete can continue. A MENU statement is used to prompt the user to confirm the Delete action. The STYLE attribute is set to "dialog" to automatically display the MENU in a modal dialog window. If the user selects "Yes", the variable **del_ok** is set to TRUE. Otherwise a message is displayed to the user indicating the Delete will be canceled.
- Line 24 returns the value of **del_ok** to the **delete_cust** function.

The function **delete_cust** is added to the **custquery.4gl** module to delete the row from the customer table.

Function delete_cust (custquery.4gl)

```
01 FUNCTION delete_cust()
02
03   WHENEVER ERROR CONTINUE
04   DELETE FROM customer
05      WHERE store_num = mr_custrec.store_num
06   WHENEVER ERROR STOP
07   IF SQLCA.SQLCODE = 0 THEN
08      MESSAGE "Row deleted"
09      INITIALIZE mr_custrec.* TO NULL
10   ELSE
11      ERROR SQLERRMESSAGE
12   END IF
13
14 END FUNCTION
```

Notes:

- Lines 04 and 05 contains an embedded SQL DELETE statement that uses the **store_num** value in the program record **mr_custrec** to delete the database row. The SQL statement is surrounded by WHENEVER ERROR statements.

This is a singleton transaction that will be automatically committed if it is successful.

- Lines 07 thru 12 check the SQLCA.SQLCODE returned for the SQL DELETE statement. If the DELETE was successful, a message is displayed and the **mr_custrec** program record values are set to NULL and automatically displayed on the form. Otherwise, an error message is displayed.

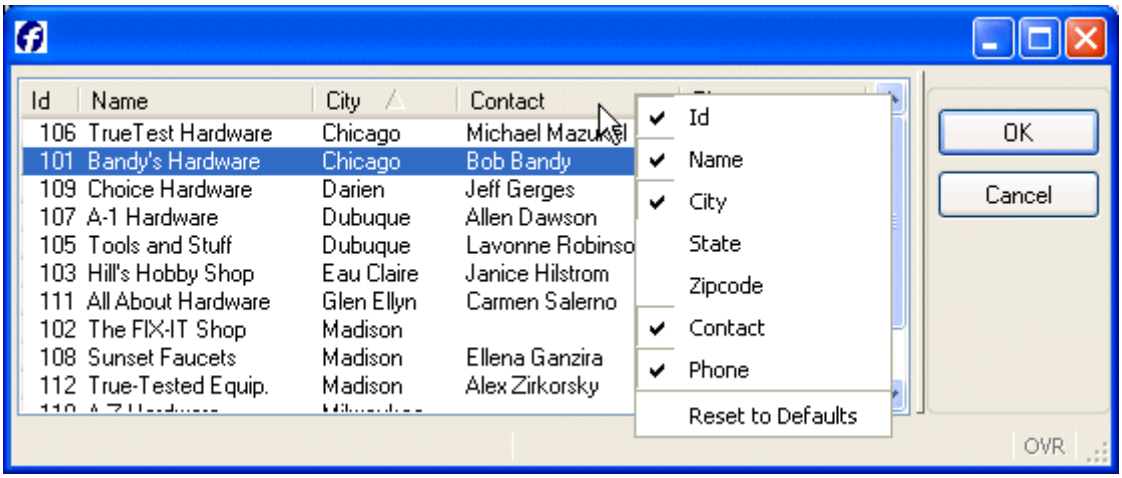
Tutorial Chapter 7: Array Display

Summary:

- Defining the Form
  - Screen Arrays
  - Table Container
  - Instructions Section
- Form example: manycust.per
- Creating the Function
  - Program Arrays
  - Loading the Array: FOREACH
  - The DISPLAY ARRAY Statement
- The DISPLAY ARRAY Statement
  - The COUNT attribute
  - The ARR_CURR function
- Example: library function to display an Array
- Compiling and using a library
- Paged mode of DISPLAY ARRAY

Unlike the previous programs, this example displays multiple customer records at once. The program defines a program array to hold the records, and displays the records in a form containing a TABLE and a screen array. The user can scroll through the records in the table, sort the table by a specific column, and hide or display columns.

This example is written as a library function so it can be used in multiple programs. This type of code re-use maximizes your programming efficiency. As you work through the examples in the other tutorial lessons, look for additional candidates for library functions.



Display on Windows platform

In the illustration, the table is sorted by City. A right mouse click has displayed a dropdown list of the columns, with check boxes allowing the user to hide or show a specific column. After the user validates the row selected, the store number and store name are returned to the calling function.

To implement this type of scrolling display, the example must:

- Create a form specification file containing a screen array of screen records
- Define an program array of records, each record having members that correspond to the fields of the screen records.

The function will use the DISPLAY ARRAY statement to display all the records in the program array into the rows of the screen array. Typically the program array has many more rows of data than will fit on the screen.

Defining the Form

Screen Arrays

A screen array is usually a repetitive array of fields in the LAYOUT section of a form specification, each containing identical groups of screen fields. Each “row” of a screen array is a screen record. Each “column” of a screen array consists of fields with the same item tag in the LAYOUT section of the form specification file. You must declare screen arrays in the INSTRUCTIONS section.

TABLE Containers

The TABLE container in a form defines the presentation of a list of records, bound to a screen array. When this layout container is used with curly braces defining the container area, the position of the static labels and item tags is automatically detected by the form compiler to build a graphical object displaying a list of records.

The first line of the TABLE area contains text entries defining the column titles. The second line contains field item tags that define the columns of the table receiving the data. This line is repeated to allow the display of multiple records at once.

The user can sort the rows displayed in the form table by a mouse-click on the title of the column that is to be used for the sort. This sort is performed on the client side only. The columns and the entire form can be stretched and re-sized. A right-mouse-click on a column title displays a dropdown list-box of column names, with radio buttons allowing the user to indicate whether a specific column is to be hidden or shown.

The INSTRUCTIONS section

You must declare a screen array in the INSTRUCTIONS section of the form with the SCREEN RECORD keyword. You can reference the names of the screen array in the DISPLAY ARRAY statement of the program.

Form example: manycust.per

Module custmain.4gl

```
01 SCHEMA custdemo
02
03 LAYOUT
04  TABLE
05  {
06   Id   Name           ...   Zipcode   Contact          Phone
07  [f01][f02           ]     [f05     ][f06           ][f07         ]
08  [f01][f02           ]     [f05     ][f06           ][f07         ]
09  [f01][f02           ]     [f05     ][f06           ][f07         ]
10  [f01][f02           ]     [f05     ][f06           ][f07         ]
11  [f01][f02           ]     [f05     ][f06           ][f07         ]
12  [f01][f02           ]     [f05     ][f06           ][f07         ]
13  }
14  END
15 END
16
17 TABLES
18   customer
19 END
20
21 ATTRIBUTES
22 EDIT f01=customer.store_num;
23 EDIT f02=customer.store_name;
24 EDIT f03=customer.city;
25 EDIT f04=customer.state;
26 EDIT f05=customer.zipcode;
27 EDIT f06=customer.contact_name;
28 EDIT f07=customer.phone;
29 END
30
31 INSTRUCTIONS
32 SCREEN RECORD sa_cust (customer.*);
33 END
```

Notes:

In order to fit on the page, the layout section of the form is truncated, not displaying the city and state columns.

- Line 01 The **custdemo** schema will be used by the compiler to determine the data types of the form fields.

- Line 06 contains the titles for the columns in the TABLE.
- Lines 07 thru 12 define the display area for the screen records. These rows must be identical in a TABLE. (The fields for **city** and **state** are indicated by **....** so the layout will fit on this page.)
- Line 21 thru 29 In the ATTRIBUTES section the field item tags are linked to the field description. Although there are multiple occurrences of each item tags in the form, the description is listed only once for each unique field item tag.
- Line 32 defines the screen array in the INSTRUCTIONS section. The screen record must contain the same number of elements as the records in the TABLE container. This example defines the screen record with all fields defined with the **customer** prefix, but you can list each field name individually.

Creating the Function

Program Arrays

A program array is an ordered set of elements all of the same data type. You can create one-, two-, or three-dimensional arrays. The elements of the array can be simple types or they can be records.

Arrays can be:

- static - defined with an explicit size for all dimensions.
- dynamic - has a variable size. Dynamic arrays have no theoretical size limit.

All elements of static arrays are initialized even if the array is not used. Therefore, defining huge static arrays may use a lot of memory. The elements of dynamic arrays are allocated automatically by the runtime system, as needed.

Example of a dynamic array of records definition:

```
01 DEFINE cust_arr DYNAMIC ARRAY OF RECORD
02                    store_num LIKE customer.store_num,
03                    city      LIKE customer.city
04                  END RECORD
```

This array variable is named **cust_arr**; each element of the array contains the members **store_num** and **city**. The size of the array will be determined by the runtime system, based on the program logic that is written to fill the array. The first element of any array is indexed with subscript 1. You would access the **store_num** member of the 10th element of the array by writing **cust_arr[10].store_num**.

Loading the Array: the FOREACH Statement

To load the program array in the example, you must retrieve the values from the result set of a query and load them into the elements of the array. You must DECLARE the cursor before the FOREACH statement can retrieve the rows.The FOREACH statement

is equivalent to using the OPEN, FETCH and CLOSE statements to retrieve and process all the rows selected by a query, and is especially useful when loading arrays.

```
01  DECLARE custlist_curs CURSOR FOR
02        SELECT store_num, city FROM customer
03  CALL cust_arr.clear()
04  FOREACH custlist_curs INTO cust_rec.*
05      CALL cust_arr.appendElement()
06     LET cust_arr[cust_arr.getLength()].* = cust_rec.*
07   END FOREACH
```

The FOREACH statement shown above:

1. Opens the **custlist_curs** cursor.
2. Clears the **cust_arr** array.
3. Fetches a row into the record **cust_rec**. This record must be defined as having the same structure as a single element of the **cust_arr** array (store_num, city).
4. Appends an empty element to the **cust_arr** array.
5. Copies the cust_rec record into the array **cust_arr** using the getLength method to determine the index of the element that was newly appended to the array.
6. Repeats steps 3, 4 and 5 until no more rows are retrieved from the database table (automatically checks for the NOTFOUND condition).
7. Closes the cursor and exits from the FOREACH loop.

The DISPLAY ARRAY Statement

The DISPLAY ARRAY statement lets the user view the contents of an array of records, scrolling through the display, but the user cannot change them.

The COUNT attribute

- With static arrays

  When using a static array, the number of rows to be displayed is defined by the COUNT attribute. If you do not use the COUNT attribute, the runtime system cannot determine how much data to display, and so the screen array remains empty.

- With dynamic arrays

  When using a dynamic array, the number of rows to be displayed is defined by the number of elements in the dynamic array; the COUNT attribute is ignored.

Example:

```
01   DISPLAY ARRAY cust_arr TO sa_cust.*
```

This statement will display the program array **cust_arr** to the form fields defined in the **sa_cust** screen array of the form.

By default, the DISPLAY ARRAY statement does not terminate until the user accepts or cancels the dialog; the Accept and Cancel actions are predefined and display on the form. Your program can accept the dialog instead, using the ACCEPT DISPLAY instruction.

The ARR_CURR function

When the user accepts or cancels a dialog, the ARR_CURR built-in function returns the index (subscript number) of the row in the program array that was selected (current).

Example Library module: cust_lib.4gl

Module cust_lib.4gl

```
01 SCHEMA custdemo
02
03 FUNCTION display_custarr()
04
05   DEFINE cust_arr DYNAMIC ARRAY OF RECORD
06          store_num    LIKE customer.store_num,
07          store_name   LIKE customer.store_name,
08          city         LIKE customer.city,
09          state        LIKE customer.state,
10          zipcode      LIKE customer.zipcode,
11          contact_name LIKE customer.contact_name,
12          phone        LIKE customer.phone
13    END RECORD,
14    cust_rec RECORD
15          store_num    LIKE customer.store_num,
16          store_name   LIKE customer.store_name,
17          city         LIKE customer.city,
18          state        LIKE customer.state,
19          zipcode      LIKE customer.zipcode,
20          contact_name LIKE customer.contact_name,
21          phone        LIKE customer.phone
22    END RECORD,
23    ret_num LIKE customer.store_num,
24    ret_name LIKE customer.store_name,
25    curr_pa SMALLINT
26
27    OPEN WINDOW wcust WITH FORM "manycust"
28
29    DECLARE custlist_curs CURSOR FOR
30      SELECT store_num,
31             store_name,
32             city,
33             state,
34             zipcode,
35             contact_name,
36             phone
37        FROM customer
38        ORDER BY store_num
```

```
39
40
41   CALL cust_arr.clear()
42   FOREACH custlist_curs INTO cust_rec.*
43    CALL cust_arr.appendElement()
44    LET cust_arr[cust_arr.getLength()].* = cust_rec.*
45   END FOREACH
46
47   LET ret_num = 0
48   LET ret_name = NULL
49
50   IF (cust_arr.getLength() > 0) THEN
51    DISPLAY ARRAY cust_arr TO sa_cust.*
52    IF (NOT INT_FLAG) THEN
53       LET curr_pa = arr_curr()
54       LET ret_num = cust_arr[curr_pa].store_num
55       LET ret_name = cust_arr[curr_pa].store_name
56    END IF
57
58
59   CLOSE WINDOW wcust
60   RETURN ret_num, ret_name
61
62 END FUNCTION
```

Notes:

- Lines 05 thru 13 define a local program array, **cust_arr**.
- Lines 14 thru 22 define a local program record, **cust_rec**. This record is used as temporary storage for the row data retrieved by the FOREACH loop in line 42.
- Lines 23 and 24 define local variables to hold the store number and name values to be returned to the calling function.
- Line 25 defines a variable to store the value of the program array index.
- Line 27 opens a window with the form containing the array.
- Lines 29 thru 38 DECLARE the cursor **custlist_curs** to retrieve the rows from the customer table.
- Line 40 sets the variable **idx** to 0, this variable will be incremented in the FOREACH loop.
- Line 41 clear the dynamic array.
- Line 42 uses FOREACH to retrieve each row from the result set into the program record, **cust_rec**.
- Lines 43 thru 44 are executed for each row that is retrieved by the FOREACH. They append a new element to the array cust_arr, nd transfer the data from the program record into new element, using the method getLength to identify the index of the element. When the FOREACH statement has retrieved all the rows the cursor is closed and the FOREACH is exited.
- Lines 47 and 48 Initialize the variables used to return the customer number and customer name.
- Lines 50 thru 57 If the length of the **cust_arr** array is greater than 0, the FOREACH statement did retrieve some rows.
- Line 52 DISPLAY ARRAY turns control over to the user, and waits for the user to accept or cancel the dialog.

- Line 52 The INT_FLAG variable is tested to check if the user validated the dialog.
- Line 53 If the user has validated the dialog, the built-in function ARR_CURR is used to store the index for the program array element the user had selected (corresponding to the highlighted row in the screen array) in the variable **curr_pa**.
- Lines 54 and 55 The variable **curr_pa** is used to retrieve the current values of **store_num** and **store_name** from the program array and store them in the variables **ret_num** and **ret_name**.
- Line 59 closes the window.
- Line 60 returns **ret_num** and **ret_name** to the calling function.

Compiling and using a Library

Since this is a function that could be used by other programs that reference the **customer** table, the function will be compiled into a library. The library can then be linked into any program, and the function called. The function will always return **store_num** and **store_name**. If the FOREACH fails, or returns no rows, the calling program will have a **store_num** of zero and a NULL **store_name** returned.

The function is contained in a file named **cust_lib.4gl**. This file would usually contain additional library functions. To compile (and link, if there were additional .4gl files to be included in the library):

```
fgl2p -o cust_lib.42x cust_lib.4gl
```

Since a library has no MAIN function, we will need to create a small stub program if we want to test the library function independently. This program contains the minimal functionality to test the function.

Example: cust_stub.4gl

Module cust_stub.4gl

```
01 SCHEMA custdemo
02
03 MAIN
04   DEFINE store_num  LIKE customer.store_num,
05          store_name LIKE customer.store_name
06
07  DEFER INTERRUPT
08  CONNECT TO "custdemo"
09  CLOSE WINDOW SCREEN
10
11  CALL display_custarr()
12            RETURNING store_num, store_name
13  DISPLAY store_num, store_name
14
15  DISCONNECT CURRENT
16
```

```
17 END MAIN
```

Notes:

- Lines 04 and 05 define variables to hold the values returned by the display_custarr function.
- Lines 07 thru 09 are required simply for the test program, to set the program up and connect to the database.
- Line 11 calls the library function **display_custarr**.
- Line 13 displays the returned values to standard output for the purposes of the test.

Now we can compile the form file and the test program, and link the library, and then test to see if it works properly.

```
fglform manycust.per
fgl2p -o test.42r cust_stub.4gl cust_lib.42x
fglrun test.42r
```

Paged Mode of DISPLAY ARRAY

The previous example retrieves all the rows from the customer table into the program array prior to the data being displayed by the DISPLAY ARRAY statement. Using this full list mode, you must copy into the array all the data you want to display. Using the DISPLAY ARRAY statement in "paged" mode allows you to provide data rows dynamically during the dialog, using a dynamic array to hold one page of data.

The following example modifies the program to use a SCROLL CURSOR to retrieve only the **store_num** values from the customer table. As the user scrolls thru the result set, statements in the ON FILL BUFFER clause of the DISPLAY ARRAY statement are used to retrieve and display the remainder of each row, a page of data at a time. This helps to minimize the possibility that the rows have been changed, since the rows are re-selected immediately prior to the page being displayed.

What is the "Paged mode"?

A "page" of data is the total number of rows of data that can be displayed in the form at one time. The length of a page can change dynamically, since the user has the option of re-sizing the window containing the form. The run-time system automatically keeps track of the current length of a page.

The ON FILL BUFFER clause feeds the DISPLAY ARRAY instruction with pages of data. The following built-in functions are used in the ON FILL BUFFER clause to provide the rows of data for the page:

- FGL_DIALOG_GETBUFFER START() - retrieves the offset in the SCROLL CURSOR result set, and is used to determine the starting point for retrieving and displaying the complete rows.
- FGL_DIALOG_GETBUFFERLENGTH() - retrieves the current length of the page, and is used to determine the number of rows that must be provided.

The statements in the ON FILL BUFFER clause of DISPLAY ARRAY are executed automatically by the runtime system each time a new page of data is needed. For example, if the current size of the window indicates that ten rows can be displayed at one time, the statements in the ON FILL BUFFER clause will automatically maintain the dynamic array so that the relevant ten rows are retrieved and/or displayed as the user scrolls up and down through the table on the form. If the window is re-sized by the user, the statements in the ON FILL BUFFER clause will automatically retrieve and display the new number of rows.

AFTER DISPLAY block

The AFTER DISPLAY block is executed one time, after the user has accepted or canceled the dialog, but before executing the next statement in the program. In this program, the statements in this block determine the current position of the cursor when user pressed OK or Cancel, so the correct store number and name can be returned to the calling function.

Example of paged mode

In the first example, the records in the customer table are loaded into the program array and the user uses the form to scroll through the program array. In this example, the user is actually scrolling through the result set created by a SCROLL CURSOR. This SCROLL CURSOR retrieves only the store number, and another SQL SELECT statement is used to retrieve the remainder of the row as needed.

Module cust_lib2.4gl

```
01 SCHEMA custdemo
02
03 FUNCTION display_custarr()
04
05  DEFINE cust_arr DYNAMIC ARRAY OF RECORD
06          store_num     LIKE customer.store_num,
07          store_name    LIKE customer.store_name,
08          city          LIKE customer.city,
09          state         LIKE customer.state,
10          zipcode       LIKE customer.zipcode,
11          contact_name  LIKE customer.contact_name,
12          phone         LIKE customer.phone
13      END RECORD,
14      ret_num      LIKE customer.store_num,
15      ret_name     LIKE customer.store_name,
16      ofs, len, i  SMALLINT,
```

```
17      sql_text     STRING,
18      rec_count    SMALLINT,
19      curr_pa      SMALLINT
20
21  OPEN WINDOW wcust WITH FORM "manycust"
22
23  LET rec_count = 0
24  SELECT COUNT(*) INTO rec_count FROM customer
25  IF (rec_count == 0) THEN
26     RETURN 0, NULL
27  END IF
28
29  LET sql_text =
30     "SELECT store_num, store_name, city,"
31     || " state, zipcode, contact_name,"
32     || " phone"
33     || " FROM customer WHERE store_num = ?"
34  PREPARE rec_all FROM sql_text
35
36  DECLARE num_curs SCROLL CURSOR FOR
37         SELECT store_num FROM customer
38  OPEN num_curs
39
40  DISPLAY ARRAY cust_arr TO sa_cust.*
41       ATTRIBUTES(UNBUFFERED, COUNT=rec_count)
42
43     ON FILL BUFFER
44      LET ofs = FGL_DIALOG_GETBUFFERSTART()
45      LET len = FGL_DIALOG_GETBUFFERLENGTH()
46      FOR i = 1 TO len
47        WHENEVER ERROR CONTINUE
48        FETCH ABSOLUTE ofs+i-1 num_curs
49                     INTO cust_arr[i].store_num
50         EXECUTE rec_all INTO cust_arr[i].*
51                 USING cust_arr[i].store_num
52        WHENEVER ERROR STOP
53        IF (SQLCA.SQLCODE = NOTFOUND) THEN
54          MESSAGE "Row deleted" by another user."
55          CONTINUE FOR
56        ELSE
57          IF (SQLCA.SQLCODE < 0) THEN
58            ERROR SQLERRMESSAGE
59            CONTINUE FOR
60          END IF
61        END IF
62      END FOR
62
64     AFTER DISPLAY
65      IF (INT_FLAG) THEN
66         LET ret_num = 0
67         LET ret_name = NULL
68      ELSE
69         LET curr_pa = ARR_CURR()- ofs + 1
70         LET ret_num = cust_arr[curr_pa].store_num
71         LET ret_name = cust_arr[curr_pa].store_name
72      END IF
```

```
73
74  END DISPLAY
75
76  CLOSE num_curs
77  FREE num_curs
78  FREE rec_all
79
80  CLOSE WINDOW wcust
81  RETURN ret_num, ret_name
82
83 END FUNCTION
```

Notes:

- Lines 16 thru 19 define some new variables to be used, including **cont_disp** to indicate whether the function should continue.
- Line 24 uses an embedded SQL statement to store the total number of rows in the customer table in the variable **rec_count**.
- Lines 25 thru 27 If the total number of rows is zero, function returns immediately 0 and NULL.
- Lines 29 thru 33 contain the text of an SQL SELECT statement to retrieve values from a single row in the **customer** table. The **?** placeholder will be replaced with the store number when the statement is executed. This text is assigned to a STRING variable, **sql_text**.
- Line 34 uses the SQL PREPARE statement to convert the STRING into an executable statement, **rec_all**. This statement will be executed when needed, to populate the rest of the values in the row of the program array.
- Lines 36 thru 37 DECLARE a SCROLL CURSOR **num_curs** to retrieve only the store number from the customer table.
- Line 38 opens the SCROLL CURSOR **num_curs**.
- Lines 40 and 41 call the DISPLAY ARRAY statement, providing the COUNT to let the statement know the total number of rows in the SQL result set.
- Lines 43 thru 62 contain the logic for the ON FILL BUFFER clause of the DISPLAY ARRAY statement. This control block will be executed automatically whenever a new page of data is required.
- Line 44 uses the built-in function to get the offset for the page, the starting point for the retrieval of rows, and stores it in the variable **ofs**.
- Line 45 uses the built-in function to get the page length, and stores it in the variable **len**.
- Lines 46 thru 62 contain a FOR loop to populate each row in the page with values from the customer table. The variable **i** is incremented to populate successive rows. The first value of **i** is 1.
- Lines 48 and 49 use the SCROLL CURSOR **num_curs** with the syntax FETCH ABSOLUTE **<***row_number***>** to retrieve the store number from a specified row in the result set, and to store it in row **i** of the program array. Since **i** was started at 1, the following calculation is used to determine the row number of the row to be retrieved:

```
(Offset for the page) PLUS i MINUS 1
```

Notice that rows 1 thru (*page_ length*) of the program array are filled each time a new page is required.

- Lines 50 and 51 execute the prepared statement **rec_all** to retrieve the rest of the values for row **i** in the program array, using the store number retrieved by the SCROLL CURSOR. Although this statement is within the FOR loop, it was prepared earlier in the program, outside of the loop, to avoid unnecessary re-processing each time the loop is executed.
- Lines 53 thru 61 test whether fetching the entire row was successful. If not, a message is displayed to the user, and the CONTINUE FOR instruction continues the FOR loop with the next iteration.
- Lines 64 thru 72 use an AFTER DISPLAY statement to get the row number of the row in the array that the user had selected. If the dialog was cancelled, **ret_num** is set to 0 and **ret_name** is set to blanks. Otherwise the values of **ret_num** and **ret_name** are set based on the row number. The row number in the SCROLL CURSOR result set does not correlate directly to the program array number, because the program array was filled starting at row 1 each time. So the following calculation is used to return the correct row number of the program array:

```
(Row number returned by ARR_CURR) MINUS
                (Offset for the page) PLUS 1
```

- Line 74 is the end of the DISPLAY ARRAY statement.
- Lines 76 and 77 close and free the cursor.
- Line 78 frees the prepared statement.
- Line 81 closes the window.
- Line 82 returns the values of the variables **ret_num** and **ret_name** to the calling function.

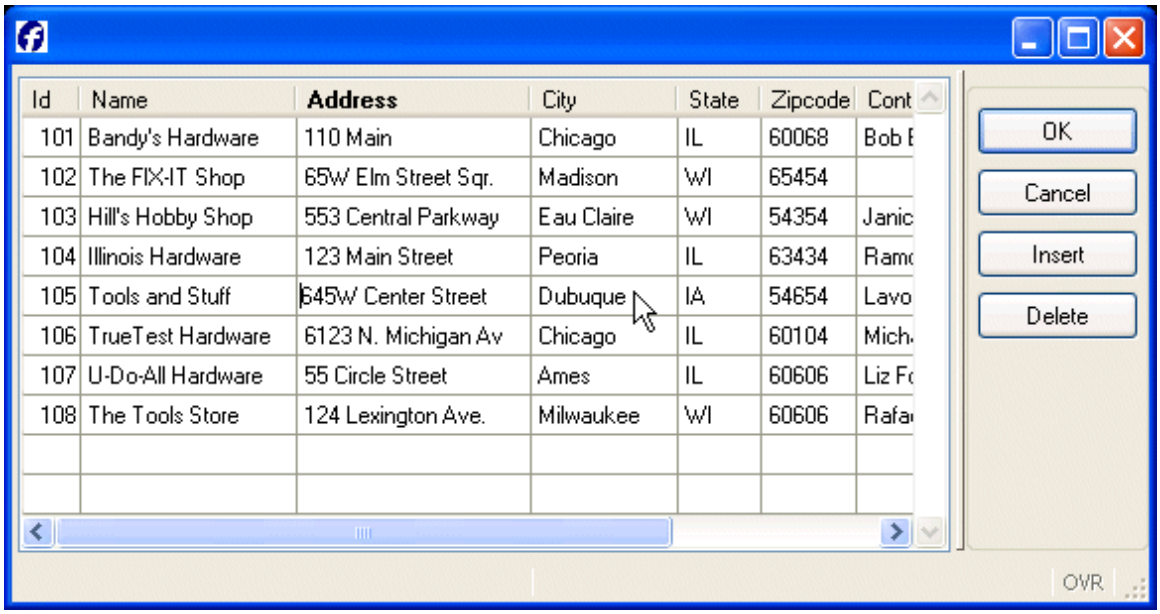
Tutorial Chapter 8: Array Input

Summary:

- INPUT ARRAY statement
- WITHOUT DEFAULTS clause
- The UNBUFFERED attribute
- COUNT and MAXCOUNT attributes
- Control Blocks
- Built-in Functions - ARR_CURR()
- Predefined Actions - insert and delete
- Example: Using a Screen Array to Modify Data
  - Form Specification File
  - The Main Block
  - Function load_custall
  - Function inparr_custall
  - Function store_num_ok
  - Function insert_cust
  - Function update_cust
  - Function delete_cust

This program uses a form and a screen array to allow the user to view and change multiple records of a program array at once. The INPUT ARRAY statement and its control blocks are used by the program to control and monitor the changes made by the user to the records. As each record in the program array is Added, Updated, or Deleted, the program logic makes corresponding changes in the rows of the **customer** database table.

The example window shown below has been re-sized to fit on this page.



The INPUT ARRAY statement

The INPUT ARRAY statement supports data entry by users into a screen array, and stores the entered data in a program array of records. During the INPUT ARRAY execution, the user can edit or delete existing records, insert new records, and move inside the list of records. The program can then use the INSERT, DELETE or UPDATE SQL statements to modify the appropriate database tables. The INPUT ARRAY statement does not terminate until the user validates or cancels the dialog.

```
INPUT ARRAY cust_arr WITHOUT DEFAULTS FROM sa_cust.*
     ATTRIBUTES (UNBUFFERED)
```

The example INPUT ARRAY statement binds the screen array fields in **sa_cust** to the member records of the program array **cust_arr**. The number of variables in each record of the program array must be the same as the number of fields in each screen record (that is, in a single row of the screen array). Each mapped variable must have the same data type or a compatible data type as the corresponding field.

WITHOUT DEFAULTS clause

The WITHOUT DEFAULTS clause prevents BDL from displaying any default values that have been defined for form fields. You must use this clause if you want to see the values of the program array.

The UNBUFFERED attribute

As in the INPUT statement, when the UNBUFFERED attribute is used, the INPUT ARRAY statement is sensitive to program variable changes. If you need to display new data during the execution, use the UNBUFFERED attribute and assign the values to the program array row; the runtime system will automatically display the values to the screen. This sensitivity applies to ON ACTION control blocks, as well.

COUNT and MAXCOUNT attributes

Some other attributes that can be used with an INPUT ARRAY statement are:

- The COUNT attribute of INPUT ARRAY defines the number of valid rows in the program array to be displayed as default rows. When using a static array, if you do not use the COUNT attribute, the runtime system cannot determine how much data to display, so the screen array remains empty. When using a dynamic array, the COUNT attribute is ignored: The number of elements in the dynamic array is used.

- The MAXCOUNT attribute defines the maximum number of data rows that can be entered in the program array. In a dynamic array, the user can enter an infinite number of rows if the MAXCOUNT attribute is not set.

Control Blocks

Your program can control and monitor the changes made by the user by using control blocks with the INPUT ARRAY statement. The control blocks that are used in the example program are:

- The BEFORE INPUT block - executed one time, before the runtime system gives control to the user. You can implement initialization in this block.
- The BEFORE ROW block - executed each time the user moves to another row, after the destination row is made the current one.
- The ON ROW CHANGE block - executed when the user moves to another row after modifications have been made to the current row.
- The ON CHANGE <fieldname> block - executed when the cursor leaves a specified field and the value was changed by the user after the field got the focus.
- The BEFORE INSERT block - executed each time the user inserts a new row in the array, before the new row is created and made the current one.
- The AFTER INSERT block - executed each time the user inserts a new row in the array, after the new row is created. You can cancel the insert operation with the CANCEL INSERT keywords.
- The BEFORE DELETE block - executed each time the user deletes a row from the array, before the row is removed from the list. You can cancel the delete operation with the CANCEL DELETE keywords.

For a more detailed explanation of the priority of control blocks see Input Array.

Built-in Functions - ARR_CURR

The language provides several built-in functions to use in an INPUT ARRAY statement. The example program uses the ARR_CURR function to tell which array element is being changed. This function returns the row number within the program array that is displayed in the current line of a screen array.

Predefined actions

There are some pre-defined actions that are specific to the INPUT ARRAY statement, to handle the insertion and deletion of rows in the screen array automatically:

- The **insert** action inserts a new row before current row. When the user has filled this record, BDL inserts the data into the program array.
- The **delete** action deletes the current record from the display of the screen array and from the program array, and redraws the screen array so that the deleted record is no longer shown.

- The **append** action adds a new row at the end of the list. When the user has filled this record, BDL inserts the data into the program array.

As with the pre-defined actions **accept** and **cancel** actions discussed in Chapter 4, if your form specification does not contain action views for these actions, default action views (buttons on the form) are automatically created. Control attributes of the INPUT ARRAY statement allow you to prevent the creation of these actions and their accompanying buttons.

Example: Using a Screen Array to modify Data

The Form Specification File

The **custallform.per** form specification file displays multiple records at once, and is similar to the form used in chapter 7. The item type of field **f6**, containing the **state** values, has been changed to COMBOBOX to provide the user with a dropdown list when data is being entered.

Form file (custallform.per)

```
01 SCHEMA custdemo
02
03 LAYOUT
04  TABLE
05  {
06   Id    Name          ..  Zipcode   Contact          Phone
07  [f01][f02          ]   [f07     ][f08            ][f09         ]
08  [f01][f02          ]   [f07     ][f08            ][f09         ]
09  [f01][f02          ]   [f07     ][f08            ][f09         ]
10  [f01][f02          ]   [f07     ][f08            ][f09         ]
11  [f01][f02          ]   [f07     ][f08            ][f09         ]
12  [f01][f02          ]   [f07     ][f08            ][f09         ]
13  }
14  END
15 END
16
17 TABLES
18  customer
19 END
20
21 ATTRIBUTES
22 EDIT     f01 = customer.store_num, REQUIRED;
23 EDIT     f02 = customer.store_name, REQUIRED;
24 EDIT     f03 = customer.addr;
25 EDIT     f04 = customer.addr2;
26 EDIT     f05 = customer.city;
27 COMBOBOX f6  = customer.state, ITEMS = ("IA", "IL", "WI");
28 EDIT     f07 = customer.zipcode;
29 EDIT     f08 = customer.contact_name;
30 EDIT     f09 = customer.phone;
31 END
```

```
32
33 INSTRUCTIONS
34 SCREEN RECORD sa_cust (customer.*);
35 END
```

The Main block

The single module program **custall.4gl** allows the user to update the **customer** table using a form that displays multiple records at once.

Main block (custall.4gl)

```
01 SCHEMA custdemo
02
03 DEFINE cust_arr DYNAMIC ARRAY OF RECORD
04         store_num    LIKE customer.store_num,
05         store_name   LIKE customer.store_name,
06         addr         LIKE customer.addr,
07         addr2        LIKE customer.addr2,
08         city         LIKE customer.city,
09         state        LIKE customer.state,
10         zipcode      LIKE customer.zipcode,
11         contact_name LIKE customer.contact_name,
12         phone        LIKE customer.phone
13        END RECORD
14
15
16 MAIN
17   DEFINE idx SMALLINT
18
19   DEFER INTERRUPT
20   CONNECT TO "custdemo"
21   CLOSE WINDOW SCREEN
22   OPEN WINDOW w3 WITH FORM "custallform"
23
24   CALL load_custall() RETURNING idx
25   IF idx > 0 THEN
26      CALL inparr_custall()
27   END IF
28
29   CLOSE WINDOW w3
30   DISCONNECT CURRENT
31
32 END MAIN
```

Notes:

- Lines 03 thru 13 define a dynamic array **cust_arr** having the same structure as the **customer** table. The array is modular is scope.
- Line 17 defines a local variable **idx**, to hold the returned value from the **load_custall** function.

- Line 20 connects to the **custdemo** database.
- Line 22 opens a window with the form **manycust.** This form contains a screen array **sa_cust** which is referenced in the program.
- Line 24 thru 27 call the function l**oad_custall** to load the array, which returns the index of the array. If the load was successful (the returned index is greater than 0) the function **inparr_custall** is called. This function contains the logic for the Input/Update/Delete of rows.
- Line 29 closes the window.
- Line 30 disconnects from the database.

Function load_custall

This function loads the program array with rows from the **customer** database table. The logic to load the rows is identical to that in Chapter 7. Although this program loads all the rows from the **customer** table, the program could be written to allow the user to query first, for a subset of the rows. A query-by-example, as illustrated in chapter 4, can also be implemented using a form containing a screen array such as **manycust**.

Function load_custall (custall.4gl)

```
01 FUNCTION load_custall()
02   DEFINE cust_rec RECORD LIKE customer.*
03
04
05   DECLARE custlist_curs CURSOR FOR
06      SELECT store_num,
07          store_name,
08          addr,
09          addr2,
10          city,
11          state,
12          zipcode,
13          contact_name,
14          phone
15        FROM customer
16        ORDER BY store_num
17
18
19   CALL cust_arr.clear()
20   FOREACH custlist_curs INTO cust_rec.*
21      CALL cust_arr.appendElement()
22      LET cust_arr[cust_arr.getLength()].* = cust_rec.*
23   END FOREACH
24
25   IF (cust_arr.getLength() == 0) THEN
26      DISPLAY "No rows loaded."
27   END IF
28
29   RETURN cust_arr.getLength()
30
31 END FUNCTION
```

Notes:

- Line 02 defines a local record variable, **cust_rec**, to hold the rows fetched in FOREACH.
- Lines 05 thru 16 declare the cursor **custlist_curs** to retrieve the rows from the **customer** table.
- Lines 20 thru 23 retrieve the rows from the result set into the program array.
- Lines 25 thru 27 If the array is empty, we display a warning message.
- Line 29 returns the number of rows to the MAIN function.

Function inparr_custall

This is the primary function of the program, driving the logic for inserting, deleting, and changing rows in the **customer** database table. Each time a record is added, deleted, or changed on the form, the values from the current record in the program array are used to update the **customer** table.

Function inparr_custall (custall.4gl)

```
01 FUNCTION inparr_custall(idx)
02
03   DEFINE curr_pa SMALLINT,
04          opflag  CHAR(1)
05
06  INPUT ARRAY cust_arr WITHOUT DEFAULTS
07      FROM sa_cust.*
08      ATTRIBUTES (UNBUFFERED)
09
10   BEFORE INPUT
11    MESSAGE "OK exits/" ||
12    "Cancel exits & cancels current operation"
13
14   BEFORE ROW
15     LET curr_pa = ARR_CURR()
16     LET opflag = "N"
17
18   BEFORE INSERT
19     LET opflag = "T"
20
21   AFTER INSERT
22     LET opflag = "I"
23
24   BEFORE DELETE
25     IF NOT (delete_cust(curr_pa)) THEN
26         CANCEL DELETE
27       END IF
28
29    ON ROW CHANGE
30      IF (opflag <> "I") THEN
31        LET opflag = "U"
32       END IF
```

```
33
34   BEFORE FIELD store_num
35    IF (opflag <> "T") THEN
36      NEXT FIELD store_name
37    END IF
38
39   ON CHANGE store_num
40    IF (opflag = "T") THEN
41       IF NOT store_num_ok(curr_pa) THEN
42        MESSAGE "Store already exists"
43        LET cust_arr[curr_pa].store_num = NULL
44        NEXT FIELD store_num
45      END IF
46    END IF
47
48    AFTER ROW
49      IF (INT_FLAG) THEN EXIT INPUT END IF
50      CASE
51      WHEN opflag = "I"
52          CALL insert_cust(curr_pa)
53        WHEN opflag = "U"
54          CALL update_cust(curr_pa)
55      END CASE
56
57  END INPUT
58
59  IF (INT_FLAG) THEN
60    LET INT_FLAG = FALSE
61  END IF
62
63 END FUNCTION -- inparr_custall
```

Notes:

- Line 03 defines the variable **curr_pa**, to hold the index number of the current record in the program array.
- Line 04 defines the variable **opflag**, to indicate whether the operation being performed on a record is an Insert ("I") or an Update ("U").
- Lines 06 thru 57 contain the INPUT ARRAY statement, associating the program array **cust_arr** with the **sa_cust** screen array on the form. The attribute WITHOUT DEFAULTS is used so the same statement can handle both Updates and Inserts of new records. The UNBUFFERED attribute insures that values entered into the program array are automatically displayed in the screen array on the form.
- Lines 10 thru 12 BEFORE INPUT control block: before the INPUT ARRAY statement is executed a MESSAGE is displayed to the user.
- Lines 14 thru 16 BEFORE ROW control block: when called in this block, the ARR_CURR function returns the index of the record that the user is moving into (which will become the current record). This is stored in a variable **curr_pa**, so the index can be passed to other control blocks. We also initialize the **opflag** to "N": This will be its value unless an update or insert is performed.

- Lines 18 and 19 BEFORE INSERT control block: just before the user is allowed to enter the values for a new record, the variable **opflag** is set to "**T**", indicating an Insert operation is in progress.
- Lines 21 and 22 AFTER INSERT control block sets the **opflag** to "**I**" after the insert operation has been completed.
- Lines 24 thru 27 BEFORE DELETE control block: Before the record is removed from the program array, the function **delete_cust** is called, which verifies that the user wants to delete the current record. In this function, when the user verifies the delete, the index of the record is used to remove the corresponding row from the database. Unless the **delete_cust** function returns TRUE, the record is not removed from the program array.
- Lines 29 thru 32 ON ROW CHANGE control block: After row modification, the program checks whether the modification was an insert of a new row. If not, the **opflag** is set to "**U**" indicating an update of an existing row.
- Lines 34 thru 37 BEFORE FIELD **store_num** control block: the **store_num** field should not be entered by the user unless the operation is an Insert of a new row, indicated by the "**T**" value of **opflag**. The **store_num** column in the **customer** database table is a primary key and cannot be updated. If the operation is not an insert, the NEXT FIELD statement is used to move the cursor to the next field in the program array, **store_name,** allowing the user to change all the fields in the record of the program array except **store_num**.
- Lines 39 thru 46 ON CHANGE **store_num** control block: if the operation is an Insert, the **store_num_ok** function is called to verify that the value that the user has just entered into the field **store_num** of the current program array does not already exist in the **customer** database table. If the store number does exist, the value entered by the user is nulled out, and the cursor is returned to the **store_num** field.
- Lines 48 thru 55 AFTER ROW control block: First, the program checks to see whether the user wants to interrupt the INPUT operation. Then, **opflag** is checked in a CASE statement, and the **insert_cust** or **update_cust** function is called based on the opflag value. The index of the current record is passed to the function so the database table can be modified.
- Line 57 indicates the end of the INPUT statement.
- Lines 59 thru 61 check the value of the interrupt flag INT_FLAG and re-set it to FALSE if necessary.

Function store_num_ok

When a new record is being inserted into the program array, this function verifies that the store number does not already exist in the **customer** database table. The logic in this function is virtually identical to that used in Chapter 5.

Function store_num_ok (custall.4gl)

```
01 FUNCTION store_num_ok(idx)
02   DEFINE idx SMALLINT,
03          checknum LIKE customer.store_num,
04          cont_ok SMALLINT
```

```
05
06   LET cont_ok = FALSE
07   WHENEVER ERROR CONTINUE
08   SELECT store_num INTO checknum
09     FROM customer
10     WHERE store_num =
11          cust_arr[idx].store_num
12   WHENEVER ERROR STOP
13   IF (SQLCA.SQLCODE = NOTFOUND) THEN
14     LET cont_ok = TRUE
15   ELSE
16     LET cont_ok = FALSE
17     IF (SQLCA.SQLCODE = 0) THEN
18       MESSAGE "Store Number already exists."
19     ELSE
20       ERROR SQLERRMESSAGE
21     END IF
22   END IF
23
24   RETURN cont_ok
25
26 END FUNCTION
```

Notes:

- Line 02 The index of the current record in the program array is stored in the variable **idx,** passed to this function from the INPUT ARRAY control block ON CHANGE **store_num**.
- Line 03 The variable **checknum** is defined to hold the **store_num** returned by the SELECT statement.
- Line 06 sets the variable **cont_ok** to an initial value of FALSE. This variable is used to indicate whether the store number is unique.
- Lines 07 thru 12 use an embedded SQL SELECT statement to check whether the s**tore_num** already exists in the customer table. The index passed to this function is used to obtain the value that was entered into the store_num field on the form. The entire database row is not retrieved by the SELECT statement since the only information required by this program is whether the store number already exists in the table. The SELECT is surrounded by WHENEVER ERROR statements.
- Lines 13 thru 22 test SQLCA.SQLCODE to determine the success of the SELECT statement. The variable **cont_ok** is set to indicate whether the store number entered by the user is unique.
- Line 24 returns the value of **cont_ok** to the calling function.

Function insert_cust

This function inserts a new row into the **customer** database table.

Function insert_cust (custall.4gl)

```
01 FUNCTION insert_cust(idx)
02   DEFINE idx SMALLINT
03
04   WHENEVER ERROR CONTINUE
05   INSERT INTO customer
06       (store_num,
07        store_name,
08        addr,
09        addr2,
10        city,
11        state,
12        zipcode,
13        contact_name,
14        phone)
15     VALUES (cust_arr[idx].* )
16   WHENEVER ERROR STOP
17
18   IF (SQLCA.SQLCODE = 0) THEN
19     MESSAGE "Store added"
20   ELSE
21     ERROR SQLERRMESSAGE
22   END IF
23
24 END FUNCTION
```

Notes:

- Line 02 This function is called from the AFTER INSERT control block of the INPUT ARRAY statement. The index of the record that was inserted into the **cust_arr** program array is passed to the function and stored in the variable **idx**.
- Lines 04 thru 16 uses an embedded SQL INSERT statement to insert a row into the **customer** database table. The values to be inserted into the **customer** table are obtained from the record just inserted into the program array. The INSERT is surrounded by WHENEVER ERROR statements.
- Lines 18 thru 22 test the SQLCA.SQLCODE to see if the insert into the database was successful, and return an appropriate message to the user.

Function update_cust

This function updates a row in the **customer** database table. The functionality is very simple for illustration purposes, but it could be enhanced with additional error checking routines similar to the example in chapter 6.

Function update_cust (custall.4gl)

```
01 FUNCTION update_cust(idx)
02   DEFINE idx SMALLINT
03
04   WHENEVER ERROR CONTINUE
05   UPDATE customer
06    SET
```

```
07     store_name   = cust_arr[idx].store_name,
08     addr         = cust_arr[idx].addr,
09     addr2        = cust_arr[idx].addr2,
10     city         = cust_arr[idx].city,
11     state        = cust_arr[idx].state,
12     zipcode      = cust_arr[idx].zipcode,
13     contact_name = cust_arr[idx].contact_name,
14     phone        = cust_arr[idx].phone
15    WHERE store_num = cust_arr[idx].store_num
16   WHENEVER ERROR STOP
17
18   IF (SQLCA.SQLCODE = 0) THEN
19      MESSAGE "Dealer updated."
20   ELSE
21      ERROR SQLERRMESSAGE
22   END IF
23
24 END FUNCTION
```

Notes:

- Line 02 The index of the current record in the cust_arr program array is passed as **idx** from the ON ROW CHANGE control block.
- Lines 04 thru 16 use an embedded SQL UPDATE statement to update a row in the **customer** database table. The index of the current record in the program array is used to obtain the value of **store_num** that is to be matched in the **customer** table. The **customer** row is updated with the values stored in the current record of the program array. The UPDATE is surrounded by WHENEVER ERROR statements.
- Lines 18 thru 22 test the SQLCA.SQLCODE to see if the update of the row in the database was successful, and return an appropriate message to the user.

Function delete_cust

This function deletes a row from the **customer** database table. A modal Menu similar to that illustrated in Chapter 6 is used to verify that the user wants to delete the row.

Function delete_cust (custall.4gl)

```
01 FUNCTION delete_cust(idx)
02   DEFINE idx     SMALLINT,
03          del_ok      SMALLINT
04
05   LET del_ok = FALSE
06
07   MENU "Delete" ATTRIBUTES (STYLE="dialog",
08               COMMENT="Delete this row?")
09    COMMAND "OK"
10      LET del_ok = TRUE
11      EXIT MENU
12    COMMAND "Cancel"
```

```
13      LET del_ok = FALSE
14      EXIT MENU
15   END MENU
16
17   IF del_ok = TRUE THEN
18     WHENEVER ERROR CONTINUE
20     DELETE FROM customer
21        WHERE store_num = cust_arr[idx].store_num
22     WHENEVER ERROR STOP
23
24     IF (SQLCA.SQLCODE = 0) THEN
25       LET del_ok = TRUE
26       MESSAGE "Dealer deleted."
27     ELSE
28       LET del_ok = FALSE
29       ERROR SQLERRMESSAGE
30     END IF
31   END IF
32
33   RETURN del_ok
34
35 END FUNCTION
```

Notes:

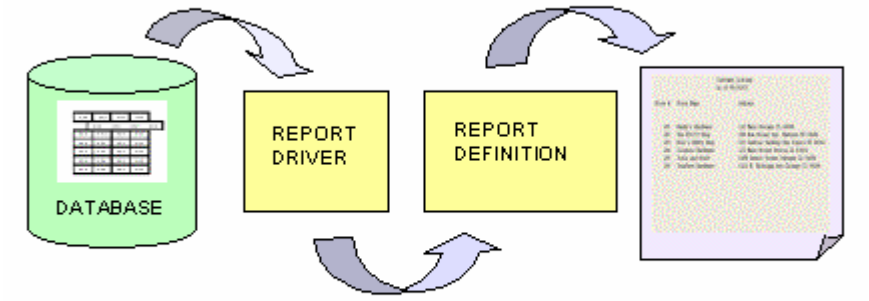
- Line 02 The index of the current record in the **cust_arr** program array is passed from the BEFORE DELETE control block of INPUT ARRAY, and stored in the variable **idx**. The BEFORE DELETE control block is executed immediately before the record is deleted from the program array, allowing the logic in this function to be executed before the record is removed from the program array.
- Line 05 sets the initial value of **del_ok** to FALSE.
- Lines 07 thru 15 display the modal Menu to the user for confirmation of the Delete.
- Lines 18 thru 22 use an embedded SQL DELETE statement to delete the row from the **customer** database table. The variable **idx** is used to determine the value of **store_num** in the program array record that is to be used as criteria in the DELETE statement. This record in the program array has not yet been removed, since this **delete_cust** function was called in a BEFORE DELETE control block. The DELETE is surrounded by WHENEVER ERROR statements.
- Lines 24 thru 30 test the SQLCA.SQLCODE to see if the update of the row in the database was successful, and return an appropriate message to the user. The value **del_ok** is set based on the success of the SQL DELETE statement.
- Line 33 returns the variable **del_ok** to the BEFORE DELETE control block, indicating whether the Delete of the **customer** row was successful.

Tutorial Chapter 9: Reports

Summary:

- Genero BDL Reports
- The Report Driver
- The Report Definition
  - DEFINE section
  - OUTPUT section
  - ORDER section
  - FORMAT section
- Two-Pass Reports
- Example: Customer Report
- Interrupting a Report
  - The interrupt Action View
  - Refreshing the Display
  - Using a ProgressBar
- Example: Interruption Handling

This program generates a simple report of the data in the **customer** database table. The two parts of a report, the report driver logic and the REPORT program block (report definition) are illustrated. Then the program is modified to display a window containing a Progressbar, and allowing the user to interrupt the report before it is finished.



BDL Reports

Genero BDL reports are easy to design and generate. The output from a report can be formatted so that the eye of the reader can easily pick out the important data.

The program logic that specifies what data to report (the report driver) is separate from the program logic that formats the output of the report (the report definition). This allows the report driver to supply data for multiple reports simultaneously, if desired. And, you can design template report definitions that might be used with report drivers that access different database tables.

The Report Driver

The part of a program that generates the rows of report data (also known as input records) is called the report driver. The primary concern of the row-producing logic is the selection of rows of data. The actions of a report driver are:

1. Use the START REPORT statement to initialize each report to be produced. We recommend that clauses regarding page setup and report destination be included in this statement.
2. Use a forward-only database cursor to read rows from a database, if that is the source of the report data.
3. Whenever a row of report data is available, use OUTPUT TO REPORT to send it to the report definition.
4. If an error is detected, use TERMINATE REPORT to stop the report process.
5. When the last row has been sent, use FINISH REPORT to end the report.

From the standpoint of the row-producing side, these are the only statements required to create a report.

The Report Definition

The report definition uses a REPORT program block to format the input records. REPORT is global in scope. It is not, however, a function; it is not reentrant, and CALL cannot invoke it.

The code within a REPORT program block consists of several sections, which must appear in the order shown:

The DEFINE section

Here you define the variables passed as parameter to the report, and the local variables. A report can have its own local variables for subtotals, calculated results, and other uses.

The OUTPUT section (optional)

Although you can define page setup and destination information in this section, the format of the report will be static. Providing this same information in the START REPORT statement provides more flexibility.

The ORDER BY section (optional)

Here you specify the required order for the data rows, when using groupping. Include this ORDER BY section if values that the report definition receives from the report driver are significant in determining how BEFORE GROUP OF or AFTER GROUP OF control blocks will process the data in the formatted report output. To avoid the creation of additional resourcesto sort the data, use the ORDER EXTERNAL statement in this section if the data to be used in the report has already been sorted by an ORDER BY clause in the SQL statement.

The FORMAT section

Here you describe what is to be done at a particular stage of report generation. The code blocks you write in the FORMAT section are the heart of the report program block and contain all its intelligence. You can use most BDL statements in the FORMAT section of a report; you cannot, however, include any SQL statements.

BDL invokes the sections and blocks within a report program block non-procedurally, at the proper time, as determined by the report data. You do not have to write code to calculate when a new page should start, nor do you have to write comparisons to detect when a group of rows has started or ended. All you have to write are the statements that are appropriate to the situation, and BDL supplies the “glue” to make them work.

You can write control blocks in the FORMAT section to be executed for the following events:

- Top (header) of the first page of the report (FIRST PAGE HEADER)
- Top (header) of every page after the first (PAGE HEADER)
- Bottom (footer) of every page (PAGE TRAILER)
- Each new row as it arrives (ON EVERY ROW)
- The start end of a group of rows (BEFORE GROUP OF) - a group is one or more rows having equal values in a particular column.
- The end of a group of rows (AFTER GROUP OF) - in this block, you typically print subtotals and other aggregate data for the group that is ending. You can use aggregate functions to calculate and display frequencies, percentages, sums, averages, minima, and maxima for this information.
- After the last row has been processed (ON LAST ROW) - aggregate functions calculated over all the rows of the report are typically printed here.

Two-pass reports

A two-pass report is one that creates temporary tables, therefore there must be an active connection to the database. The two-pass report handles sorts internally. During the first pass, the report engine sorts the data and stores the sorted values in a temporary file in the database. During the second pass, it calculates any aggregate values and produces output from data in the temporary files.

If your report definition includes any of the following, a two-pass report is required:

- An ORDER BY section without the EXTERNAL keyword.
- The GROUP PERCENT(*) aggregate function anywhere in the report.
- Any aggregate function outside the AFTER GROUP OF control block.

Warning: Some databases do not support temporary tables. Avoid a two-pass report for performance reasons and for portability.

Example: Customer Report

The Report Driver

Report Driver (custreport.4gl)

```
01 SCHEMA custdemo
02
03 MAIN
04  DEFINE pr_custrec RECORD
05   store_num  LIKE customer.store_num,
06   store_name LIKE customer.store_name,
07   addr       LIKE customer.addr,
08   addr2      LIKE customer.addr2,
09   city       LIKE customer.city,
10   state      LIKE customer.state,
11   zipcode    LIKE customer.zipcode
12  END RECORD
13
14  CONNECT TO "custdemo"
15
16  DECLARE custlist CURSOR FOR
17     SELECT store_num,
18            store_name,
19            addr,
20            addr2,
21            city,
22            state,
23            zipcode
24       FROM customer
25       ORDER BY state, city
26
27  START REPORT cust_list TO FILE "customers.txt"
```

```
28    WITH LEFT MARGIN = 5, TOP MARGIN = 2,
29          BOTTOM MARGIN = 2
30
31  FOREACH custlist INTO pr_custrec.*
32   OUTPUT TO REPORT cust_list(pr_custrec.*)
33  END FOREACH
34
35  FINISH REPORT cust_list
36
37  DISCONNECT CURRENT
38
39 END MAIN
```

Notes:

- Lines 04 thru 12 define a local program record **pr_custrec**, with a structure like the **customer** database table.
- Line 14 connects to the **custdemo** database.
- Lines 16 thru 25 define a **custlist** cursor to retrieve the **customer** table data rows, sorted by state, then city.
- Lines 27 thru 29 starts the REPORT program block named **cust_list**, and includes a report destination and page formatting information.
- Lines 31 thru 33 retrieve the data rows one by one into the program record **pr_custrec** and pass the record to the REPORT program block.
- Line 35 closes the report driver and executes any final REPORT control blocks to finish the report.
- Line 37 disconnects from the **custdemo** database.

The Report Definition

Report definition (custreport.4gl

```
01 REPORT cust_list(r_custrec)
02  DEFINE r_custrec RECORD
03         store_num  LIKE customer.store_num,
04         store_name LIKE customer.store_name,
05         addr       LIKE customer.addr,
06         addr2      LIKE customer.addr2,
07         city       LIKE customer.city,
08         state      LIKE customer.state,
09         zipcode    LIKE customer.zipcode
10        END RECORD
11
12  ORDER EXTERNAL BY r_custrec.state, r_custrec.city
13
14  FORMAT
15
16   PAGE HEADER
17     SKIP 2 LINES
18     PRINT COLUMN 30, "Customer Listing"
19     PRINT COLUMN 30, "As of ", TODAY USING "mm/dd/yy"
```

```
20      SKIP 2 LINES
21
22      PRINT  COLUMN 2, "Store #",
23             COLUMN 12, "Store Name",
24             COLUMN 40, "Address"
25
26      SKIP 2 LINES
27
28    ON EVERY ROW
29      PRINT COLUMN 5, r_custrec.store_num USING "####",
30            COLUMN 12, r_custrec.store_name CLIPPED,
31            COLUMN 40, r_custrec.addr CLIPPED;
32
33      IF r_custrec.addr2 IS NOT NULL THEN
34        PRINT 1 SPACE, r_custrec.addr2 CLIPPED, 1 space;
35      ELSE
36         PRINT 1 SPACE;
37      END IF
38
39      PRINT r_custrec.city CLIPPED, 1 SPACE,
40            r_custrec.state, 1 SPACE,
41            r_custrec.zipcode CLIPPED
42
43    BEFORE GROUP OF r_custrec.city
44      SKIP TO TOP OF PAGE
45
46    ON LAST ROW
47      SKIP 1 LINE
48      PRINT "TOTAL number of customers: ",
49              COUNT(*) USING "#,###"
50
51    PAGE TRAILER
52      SKIP 2 LINES
53      PRINT COLUMN 30, "-", PAGENO USING "<<", " -"
54
55  END REPORT
```

Notes:

- Line 01 The REPORT control block has the **pr_custrec** record passed as an argument.
- Lines 02 thru 10 define a local program record **r_custrec** to store the values that the calling routine passes to the report.
- Line 12 tells the REPORT control block that the records will be passed sorted in order by state, then city. The ORDER EXTERNAL syntax is used to prevent a second sorting of the program records, since they have already been sorted by the SQL statement in the report driver.
- Line 14 is the beginning of the FORMAT section.
- Lines 16 thru 20 The PAGE HEADER block specifies the layout generated at the top of each page.
  Each PRINT statement starts a new line containing text or a value. The PRINT statement can have multiple COLUMN clauses, which all print on the same line. The COLUMN clause specifies the offset of the first character from the first position after the left margin. The values to be printed can be program variables,

static text, or built-in functions.
The built-in TODAY operator generates the current date; the USING clauses formats this.
The SKIP statement inserts empty lines.
The PAGE HEADER for this report will appear as follows:

```
<skipped line>
<skipped line>
          Customer Listing
          As of <date>
<skipped line>
<skipped line>
Store #    Store Name          Address
<skipped line>
<skipped line>
```

- Lines 28 thru 41 specifies the layout generated for each row. The data can be read more easily if each program record passed to the report is printed on a single row. Although there are four PRINT statements in this control block, the first three PRINT statements are terminated by semi-colons. This suppresses the new line signal, resulting in just a single row of printing.
  The CLIPPED keyword eliminates any trailing blanks after the name, addresses, and city values.
  Any IF statement that is included in the FORMAT section must contain the same number of PRINT/SKIP statements regardless of which condition is met. Therefore, if **r_custrec.addr2** is not NULL, a PRINT statement prints the value followed by a single space; if it is NULL, a PRINT statement prints a single space. As mentioned earlier, each PRINT statement is followed by a semicolon to suppress the new-line.
  The output for each row will be as follows:

```
    106 TrueTest Hardware     6123 N. Michigan Ave Chicago IL
60104
    101 Bandy's Hardware      110 Main Chicago IL 60068
```

- Lines 43 and 44 start a new page for each group containing the same value for **r_custrec.city**.
- Lines 46 thru 49 specify a control block to be executed after the statements in ON EVERY ROW and AFTER GROUP OF control block. This prints at the end of the report. The aggregate function COUNT(*) is used to print the total number of records passed to the report. The USING keyword formats the number. This appears as follows:

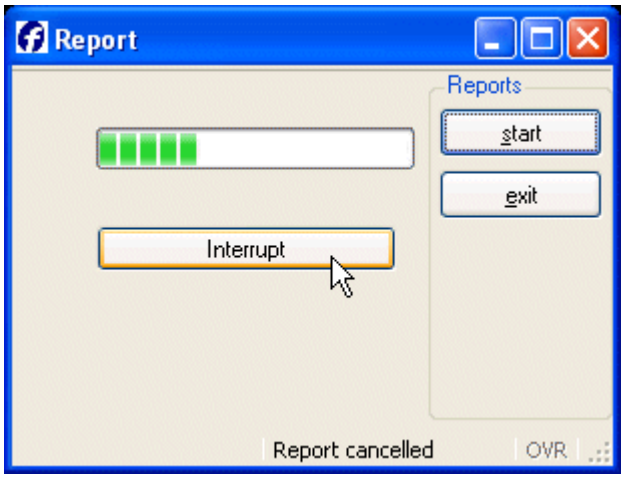
```
    <skipped line>
    Total number of customers:   <count>
```

- Lines 51 thru 53 specifies the layout generated at the bottom of each page. The built-in function PAGENO is used to print the page number. The USING keyword formats the number, left-justified.
  This appears as follows:

```
        <skipped line>
        <skipped line>
                        - <pageno> -
```

Interrupting a Report

When a program performs a long process like a loop, a report, or a database query, the lack of user interaction statements within the process can prevent the user from interrupting it. In this program, the preceding example is modified to display a form containing start, exit, and interrupt buttons, as well as a progress bar showing how close the report is to completion.



The interrupt action view

In order to allow a user to stop a long-running report, for example, you can define an action view with the name "interrupt". When the runtime system takes control of the program, the client automatically enables a local interrupt action to let the user send an asynchronous request to the program. This interruption request is interpreted by the runtime system as a traditional interruption signal, as if it was generated on the server side, and the INT_FLAG variable is set to TRUE.

Refreshing the Display

The Abstract User Interface tree on the front end is synchronized with the runtime system AUI tree when a user interaction instruction takes the control. This means that the user will not see any display as long as the program is doing batch processing, until an interactive statement is reached. If you want to show something on the screen while the program is running in a batch procedure, you must force synchronization with the front end.

The Interface class is a built-in class provided to manipulate the user interface. The **refresh()** method of this class synchronizes the front end with the current AUI tree. You do not need to instantiate this class before calling any of its methods:

```
CALL ui.Interface.refresh()
```

Using a ProgressBar

One of the form item types is a PROGRESSBAR, a horizontal line with a progress indicator. The position of the PROGRESSBAR is defined by the value of the corresponding form field. The value can be changed from within a BDL program by using the DISPLAY instruction to set the value of the field.

This type of form item does not allow data entry; it is only used to display integer values. The VALUEMIN and VALUEMAX attributes of the PROGRESSBAR define the lower and upper integer limit of the progress information. Any value outside this range will not be displayed.

Example: Interruption Handling

The Form Specification File

A form containing a progress bar is defined in the form specification file **reportprog.per**.

Form (reportprog.per)

```
01 LAYOUT (TEXT="Report")
02  GRID
03  {
04
05        [f001                 ]
06
07        [ib                   ]
08
09
10  }
11  END
12 END
13
14 ATTRIBUTES
15 PROGRESSBAR f001 = formonly.rptbar, VALUEMIN=1,VALUEMAX=10;
16 BUTTON ib : interrupt, TEXT="Stop";
17 END
```

Notes:

- Line 05 contains the form field for the PROGRESSBAR.
- Line 07 contains the form field for the interrupt action view.

- Line 15 defines the PROGRESSBAR as formonly since its type is not derived from a database column. The values range from 1 to 10. The maximum value for the PROGRESSBAR was chosen arbitrarily, and was set rather low since there aren't many rows in the **customer** database table.
- Line 16 defines the button **ib** as an interrupt action view with TEXT of "Stop".

Modifications to custreports.4gl

The MAIN program block has been modified to open a window containing the form with a PROGRESSBAR and a MENU, to allow the user to start the report and to exit. A new function, **cust_report**, is added for interruption handling. The report definition, the **cust_list** REPORT block, remains the same as in the previous example.

Changes to the MAIN program block (custreport2.4gl)

```
01 MAIN
02
03   DEFER INTERRUPT
04   CONNECT TO "custdemo"
05   CLOSE WINDOW SCREEN
06   OPEN WINDOW w3 WITH FORM "reportprog"
07
08   MENU "Reports"
09   ON ACTION start
10      MESSAGE "Report starting"
11       CALL cust_report()
12   ON ACTION exit
13       EXIT MENU
14   END MENU
15
16   CLOSE WINDOW w3
17   DISCONNECT CURRENT
18
19 END MAIN
```

Notes:

- Line 03 prevents the user from interrupting the program except by using the interrupt action view.
- Line 06 Opens the window and form containing the PROGRESSBAR.
- Lines 08 thru 14 define a MENU with two actions:
  start - displays a MESSAGE and calls the function **cust_report**.
  exit - quits the MENU.

The cust_report function

This new function contains the report driver, together with the logic to determine whether the user has attempted to interrupt the report.

Function cust_report (custreport2.4gl)

```
21 FUNCTION cust_report()
22
23 DEFINE pr_custrec RECORD
24       store_num   LIKE customer.store_num,
25       store_name  LIKE customer.store_name,
26       addr        LIKE customer.addr,
27       addr2       LIKE customer.addr2,
28        city        LIKE customer.city,
29       state       LIKE customer.state,
30       zipcode     LIKE customer.zipcode
31       END RECORD,
32       rec_count, rec_total,
33       pbar, break_num  INTEGER
34
35   LET rec_count = 0
36   LET rec_total = 0
37   LET pbar = 0
38   LET break_num = 0
39   LET INT_FLAG = FALSE
40
41   SELECT COUNT(*) INTO rec_total FROM customer
42
43   LET break_num = (rec_total/10)
44
45   DECLARE custlist CURSOR FOR
46     SELECT store_num,
47        store_name,
48        addr,
49        addr2,
50        city,
51        state,
52        zipcode
53      FROM CUSTOMER
54      ORDER BY state, city
55
56   START REPORT cust_list TO FILE "customers.txt"
57   FOREACH custlist INTO pr_custrec.*
58     OUTPUT TO REPORT cust_list(lr_custrec.*)
59     LET rec_count = rec_count+1
60     IF (rec_count MOD break_num)= 0 THEN
61       LET pbar = pbar+1
62       DISPLAY pbar TO rptbar
63       CALL ui.Interface.refresh()
64       IF (INT_FLAG) THEN
65         EXIT FOREACH
66       END IF
67     END IF
68   END FOREACH
69
70   IF (INT_FLAG) THEN
71     LET INT_FLAG = FALSE
72     MESSAGE "Report cancelled"
73   ELSE
74     FINISH REPORT cust_list
75     MESSAGE "Report finished"
```

```
76   END IF
77
78 END FUNCTION
```

Notes:

- Lines 23 thru 31 now define the **pr_custrec** record in this function.
- Lines 32 thru 33 define some additional variables.
- Lines 35 thru 39 initialize the local variables.
- Line 38 sets INT_FLAG to FALSE.
- Line 41 uses an embedded SQL statement to retrieve the count of the rows in the **customer** table and stores it in the variable **rec_total**.
- Line 43 calculates the value of **break_num** based on the maximum value of the PROGRESSBAR, which is set at 10. After **break_num** rows have been processed, the program will increment the PROGRESSBAR. The front end cannot handle interruption requests properly if the display generates a lot of network traffic, so we do not recommend refreshing the AUI and checking INT_FLAG after every row.
- Lines 45 thru 54 declare the **custlist** cursor for the customer table.
- Line 56 starts the report, sending the output to the file **custout**.
- Lines 58 thru 68 contain the FOREACH statement to output each record to the same report **cust_list** used in the previous example.
- Line 59 increments **rec_count** to keep track of how many records have been output to the report.
- Line 60 tests whether a break point has been reached, using the MOD (Modulus) function.
- Line 61 If a break point has been reached, the value of **pbar** is incremented.
- Line 62 The **pbar** value is displayed to the **rptbar** PROGRESSBAR form field.
- Line 63 The front end is synced with the current AUI tree.
- Line 64 thru 66 The value of INT_FLAG is checked to see whether the user has interrupted the program. If so, the FOREACH loop is exited prematurely.
- Lines 70 thru 76 test INT_FLAG again and display a message indicating whether the report finished or was interrupted. If the user did not interrupt the report, the FINISH REPORT statement is executed.

Tutorial Chapter 10: Localization

Summary:

- Localization Support
- Localized Strings
- Programming Steps
- Strings in Sources
- Generating Source String Files
- Compiling Source String Files
- Deploying Compiled String Files
- Example

Localization Support

Localization Support is a feature of the language that allows you to write application supporting multi-byte character sets as well as date, numeric and currency formatting in accordance with a locale.

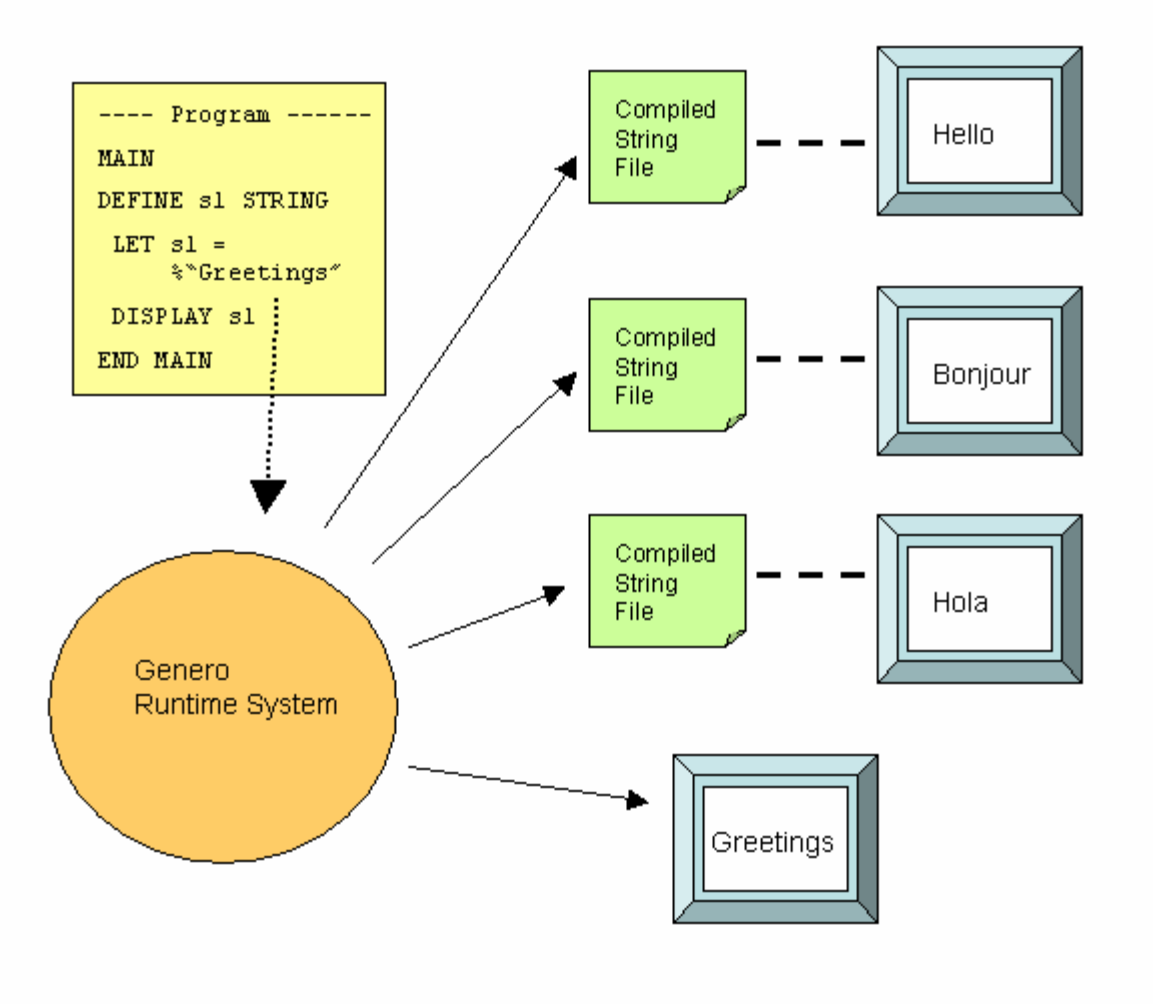
Localization Support is based on the system libraries handling the locale, a set of language and cultural rules.

See Localization for more details.

Localized Strings

Localized Strings allow you to internationalize your application using different languages, and to customize it for specific industry markets in your user population. Any string that is used in your Genero BDL program, such as messages to be displayed or the text on a form, can be defined as a Localized String. At runtime, the Localized String is replaced with text stored in a String File.

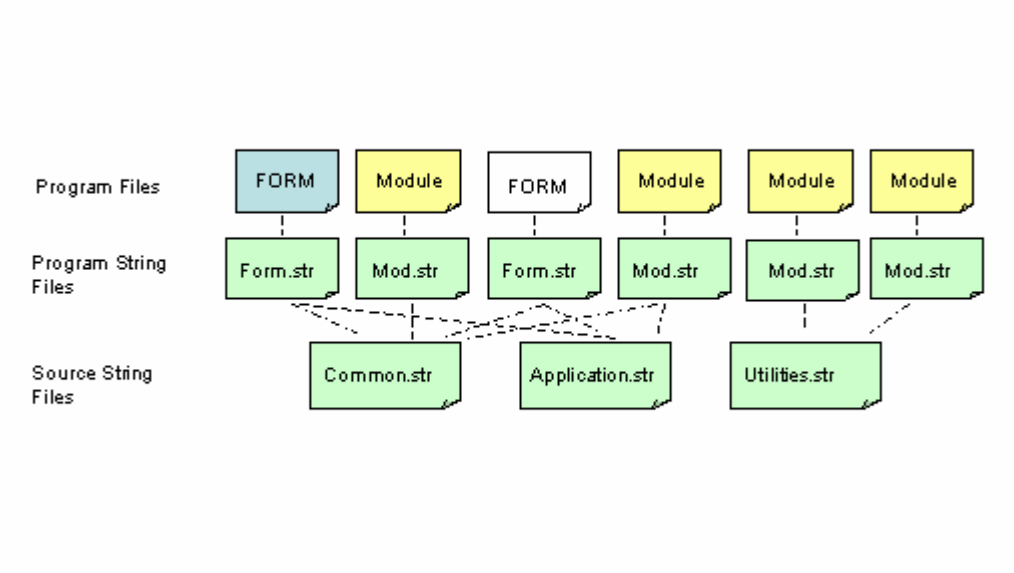
String Files must be compiled, and then deployed at the user site.



Programming Steps

The following steps describe how to use localized strings in your sources:

1. Modify your form specification files and program module files to contain Localized Strings by inserting the % sign in front of the strings that you wish to be replaced.
2. Use the **-m** option of **fglform** to extract the Localized Strings from each form specification file into a separate Source String File (extension **.str**).
3. Use the -**m** option of **fglcomp** to extract the Localized Strings from each program module into a separate Source String File (extension **.str**).
4. Concatenate the Source String Files together logically; for example, you may have a **common.str** file containing the strings common to all applications, a **utility.str** file containing the strings common to utilities, and an **application.str** file with the strings specific to the particular application.



5. At this point the names of the Localized Strings may be unwieldy, since they were derived from the actual strings in the program files. You can modify the string names in your Source String Files and the corresponding program files so they form keys that are logical. For example:

```
$"common.accept" = "OK"
$"common.cancel"= "Cancel"
$"common.quit" = "Quit"
```

6. Make the Source String Files available to the programming teams for use as a reference when creating or modifying programs.
7. Copy the Source String Files, and modify the replacement text for each of your market segments or user languages.
8. Compile the Source String Files (**.42s**).
9. Create the entries in FGLPROFILE to specify what string files must be used at runtime.
10. Deploy **.42s** compiled string files to user sites.

Strings in Sources

A Localized String begins with a percent sign (%), followed by the name of the string identifying the replacement text to be loaded from the Compiled String File. Since the name is a string, you can use any characters in the name, including blanks.

```
LET s1 = %"Greetings"
```

The string "Greetings" is both the name of the string and the default text which would be used if no string resource files are provided at runtime.

Localized Strings can be used any place where a STRING literal can be used, including form specification files.

The SFMT() and LSTR() operators can be used to manipulate the contents of Localized Strings. For example, the program line:

```
DISPLAY SFMT( %"cust.valid", custnum )
```

reads from the associated Compiled String File:

```
"cust.valid"="customer %1 is valid"
```

resulting in the following display when the value of custnum is 200:

```
"customer 200 is valid"
```

Extracting Strings

You can generate a Source String File by extracting all of the Localized Strings from your program module or form specification file, using the **-m** option of fglcomp or fglform:

```
fglcomp -m mystring.4gl > mystring.str
```

The generated file would have the format:

```
"Greetings" = "Greetings"
```

You could then change the replacement text in the file:

```
"Greetings" = "Hello"
```

The source string file must have the extension **.str**.

Compiling Source String Files (fglmkstr)

Source String Files must be compiled to binary files in order to be used at runtime:

```
fglmkstr mystring.str
```

The resulting Compiled String File has the extension **.42s** (**mystring.42s**).

Deploying String Files

The Compiled String Files, produced by the fglmkstr tool, must be deployed on the production sites. The file extension is **.42s**.

By default, the runtime system searches for a **.42s** file with the same name prefix as the current **.42r** program.

You can specify a list of string files with entries in the FGLPROFILE configuration file:

```
fglrun.localization.file.count = 2
fglrun.localization.file.1.name = "firstfile"
fglrun.localization.file.2.name = "secondfile"
```

The current directory and the path defined in the DBPATH environment variable, are searched for the **.42s** Compiled String File.

Tip:

1. Create several string files with the same names, but locate them in different directories. You can then easily switch from one set of string files to another, just by changing the DBPATH environment variable. You typically create one string file directory per language, and if needed, you can create subdirectories for each code-set (strings/english/iso8859-1, strings/french/windows1252).

Example:

form.per - the form specification file

This form specification file uses the LABEL form item type to display the text associated with the form fields containing data from the **customer** database table. LABEL item types contain read-only values.

form.per

```
01 SCHEMA custdemo
02
03 LAYOUT
04   GRID
05   {
06    [lab1      ] [f01  ]
07
08    [lab2      ] [f02                ]
09
10    [lab3      ] [f03            ]
11   }
12   END   --grid
13 END  -- layout
```

```
14
15 TABLES customer
16
17 ATTRIBUTES
18 LABEL lab1 : TEXT=%"customer.store_num";
19 EDIT  f01  = customer.store_num,
20             COMMENT=%"customer.dealermsg";
21 LABEL lab2 : TEXT=%"customer.store_name";
22 EDIT  f02  = customer.store_name;
23 LABEL lab3 : TEXT=%"customer.city";
24 EDIT  f03  = customer.city;
25 END  -- attributes
```

Notes:

- Lines 06 and 18: The form contains a LABEL, **lab1**; the TEXT of the LABEL is a Localized String, **customer.store_num**.
- Line 20: The COMMENT of the EDIT **f01** is a Localized String, **customer.dealermsg**.
- Lines 08 and 21: The TEXT of the LABEL **lab2** is a Localized String, **customer.store_name**.
- Lines 10 and 23: The TEXT of the LABEL **lab3** is a Localized String, **customer.city**.

These strings will be replaced at runtime.

form.str - the String File associated with this form

After translation, the string source file would look like this:

```
01 "customer.store_num"="Store No"
02 "customer.dealernummsg"="This is the dealer number"
03 "customer.store_name"="Store Name"
04 "customer.city"="City"
```

prog.4gl - the program module

This program module opens the form containing Localized Strings:

Module prog.4gl

```
01 SCHEMA custdemo
02
03 MAIN
04  CONNECT TO "custdemo"
05  CLOSE WINDOW SCREEN
06  OPEN WINDOW w1 WITH FORM "stringform"
07  MESSAGE %"customer.msg"
08  MENU %"customer.menu"
09     ON ACTION query
10       CALL query_cust()
11     ON ACTION exit
```

```
12       EXIT MENU
13  END MENU
14  CLOSE WINDOW w1
15  DISCONNECT CURRENT
16 END MAIN
17
18 FUNCTION query_cust() -- displays one row
19   DEFINE l_custrec RECORD
20          store_num LIKE customer.store_num,
21          store_name LIKE customer.store_name,
22          city LIKE customer.city
23   END RECORD,
24   msg STRING
25
26   WHENEVER ERROR CONTINUE
27   SELECT store_num, store_name, city
28     INTO l_custrec.*
29     FROM customer
30     WHERE store_num = 101
31   WHENEVER ERROR STOP
32
33   IF SQLCA.SQLCODE = 0 THEN
34      LET msg = SFMT( %"customer.valid",
35                      l_custrec.store_num )
36      MESSAGE msg
37      DISPAY BY NAME l_custrec.*
38   ELSE
39      MESSAGE %"customer.notfound"
40   END IF
41
42 END FUNCTION
```

Notes:

- Lines 07, 08, 34 and 39 contain Localized Strings for the messages that the program displays.

These strings will be replaced at runtime.

prog.str - the String File associated with this program module

After translation, the string source file would look like this:

```
01 "customer.msg"="Displays a Dealer"
02 "customer.menu"="Dealer"
03 "customer.valid"="Customer %1 is valid"
04 "customer.notfound"="Customer was not found"
```

Compiling the program

The program is compiled into **cust.42r.**

```
fgl2p -o cust.42r prog.4gl
```

Compiling the string files

Both string files must be compiled:

```
fglmkstr form.str
fglmkstr prog.str
```

The resulting Compiled String Files are **form.42s** and **prog.42s**.

Setting the list of compiled string files in FGLPROFILE

The list of Compiled String Files is specified in the FGLPROFILE configuration file. The runtime system searches for a file with the "42s" extension in the current directory and in the path list defined in the DBPATH environment variable. Specify the total number of files, and list each file with an index number.

Example file fglprofile:

```
01 fglrun.localization.file.count = 2
02 fglrun.localization.file.1.name = "form"
03 fglrun.localization.file.2.name = "prog"
```

Setting the environment

Set the FGLPROFILE environment variable:

```
export FGLPROFILE=./fglprofile
```

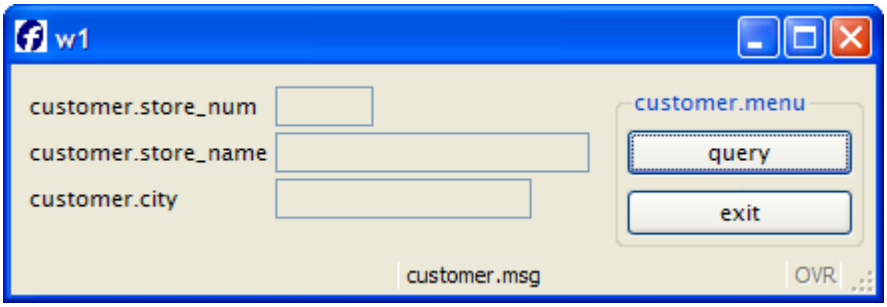
Running the program

Run the program:

```
fglrun cust
```

The Resulting Form Display

Display of the form using the default values for the strings:



Display of the form when the Compiled String File is deployed:

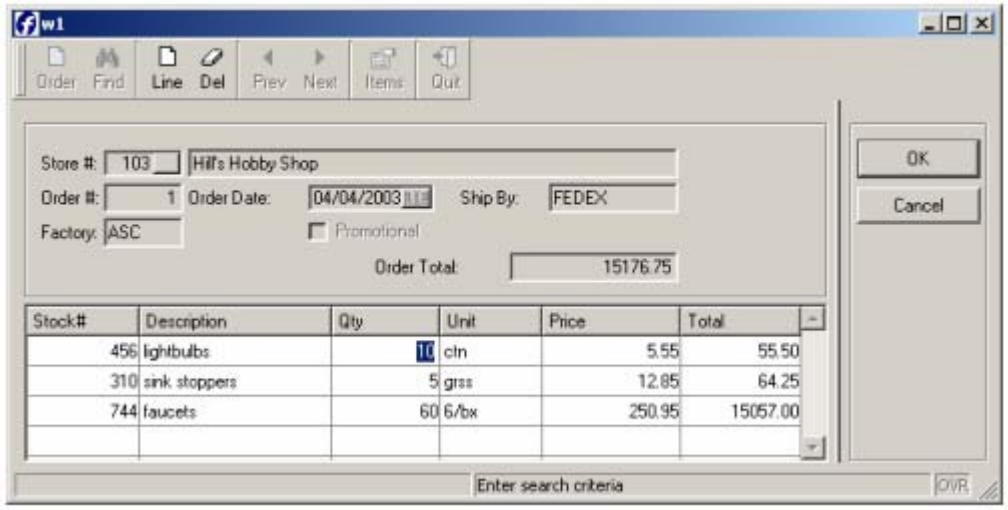
Tutorial Chapter 11: Master/Detail

Summary:

- The Master-Detail sample
- The Makefile
- The Customer List module
- The Stock List mode
- The Master-Detail Form Specification File
- The Orders Program
  - The MAIN program block
  - Function setup_actions
  - Function order_new
  - Function order_insert
  - Function order_query
  - Function order_fetch
  - Function order_select
  - Function order_fetch_rel
  - Function order_total
  - Function order_close
  - Function items_fetch
  - Function items_show
  - Function items_inpupd
  - Function items_line_total
  - Function item_insert
  - Function item_update
  - Function item_delete
  - Function get_stock_info

The Master-Detail sample

The example discussed in this chapter is designed for the input of order information (headers and order lines), illustrating a typical master-detail relationship. The form used by the example contains fields from both the **orders** and **items** tables in the **custdemo** database.



Display on Windows platforms

Since there are multiple items associated with a single order, the rows from the **items** table are stored in a program array and displayed in a table container on the form. Most of the functionality to query/add/update/delete has been covered in previous chapters; this chapter will focus on the master/detail form and the unique features of the corresponding program.

Note: This type of relationship could also be handled using Multiple Dialogs, allowing the interactive statements to operate in parallel. See Multiple Dialogs for additional information. There are extensive sample programs using Multiple Dialogs in the **<*Genero-BDL-install-directory*>/demo/MultipleDialogs** directory.

The Makefile

The BDL modules and forms used by the application in this chapter are compiled/linked by using a **Makefile**. This file is interpreted by the '**make**' utility, which is a well-known tool to build large programs based on multiple sources and forms.

The **make** utility reads the dependency rules defined in the **Makefile** for each program component, and executes the commands associated with the rules.

This section only describes the **Makefile** used in this example. For more details about Makefiles, see the **make** utility documentation.

Makefile

```
01  all:: orders
02
03  orders.42m: orders.4gl
```

```
04          fglcomp -M orders.4gl
05
06  orderform.42f: orderform.per
07          fglform -M orderform.per
08
09  custlist.42m: custlist.4gl
10          fglcomp -M custlist.4gl
11
12  custlist.42f: custlist.per
13          fglform -M custlist.per
14
15  stocklist.42m: stocklist.4gl
16          fglcomp -M stocklist.4gl
17
18  stocklist.42f: stocklist.per
19          fglform -M stocklist.per
20
21  MODULES=\
22   orders.42m\
23   custlist.42m\
24   stocklist.42m
25
26  FORMS=\
27   orderform.42f\
28   custlist.42f\
29   stocklist.42f
30
31  orders:: $(MODULES) $(FORMS)
32        fgllink -o orders.42r $(MODULES)
33
34  run::
35        fglrun orders
36
37  clean::
38          rm -f *.42?
```

Notes:

- Line 01 defines the '**all**' dependency rule that will be executed by default, and depends from the rule '**orders**' described on line 31. You execute this rule with '**make all**', or '**make**' since this is the first rule in the Makefile.
- Lines 03 and 04 define a dependency to compile the **orders.4gl** module into **orders.42m**. The file on the left (**orders.42m**) depends from the file on the right (**orders.4gl**), and the command to be executed is **fglcomp -M orders.4gl**.
- Lines 06 and 07 define a dependency to compile the **orderform.per** form.
- Lines 09 and 10 define a dependency to compile the **custlist.4gl** module.
- Lines 12 and 13 define a dependency to compile the **custlist.per** form.
- Lines 15 and 16 define a dependency to compile the **stocklist.4gl** module.
- Lines 18 and 19 define a dependency to compile the **stocklist.per** form.
- Lines 21 thru 24 define the list of compiled modules, used in the global '**orders**' dependency rule.
- Lines 26 thru 29 define the list of compiled form files, used in the global '**orders**' dependency rule.

- Lines 31 and 32 is the global 'orders' dependency rule, defining modules or form files to be created.
- Lines 34 and 35 define a rule and command to execute the program. You execute this rule with '**make run**'.
- Lines 37 and 38 define a rule and command to clean the directory. You execute this rule with '**make clean**'.

The Customer List Module

The **custlist.4gl** module defines a 'zoom' module, to let the user select a customer from a list. The module could be re-used for any application that requires the user to select a customer from a list.

This module uses the **custlist.per** form and is typical list handling using the DISPLAY ARRAY statement, as discussed in Chapter 07. The **display_custlist()** function in this module returns the customer id and the name. See the **custlist.4gl** source module for more details.

In the application illustrated in this chapter, the main module **orders.4gl** will call the **display_custlist()** function to retrieve a customer selected by the user.

```
01   ON ACTION zoom1
02      CALL display_custlist() RETURNING id, name
03      IF (id > 0) THEN
04         ...
```

The Stock List Module

The **stocklist.4gl** module defines a 'zoom' module, to let the user select a stock item from a list. This module uses the **stocklist.per** form and is typical list handling using the DISPLAY ARRAY statement, as discussed in Chapter 07. See the **stocklist.4gl** source module for more details.

The main module **orders.4gl** will call the **display_stocklist()** function of the **stocklist.4gl** module to retrieve a stock item selected by the user.

The function returns the stock item id only:

```
01   ON ACTION zoom2
02      LET id = display_stocklist()
03      IF (id > 0) THEN
04         ...
```

The Master-Detail Form Specification File

The form specification file **orderform.per** defines a form for the orders program, and displays fields containing the values of a single order from the **orders** table. The name of the store is retrieved from the **customer** table, using the column **store_num,** and displayed.

A screen array displays the associated rows from the **items** table. Although **order_num** is also one of the fields in the **items** table, it does not have to be included in the screen array or in the screen record, since the order number will be the same for all the items displayed for a given order. For each item displayed in the screen array, the values in the **description** and **unit** columns from the **stock** table are also displayed.

The values in FORMONLY fields are not retrieved from a database; they are calculated by the BDL program based on the entries in other fields. In this form FORMONLY fields are used to display the calculations made by the BDL program for item line totals and the order total.

This form uses some of the attributes that can be assigned to fields in a form. See Form Specification Files Attributes for a complete list of the available attributes.

Form orderform.per

```
01 SCHEMA custdemo
02
03 TOOLBAR
04  ITEM new (TEXT="Order", IMAGE="new", COMMENT="New order")
05  ITEM find (TEXT="Find", IMAGE="find")
06  SEPARATOR
07  ITEM append (TEXT="Line", IMAGE="new", COMMENT="New order line")
08  ITEM delete (TEXT="Del", IMAGE="eraser")
09  SEPARATOR
10  ITEM previous (TEXT="Prev")
11  ITEM next (TEXT="Next")
12  SEPARATOR
13  ITEM getitems (TEXT="Items", IMAGE="prop")
14  SEPARATOR
15  ITEM quit (TEXT="Quit", COMMENT="Exit the program", IMAGE="quit")
16 END
17
18 LAYOUT
19 VBOX
20 GROUP
21 GRID
22 {
23   Store #:[f01  ] [f02                                          ]
24   Order #:[f03  ]  Order Date:[f04         ] Ship By:[f06       ]
25   Factory:[f05  ]             [f07                              ]
26                                    Order Total:[f14           ]
27 }
28 END
29 END -- GROUP
30 TABLE
```

```
{
 Stock#  Description       Qty     Unit    Price       Total
[f08    |f09              |f10    |f11    |f12        |f13      ]
[f08    |f09              |f10    |f11    |f12        |f13      ]
[f08    |f09              |f10    |f11    |f12        |f13      ]
[f08    |f09              |f10    |f11    |f12        |f13      ]
}
END
END
END

TABLES
  customer, orders, items, stock
END

ATTRIBUTES
 BUTTONEDIT f01 = orders.store_num, REQUIRED, ACTION=zoom1;
 EDIT       f02 = customer.store_name, NOENTRY;
 EDIT       f03 = orders.order_num, NOENTRY;
 DATEEDIT   f04 = orders.order_date;
 EDIT       f05 = orders.fac_code, UPSHIFT;
 EDIT       f06 = orders.ship_instr;
 CHECKBOX   f07 = orders.promo, TEXT="Promotional",
                  VALUEUNCHECKED="N", VALUECHECKED="Y";
 BUTTONEDIT f08 = items.stock_num, REQUIRED, ACTION=zoom2;
 LABEL      f09 = stock.description;
 EDIT       f10 = items.quantity, REQUIRED;
 LABEL      f11 = stock.unit;
 LABEL      f12 = items.price;
 LABEL      f13 = formonly.line_total TYPE DECIMAL(9,2);
 EDIT       f14 = formonly.order_total TYPE DECIMAL(9,2), NOENTRY;
END

INSTRUCTIONS
SCREEN RECORD sa_items(
  items.stock_num,
  stock.description,
  items.quantity,
  stock.unit,
  items.price,
  line_total
)
END
```

Notes:

- Lines 03 thru 16 define a TOOLBAR section with typical actions.
- Lines 23 and 48 The field **f02** is a LABEL, allowing no editing. It displays the customer name associated with the **orders** store number
- Lines 19 and 49 Field **f03** is the order number from the **orders** table.
- Lines 25 and 53 The field **f07** is a CHECKBOX displaying the values of the column **promo** in the **orders** table. The box will appear checked if the value in the column is "Y", and unchecked if the value is "N".

- Lines 26 and 61 The field **f14** is a FORMONLY field This field displays the order total calculated by the BDL program logic.
- Lines 30 thru 38 describe the TABLE container for the screen array.
- Lines 33, 56 and 58 The fields **f09** and **f11** are LABELS, and display the description and unit of measure for the **items** stock number.
- Lines 33 and 60 the field **f13** is a LABEL and FORMONLY. This field displays the line total calculated for each line in the screen array.
- Lines 42 thru 44 The TABLES statement includes all the database tables that are listed for fields in the Attributes section of the form.
- Line 47 The attribute REQUIRED forces the user to enter data in the field during an INPUT statement.
- Line 51 The attribute UPSHIFT makes the runtime system convert lowercase letters to uppercase letters, both on the screen display and in the program variable that stores the contents of this field.
- Line 65 The screen record includes the names of all the fields shown in the screen array.

The Orders Program orders.4gl

Much of the functionality is identical to that in earlier programs. The query/add/delete/update of the orders table would be the same as the examples in Chapter 4 and Chapter 6 . Only append and query are included in this program, for simplicity. The add/delete/update of the **items** table is similar to that in Chapter 8. The complete orders program is outlined below, with examples of any new functionality.

The MAIN program block

This program block contains the menu for the Orders program.

MAIN program block (orders.4gl)

```
01 SCHEMA custdemo
02
03 DEFINE  order_rec RECORD
04            store_num    LIKE orders.store_num,
05            store_name   LIKE customer.store_name,
06            order_num    LIKE orders.order_num,
07            order_date   LIKE orders.order_date,
08            fac_code     LIKE orders.fac_code,
09            ship_instr   LIKE orders.ship_instr,
10            promo        LIKE orders.promo
11         END RECORD,
12         arr_items DYNAMIC ARRAY OF RECORD
13            stock_num    LIKE items.stock_num,
14            description  LIKE stock.description,
15            quantity     LIKE items.quantity,
16            unit         LIKE stock.unit,
```

```
17            price        LIKE items.price,
18            line_total   DECIMAL(9,2)
19         END RECORD
20
21 CONSTANT msg01 = "You must query first"
22 CONSTANT msg02 = "Enter search criteria"
23 CONSTANT msg03 = "Canceled by user"
24 CONSTANT msg04 = "No rows in table"
25 CONSTANT msg05 = "End of list"
26 CONSTANT msg06 = "Beginning of list"
27 CONSTANT msg07 = "Invalid stock number"
28 CONSTANT msg08 = "Row added"
29 CONSTANT msg09 = "Row updated"
30 CONSTANT msg10 = "Row deleted"
31 CONSTANT msg11 = "Enter order"
32 CONSTANT msg12 = "This customer does not exist"
33 CONSTANT msg13 = "Quantity must be greater than zero"
34
35 MAIN
36   DEFINE has_order, query_ok SMALLINT
37   DEFER INTERRUPT
38
39   CONNECT TO "custdemo"
40   CLOSE WINDOW SCREEN
41
42   OPEN WINDOW w1 WITH FORM "orderform"
43
44   MENU
45     BEFORE MENU
46        CALL setup_actions(DIALOG,FALSE,FALSE)
47     ON ACTION new
48        CLEAR FORM
49        LET query_ok = FALSE
50        CALL close_order()
51        LET has_order = order_new()
52        IF has_order THEN
53           CALL arr_items.clear()
54           CALL items_inpupd()
55        END IF
56        CALL setup_actions(DIALOG,has_order,query_ok)
57     ON ACTION find
58        CLEAR FORM
59        LET query_ok = order_query()
60        LET has_order = query_ok
61        CALL setup_actions(DIALOG,has_order,query_ok)
62     ON ACTION next
63        CALL order_fetch_rel(1)
64     ON ACTION previous
65        CALL order_fetch_rel(-1)
66     ON ACTION getitems
67        CALL items_inpupd()
68     ON ACTION quit
69        EXIT MENU
70   END MENU
71
72   CLOSE WINDOW w1
```

```
73
74 END MAIN
```

Notes:

- Lines 03 thru 11 define a record with fields for all the columns in the **orders** table, as well as **store_name** from the **customer** table.
- Lines 12 through 19 define a dynamic array with fields for all the columns in the **items** table, as well as **quantity** and **unit** from the **stock** table, and a calculated field **line_total**.
- Lines 21 thru 33 define constants to hold the program messages. This centralizes the definition of the messages, which can be used in any function in the module.
- Lines 44 thru 65 define the main menu of the application.
- Line 46 is executed before the menu is displayed; it calls the **setup_actions** function to disable navigation and item management actions by default. The DIALOG predefined object is passed as the first parameter to the function.
- Lines 47 thru 56 perform the 'add' action to create a new order. The **order_new** function is called, and if it returns TRUE, the **items_inpupd** function is called to allow the user to enter items for the new order. Menu actions are enabled/disabled depending on the result of the operation, using the **setup_actions** function..
- Lines 57 thru 61 perform the 'find' action to search for orders in the database. The **order_query** function is called and menu actions are enabled/disabled depending on the result of the operation, using the **setup_actions** function..
- Lines 62 thru 65 handle navigation in the order list after a search. Function **order_fetch_rel** is used to fetch the previous or next record.
- Line 67 calls the function **items_inpupd** to allow the user to edit the **items** associated with the displayed **order**.
- Line 72 closes the window before leaving the program.

Function setup_actions

This function is used by the main menu to enable or disable actions based on the context.

Function setup_actions (orders.4gl)

```
01 FUNCTION setup_actions(d, has_order, query_ok)
02   DEFINE d ui.Dialog,
03          has_order, query_ok SMALLINT
04   CALL d.setActionActive("next",query_ok)
05   CALL d.setActionActive("previous",query_ok)
06   CALL d.setActionActive("getitems",has_order)
07 END FUNCTION
```

Notes:

- Line 01 Three parameters are passed to the function:

- **d** - the predefined Dialog object
- **has_order** - if the value is TRUE, indicates that there is a new or existing order selected.
- **query_ok** - if the value is TRUE, indicates that the search for orders was successful.

- Lines 04 and 05 use the ui.Dialog.setActionActive method to enable or disable '**next**' and '**previous**' actions based on the value of **query_ok,** which indicates whether the search for orders was successful.
- Line 06 uses the same method to enable the '**getitems**' action based on the value of **has_order**, which indicates whether there is an order currently selected.

Function order_new

This function handles the input of an order record.

Function order_new (orders.4gl)

```
01 FUNCTION order_new()
02   DEFINE id INTEGER, name STRING
03
04   MESSAGE msg11
05
06   INITIALIZE order_rec.* TO NULL
07   SELECT MAX(order_num)+1 INTO order_rec.order_num
08     FROM orders
09   IF order_rec.order_num IS NULL
10    OR order_rec.order_num == 0 THEN
11      LET order_rec.order_num = 1
12   END IF
13
14   LET int_flag = FALSE
15   INPUT BY NAME
16       order_rec.store_num,
17       order_rec.store_name,
18       order_rec.order_num,
19       order_rec.order_date,
20       order_rec.fac_code,
21       order_rec.ship_instr,
22       order_rec.promo
23     WITHOUT DEFAULTS
24     ATTRIBUTES(UNBUFFERED)
25
26     BEFORE INPUT
27        LET order_rec.order_date = TODAY
28        LET order_rec.fac_code = "ASC"
29        LET order_rec.ship_instr = "FEDEX"
30
31     ON CHANGE store_num
32        SELECT store_name INTO order_rec.store_name
33          FROM customer
```

```
34          WHERE store_num = order_rec.store_num
35        IF (SQLCA.SQLCODE == NOTFOUND) THEN
36           ERROR msg12
37           NEXT FIELD store_num
38        END IF
39
40     ON ACTION zoom1
41        CALL display_custlist() RETURNING id, name
42        IF (id > 0) THEN
43           LET order_rec.store_num = id
44           LET order_rec.store_name = name
45        END IF
46
47   END INPUT
48
49   IF (int_flag) THEN
50      LET int_flag=FALSE
51      CLEAR FORM
52      MESSAGE msg03
53      RETURN FALSE
54   END IF
55
56   RETURN order_insert()
57
58 END FUNCTION
```

Notes:

- Lines 07 and 12 execute a SELECT to get a new order number from the database. If no rows are found, the order number is initialized to 1.
- Lines 15 thru 47 use the INPUT interactive dialog statement to let the user input the order data.
- Lines 26 thru 29 the BEFORE INPUT block initializes some members of the **order_rec** record, as default values for input.
- Lines 31 thru 38 the ON CHANGE block on the **store_num** field retrieves the customer name for the changed store_num from the customer table, and stores it in the **store_name** field. If the customer doesn't exist in the customer table, an error message displays.
- Lines 40 thru 45 implement the code to open the zoom window of the **store_num** BUTTONEDIT field, when the action **zoom1** is triggered. The function **display_custlist** in the custlist.4gl module allows the user to select a customer from a list. The action **zoom1** is enabled during the INPUT statement only.
- Line 56 calls the **order_insert** function to perform the INSERT SQL statement.

Function order_insert

This function inserts a new record in the orders database table.

Function order_insert (orders.4gl)

```
01 FUNCTION order_insert()
02
03   WHENEVER ERROR CONTINUE
04   INSERT INTO orders (
05      store_num,
06      order_num,
07      order_date,
08      fac_code,
09      ship_instr,
10      promo
11    ) VALUES (
12      order_rec.store_num,
13      order_rec.order_num,
14      order_rec.order_date,
15      order_rec.fac_code,
16      order_rec.ship_instr,
17      order_rec.promo
18    )
19    WHENEVER ERROR STOP
20
21    IF (SQLCA.SQLCODE <> 0) THEN
22      CLEAR FORM
23      ERROR SQLERRMESSAGE
24      RETURN FALSE
25    END IF
27
28    MESSAGE "Order added"
29    RETURN TRUE
30
31 END FUNCTION
```

Notes:

- Lines 03 thru 19 implement the INSERT SQL statement to create a new row in the **orders** table.
- Lines 21 thru 25 handle potential SQL errors, and display a message and return FALSE if the insert was not successful..
- Lines 28 and 29 display a message and return TRUE in case of success.

Function order_query

This function allows the user to enter query criteria for the **orders** table. It calls the function **order_select** to retrieve the rows from the database table.

Function order_query (orders.4gl)

```
01 FUNCTION order_query()
02   DEFINE where_clause STRING,
03          id INTEGER, name STRING
04
05   MESSAGE msg02
```

```
06
07   LET int_flag = FALSE
08   CONSTRUCT BY NAME where_clause ON
09       orders.store_num,
10       customer.store_name,
11       orders.order_num,
12       orders.order_date,
13       orders.fac_code
14
15     ON ACTION zoom1
16        CALL display_custlist() RETURNING id, name
17        IF id > 0 THEN
18           DISPLAY id TO orders.store_num
19           DISPLAY name TO customer.store_name
20        END IF
21
22   END CONSTRUCT
23
24   IF (int_flag) THEN
25      LET int_flag=FALSE
26      CLEAR FORM
27      MESSAGE msg03
28      RETURN FALSE
29   END IF
30
31   RETURN order_select(where_clause)
32
33 END FUNCTION
```

Notes:

- Lines 08 thru 22 The CONSTRUCT statement allows the user to query on specific fields, restricting the columns in the **orders** table that can be used for query criteria.
- Lines 15 thru 20 handle the '**zoom1**' action to let the user pick a customer from a list. The function **display_custlist** is called, it returns the customer number and name.
- Lines 24 through 29 check the value of the interrupt flag, and return FALSE if the user has interrupted the query.
- Line 31 the query criteria stored in the variable **where_clause** is passed to the function **order_select**. TRUE or FALSE is returned from the **order_select** function.

Function order_fetch

This function retrieves the row from the **orders** table, and is designed to be re-used each time a row is needed. If the retrieval of the row from the **orders** table is successful, the function **items_fetch** is called to retrieve the corresponding rows from the **items** table.

Function fetch_order (orders.4gl)

```
01 FUNCTION order_fetch(p_fetch_flag)
02   DEFINE p_fetch_flag SMALLINT
03
04   IF p_fetch_flag = 1 THEN
05      FETCH NEXT order_curs INTO order_rec.*
06   ELSE
07      FETCH PREVIOUS order_curs INTO order_rec.*
08   END IF
09
10   IF (SQLCA.SQLCODE == NOTFOUND) THEN
11      RETURN FALSE
12   END IF
13
14   DISPLAY BY NAME order_rec.*
15   CALL items_fetch()
16   RETURN TRUE
17
18 END FUNCTION
```

Notes:

- Line 05 When the parameter passed to this function and stored in the variable **p_fetch_flag** is 1, the FETCH statement retrieves the next row from the **orders** table.
- Line 07 When the parameter passed to this function and stored in **p_fetch_flag** is not 1, the FETCH statement retrieves the previous row from the **orders** table.
- Lines 10 thru 12 return FALSE if no row was found..
- Line 14 uses DISPLAY BY NAME to display the record **order_rec**.
- Line 15 calls the function **items_fetch**, to fetch all order lines.
- Line 16 returns TRUE indicating the fetch of the order was successful.

Function order_select

This function creates the SQL statement for the query and the corresponding cursor to retrieve the rows from the **orders** table. It calls the function **fetch_order**.

Function order_select (orders.4gl)

```
01 FUNCTION order_select(where_clause)
02   DEFINE where_clause STRING,
03          sql_text STRING
04
05   LET sql_text = "SELECT "
05       || "orders.store_num, "
06       || "customer.store_name, "
07       || "orders.order_num, "
08       || "orders.order_date, "
09       || "orders.fac_code, "
10       || "orders.ship_instr, "
```

```
11        || "orders.promo "
12        || "FROM orders, customer "
13        || "WHERE orders.store_num = customer.store_num "
14        || "AND " || where_clause
15
16    DECLARE order_curs SCROLL CURSOR FROM sql_text
17    OPEN order_curs
18    IF (NOT order_fetch(1)) THEN
19       CLEAR FORM
20       MESSAGE msg04
21       RETURN FALSE
22    END IF
23
24    RETURN TRUE
25
26 END FUNCTION
```

Notes:

- Lines 05 thru 14 contain the text of the SELECT statement with the query criteria contained in the variable **where_clause**.
- Line 16 declares a SCROLL CURSOR for the SELECT statement stored in the variable **sql_text**.
- Line 17 opens the SCROLL CURSOR.
- Line 18 thru 22 call the function **order_fetch**, passing a parameter of 1 to fetch the next row, which in this case will be the first one. If the fetch is not successful, FALSE is returned.
- Line 24 returns TRUE, indicating the fetch was successful.

Function order_fetch_rel

This function calls the function **order_fetch** to retrieve the rows in the database; the parameter p_fetch_flag indicates the direction for the cursor movement. If there are no more records to be retrieved, a message is displayed to the user.

Function order_fetch_rel

```
01 FUNCTION order_fetch_rel(p_fetch_flag)
02    DEFINE p_fetch_flag SMALLINT
03
04    MESSAGE " "
05    IF (NOT order_fetch(p_fetch_flag)) THEN
06      IF (p_fetch_flag = 1) THEN
07        MESSAGE msg05
08      ELSE
09        MESSAGE msg06
10      END IF
11    END IF
12
13 END FUNCTION
```

Notes:

- Line 05 calls the function **order_fetch**, passing the variable **p_fletch_flag** to indicate the direction of the cursor.
- Line 07 displays a message to indicate that the cursor is at the bottom of the result set.
- Line 09 displays a message to indicate that the cursor is at the top of the result set.

Function order_total

This function calculates the total price for all of the items contained on a single order.

Function order_total (orders.4gl)

```
01 FUNCTION order_total(arr_length)
02   DEFINE order_total DECIMAL(9,2),
03          i, arr_length SMALLINT
04
05   LET order_total = 0
06   IF arr_length > 0 THEN
07      FOR i = 1 TO arr_length
08         IF arr_items[i].line_total IS NOT NULL THEN
09           LET order_total = order_total + arr_items[i].line_total
10         END IF
11      END FOR
12   END IF
13
14   DISPLAY BY NAME order_total
15
16 END FUNCTION
```

Notes:

- Line 07 thru 11 contain a FOR loop adding the values of **line_total** from each item in the program array **arr_items**, to calculate the total price of the order and store it in the variable **order_total.**
- Line 14 displays the value of **order_total** on the form.

Function order_close

This function closes the cursor used to select orders from the database.

Function order_close (orders.4gl)

```
01 FUNCTION close_order()
02   WHENEVER ERROR CONTINUE
```

```
03   CLOSE order_curs
04   WHENEVER ERROR STOP
05 END FUNCTION
```

Notes:

- Line 03 closes the **order_curs** cursor. The statement is surrounded by WHENEVER ERROR, to trap errors if the cursor is not open.

Function items_fetch

This function retrieves the rows from the **items** table that match the value of **order_num** in the order currently displayed on the form. The **description** and **unit** values are retrieved from the **stock** table, using the column **stock_num.** The value for **line_total** is calculated and retrieved. After displaying the items on the form, the function **order_total** is called to calculate the total price of all the items for the current order.

Function items_fetch (orders.4gl)

```
01 FUNCTION items_fetch()
02   DEFINE item_cnt INTEGER,
03          item_rec RECORD
04             stock_num    LIKE items.stock_num,
05             description  LIKE stock.description,
06             quantity     LIKE items.quantity,
07             unit         LIKE stock.unit,
08             price        LIKE items.price,
09             line_total   DECIMAL(9,2)
10          END RECORD
11
12   IF order_rec.order_num IS NULL THEN
13      RETURN
14   END IF
15
16   DECLARE items_curs CURSOR FOR
17      SELECT items.stock_num,
18             stock.description,
19             items.quantity,
20             stock.unit,
21             items.price,
22             items.price * items.quantity line_total
23        FROM items, stock
24       WHERE items.order_num = order_rec.order_num
25         AND items.stock_num = stock.stock_num
26
27   LET item_cnt = 0
28   CALL arr_items.clear()
29   FOREACH items_curs INTO item_rec.*
30       LET item_cnt = item_cnt + 1
31       LET arr_items[item_cnt].* = item_rec.*
32   END FOREACH
33   FREE items_curs
```

```
34
35   CALL items_show()
36   CALL order_total(item_cnt)
37
38 END FUNCTION
```

Notes:

- Line 02 defines a variable **item_cnt** to hold the array count.
- Line 12 returns from the function if the order number in the program record order_rec is null.
- Lines 16 thru 25 declare a cursor for the SELECT statement to retrieve the rows from the **items** table that have the same order number as the value in the **order_num** field of the program record **order_rec**. The **description** and **unit** values are retrieved from the **stock** table, using the column **stock_num.** The value for **line_total** is calculated.
- Lines 29 thru 32 the FOREACH statement loads the dynamic array **arr_items**.
- Line 33 releases the memory associated with the cursor **items_curs**, which is no longer needed.
- Lines 35 calls the **items_show** function to display the order lines to the form.
- Line 36 calls the function **order_total** to calculate the total price of the items on the order.

Function items_show

This function displays the line items for the order in the screen array and returns immediately.

Function items_show (orders.4gl)

```
01 FUNCTION items_show()
02   DISPLAY ARRAY arr_items TO sa_items.*
03       BEFORE DISPLAY
04         EXIT DISPLAY
05   END DISPLAY
06 END FUNCTION
```

Notes:

- Line 02 executes a DISPLAY ARRAY statement with the program array containing the line items.
- Line 03 and 04 exit the instruction before control is turned over to the user.

Function items_inpupd

This function contains the program logic to allow the user to input a new row in the **arr_items** array, or to change or delete an existing row.

Function items_inpupd

```
01 FUNCTION items_inpupd()
02   DEFINE opflag CHAR(1),
03          item_cnt, curr_pa SMALLINT,
04          id INTEGER
05
06   LET opflag = "U"
07
08   LET item_cnt = arr_items.getLength()
09   INPUT ARRAY arr_items WITHOUT DEFAULTS FROM sa_items.*
10     ATTRIBUTES (UNBUFFERED, INSERT ROW = FALSE)
11
12     BEFORE ROW
13       LET curr_pa = ARR_CURR()
14       LET opflag = "U"
15
16     BEFORE INSERT
17       LET opflag = "I"
18       LET arr_items[curr_pa].quantity = 1
19
20     AFTER INSERT
21       CALL item_insert(curr_pa)
22       CALL items_line_total(curr_pa)
23
24     BEFORE DELETE
25       CALL item_delete(curr_pa)
26
27     ON ROW CHANGE
28       CALL item_update(curr_pa)
29       CALL items_line_total(curr_pa)
30
31     BEFORE FIELD stock_num
32       IF opflag = "U" THEN
33          NEXT FIELD quantity
34       END IF
35
36     ON ACTION zoom2
37        LET id = display_stocklist()
38        IF id > 0 THEN
39           IF (NOT get_stock_info(curr_pa,id) ) THEN
40              LET arr_items[curr_pa].stock_num = NULL
41           ELSE
42              LET arr_items[curr_pa].stock_num = id
43           END IF
44        END IF
45
46     ON CHANGE stock_num
47        IF (NOT get_stock_info(curr_pa,
48                       arr_items[curr_pa].stock_num) ) THEN
49           LET arr_items[curr_pa].stock_num = NULL
```

```
50           ERROR msg07
51           NEXT FIELD stock_num
52        END IF
53
54     ON CHANGE quantity
55        IF (arr_items[curr_pa].quantity <= 0) THEN
56           ERROR msg13
57           NEXT FIELD quantity
58        END IF
59
60   END INPUT
61
62   LET item_cnt = arr_items.getLength()
63   CALL ord_total(item_cnt)
64
65   IF (int_flag) THEN
66      LET int_flag = FALSE
67   END IF
68
69 END FUNCTION
```

Notes:

- Line 08 uses the **getLength** built-in function to determine the number of rows in the array **arr_items**.
- Lines 9 thru 60 contain the INPUT ARRAY statement.
- Lines 12 and 14 use a BEFORE ROW clause to store the index of the current row of the array in the variable **curr_pa**. We also set the **opflag** flag to "U", in order to indicate we are in update mode.
- Lines 16 thru 18 use a BEFORE INSERT clause to set the value of **opflag** to "I" if the current operation is an Insert of a new row in the array. Line 18 sets a default value for the quantity.
- Lines 20 thru 22 An AFTER INSERT clause calls the **item_insert** function to add the row to the database table, passing the index of the current row and calls the **items_line_total** function, passing the index of the current row.
- Lines 24 thru 25 use a BEFORE DELETE clause, to call the function **item_delete**, passing the index of the current row.
- Lines 27 thru 29 contain an ON ROW CHANGE clause to detect row modification. The **item_update** function and the **items_line_total** function are called, passing the index of the current row.
- Lines 31 thru 34 use a BEFORE FIELD clause to prevent entry in the **stock_num** field if the current operation is an Update of an existing row.
- Lines 36 thru 44 implement the code for the '**zoom2**' action, opening a list from the **stock** table for selection.
- Lines 46 thru 52 use an ON CHANGE clause to check whether the stock number for a new record that was entered in the field **stock_num** exists in the **stock** table.
- Line 62 uses the **getLength** built-in function to determine the number of rows in the array after the INPUT ARRAY statement has terminated.
- Line 63 calls the function **order_total**, passing the number of rows in the array.
- Lines 65 thru 67 re-set the INT_FLAG to TRUE if the user has interrupted the INPUT statement.

Function items_line_total

This function calculates the value of **line_total** for any new rows that are inserted into the **arr_items** array.

Function items_line_total

```
01 FUNCTION items_line_total(curr_pa)
02   DEFINE curr_pa SMALLINT
03   LET arr_items[curr_pa].line_total =
04       arr_items[curr_pa].quantity * arr_items[curr_pa].price
05 END FUNCTION
```

Notes:

- Line 02 The index of the current row in the array is passed to this function and stored in the variable **curr_pa**.
- Lines 03 and 04 calculate the line total for the current row in the array.

Function item_insert

This function inserts a new row into the **items** database table using the values input in the current array record on the form.

Function item_insert

```
01 FUNCTION item_insert(curr_pa)
02   DEFINE curr_pa SMALLINT
03
04   WHENEVER ERROR CONTINUE
05   INSERT INTO items (
06      order_num,
07      stock_num,
08      quantity,
09      price
10   ) VALUES (
11      order_rec.order_num,
12      arr_items[curr_pa].stock_num,
13      arr_items[curr_pa].quantity,
14      arr_items[curr_pa].price
15   )
16   WHENEVER ERROR STOP
17
18   IF (SQLCA.SQLCODE == 0) THEN
19      MESSAGE msg08
20   ELSE
21      ERROR SQLERRMESSAGE
22   END IF
23
```

```
24 END FUNCTION
```

Notes:

- Line 02 the index of the current row in the array is passed to this function and stored in the variable **curr_pa.**
- Lines 05 thru 15 The embedded SQL INSERT statement uses the value of **order_num** from the current order record displayed on the form, together with the values from the current row of the **arr_items** array, to insert a new row in the **items** table.

Function item_update

This function updates a row in the **items** database table using the changes made to the current array record in the form.

Function item_update

```
01 FUNCTION item_update(curr_pa)
02   DEFINE curr_pa SMALLINT
03
04   WHENEVER ERROR CONTINUE
05   UPDATE items SET
06     items.stock_num = arr_items[curr_pa].stock_num,
07     items.quantity  = arr_items[curr_pa].quantity
08      WHERE items.stock_num = arr_items[curr_pa].stock_num
09        AND items.order_num = order_rec.order_num
10   WHENEVER ERROR STOP
11
12   IF (SQLCA.SQLCODE == 0) THEN
13      MESSAGE msg09
14   ELSE
15      ERROR SQLERRMESSAGE
16   END IF
17
18 END FUNCTION
```

Notes:

- Line 02 the index of the current row in the array is passed to this function and stored in the variable **curr_pa.**
- Lines 05 thru 09 The embedded SQL UPDATE statement uses the value of **order_num** in the current **order_rec** record, and the value of **stock_num** in the current row in the a**rr_items** array, to locate the row in the **items** database table to be updated.

Function item_delete

This function deletes a row from the **items** database table, based on the values in the current record of the items array.

Function item_delete

```
01 FUNCTION item_delete(curr_pa)
02   DEFINE curr_pa SMALLINT
03
04   WHENEVER ERROR CONTINUE
05   DELETE FROM items
06      WHERE items.stock_num = arr_items[curr_pa].stock_num
07      AND items.order_num = order_rec.order_num
08   WHENEVER ERROR STOP
09
10   IF (SQLCA.SQLCODE == 0) THEN
11      MESSAGE msg10
12   ELSE
13      ERROR SQLERRMESSAGE
14   END IF
15
16 END FUNCTION
```

Notes:

- Line 02 the index of the current row in the array is passed to this function and stored in the variable **curr_pa.**
- Lines 05 thru 07 The embedded SQL DELETE statement uses the value of **order_num** in the current **order_rec** record, and the value of **stock_num** in the current row in the **arr_items** array, to locate the row in the **items** database table to be deleted.

Function get_stock_info

This function verifies that the stock number entered for a new row in the **arr_items** array exists in the **stock** table. It retrieves the description, unit of measure, and the correct price based on whether promotional pricing is in effect for the order.

Function get_stock_info

```
01 FUNCTION get_stock_info(curr_pa, id)
02   DEFINE curr_pa SMALLINT,
03          id INTEGER,
04          sqltext STRING
05
06   IF id IS NULL THEN
07      RETURN FALSE
08   END IF
09
10   LET sqltext="SELECT description, unit,"
```

```
11    IF order_rec.promo = "N" THEN
12       LET sqltext=sqltext || "reg_price"
13    ELSE
14       LET sqltext=sqltext || "promo_price"
15    END IF
16    LET sqltext=sqltext ||
17        " FROM stock WHERE stock_num = ? AND fac_code = ?"
18
19    WHENEVER ERROR CONTINUE
20    PREPARE get_stock_cursor FROM sqltext
21    EXECUTE get_stock_cursor
22          INTO arr_items[curr_pa].description,
23               arr_items[curr_pa].unit,
24               arr_items[curr_pa].price
25          USING id, order_rec.fac_code
26    WHENEVER ERROR STOP
27
28    RETURN (SQLCA.SQLCODE == 0)
29
30 END FUNCTION
```

Notes:

- Line 02 the index of the current row in the array is passed to this function and stored in the variable **curr_pa**.
- Lines 10 thru 17 check whether the promotional pricing is in effect for the current order, and build a SELECT statement to retrieve the description, unit, and regular or promotional price from the **stock** table for a new item that is being added to the **items** table.
- Lines 20 thru 25 prepare and execute the SQL statement created before.
- Line 28 checks SQLCA.SQLCODE and returns TRUE if the database could be updated without error.

Tutorial Chapter 12: Changing the User Interface Dynamically

Summary:

- Built-in Classes
- Using the Classes (Window Class example)
  - Getting a reference to the object
  - Calling a method
- Working with Forms
  - Getting a reference to the object
  - Specifying the name of a form item
- Changing the text, image, or style of a form item
- Hiding form items
- Adding Toolbars, Topmenus, and Action Defaults
- Specifying a function to initialize all forms
- Loading a ComboBox list
- Using the Dialog Class in an Interactive Statement
  - Hiding Form Items
  - Enabling and Disabling Fields
- Using the Interface Class
  - Refresh the Interface
  - Load custom XML files
  - Identify the Genero Client

Built-in Classes

Included in the predefined functions that are built into Genero are special groups (classes) of functions (methods) that act upon the objects that are created when your program is running. Each class of methods interacts with a specific program object, allowing you to change the appearance or behavior of the objects. Because these methods act upon program objects, the syntax is somewhat different from that of functions.

The classes are gathered together into packages:

- **ui** - classes related to the objects in the graphical user interface (GUI)
- **base** - classes related to non-GUI program objects
- **om** - classes that provide DOM and SAX document handling utilities

This tutorial focuses on using the classes and methods in the **ui** package to modify the user interface at runtime.

Note: Variable names, class identifiers, and method names are not case-sensitive; the capitalization used in the examples is for ease in reading.

Using the Classes

This example for the Window Class also presents the general process that you should use.

The methods in the Window Class interact with the Window objects in your program.

Getting a reference to the object

Before you can call any of the methods associated with Window objects, you must identify the specific Window object that you wish to affect, and obtain a reference to it:

- Define a variable to hold the reference to the Window object. The data type of the variable is the class identifier (ui.Window):

```
DEFINE mywin ui.Window
```

- Open a window in your program using the OPEN WINDOW or OPEN WINDOW ... WITH FORM ... instruction:

```
OPEN WINDOW w1 WITH FORM "testform"
```

- Get a reference to the specific Window object by using one of two "class methods" provided by the Window Class. Class methods are called using the class identifier (ui.Window). You can specify the Window object by name from among the open windows in your program, or choose the current window.

```
LET mywin = ui.Window.getCurrent() -- returns a reference
to
                                   the current window
object
LET mywin = ui.Window.forName("w1")-- returns a reference
to
                                   the open window named
"w1"
```

Calling a method

Now that you have a reference to the object, you can use that reference to call any of the methods listed as "object methods" in the Window Class documentation. For example, to change the window title for the window referenced by mywin:

```
CALL mywin.setText("test")
```

See Window Class for a complete list of the methods in this class.

Example 1:

```
01 MAIN
02  DEFINE mywin ui.Window
03
04  OPEN WINDOW w1 WITH FORM "testform"
05  LET mywin = ui.Window.getCurrent()
06  CALL mywin.setText("test")
07  MENU
08   ON ACTION quit
09     EXIT MENU
10  END MENU
11
12 END MAIN
```

Display on Windows platforms:

Warning: Using an object reference that has not been initialized or points to a non-existent object (the window has been closed, the form object does not exist, etc.) results in a run-time error that is not trappable; test for NULL prior to using an object reference in your program code. For example, pass the object reference to a utility function for testing:

```
FUNCTION cleanupForm(f)
  DEFINE f ui.Form
  IF f IS NULL THEN
    RETURN FALSE
  END IF
END FUNCTION
```

Working with Forms

The Form Class provides some methods that allow you to change the appearance or behavior of items on a form.

Getting a reference to the Form object

In order to use the methods, you must get a reference to the form object. The Window Class has a method to get the reference to its associated form:

- Define variables for the references to the window object and to its form object. The data type for the variables is the class identifier (ui.Window, ui.Form):

```
DEFINE f1 ui.Form, mywin ui.Window
```

- Open a form in your program using the OPEN WINDOW ... WITH FORM ... instruction:

```
OPEN WINDOW w1 WITH FORM ("testform")
```

- Next, get a reference to the window object. Then, use the **getForm()** class method of the Window Class to get a reference to the form object opened in that window:

```
LET mywin = ui.Window.getCurrent()
LET f1 = mywin.getForm() -- returns reference to form
```

Once you have the reference to the form object, you can call any of the object methods for the Form class:

```
LET mywin = ui.Window.getCurrent()
LET f1 = mywin.getForm() -- get reference to form
-- call a Form Class method
CALL f1.loadActionDefaults("mydefaults")
```

See the Form Class documentation for a complete list of methods.

Specifying the name of a form item

Some of the methods in the Form Class require you to provide the name of the form item. The name of the form item in the Attributes section of the form specification file corresponds to the **name** attribute of an element in the runtime form file. For example:

- In the Attributes section of the .per file

```
LABEL a1 : lb1, TEXT = "State";
EDIT a2 = state.state_name;
BUTTON a3 : quit, TEXT = "exit";
EDIT a4 = FORMONLY.pflag TYPE CHAR;
```

- In the runtime .42f file

```
<Label name="lb1" width="9" text="State" posY="0" posX="6"
gridWidth="9"/>
```

```
<FormField name="state.state_name" colName="state_name"
sqlType="CHAR(15)"
    fieldId="0" sqlTabName="state" tabIndex="1">
<Button name="quit" width="5" text="exit" posY="4" posX="6"
gridWidth="5"/>
<FormField name="formonly.pflag" colName="pflag"
sqlType="CHAR" fieldId="1"
    sqlTabName="formonly" tabIndex="2">
```

Note: Formfield names specified as FORMONLY (FORMONLY.pflag) are converted to lowercase (formonly.pflag).

Although Genero BDL is not case-sensitive, XML is. When Genero creates the runtime XML file, the form item types and attribute names are converted using the CamelCase convention:

- Form item type - the first letter is always capitalized, with subsequent letters in lower-case, unless the type consists of multiple words joined together. In that case, the first letter of every subsequent word is capitalized also (Label, FormField, Button).
- Attribute name - the first letter is always lower-case, with subsequent letters in lower-case, unless the name consists of multiple words joined together. In that case, the first letter of every subsequent word is capitalized also (text, gridWidth, colName).

If you use classes or methods in your code that require the form item type or attribute name, respect the naming conventions.

Changing the text, image, and style properties of a form item

Some methods of the Form Class allow you to change the value of specific properties of form items.

Call the methods using the reference to the form object. Provide the name of the form item and the value for the property:

- **Text** property - the value can be any text string. To set the text of the label named "lb1":

  ```
  CALL f1.setElementText("lb1", "Newtext")
  ```

- **Image** property - the value can be a simple file name, a complete or relative path, or an URL (Uniform Resource Locator) path to an image server. To set the image for the button named "quit":

  ```
  CALL f1.setElementImage("quit", "exit.png")
  ```

- **Style** property - the value can be a presentation style defined in the active Presentation Styles file (.4st file). To set the style for the label named "lb1":

  ```
  CALL f1.setElementStyle("lb1", "mystyle")
  ```

  The style "mystyle" is an example of a specific style that was defined in a custom Presentation Styles XML file, **customstyles.4st**. This style changes the text color to blue:

  ```
  <Style name=".mystyle" >
     <StyleAttribute name="textColor" value="blue" />
  </Style>
  ```

  By default, the runtime system searches for the **default.4st** Presentation Style file. Use the following method to load a different Presentation Style file:

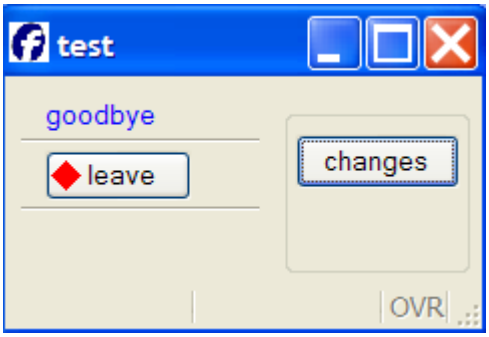
  ```
  CALL ui.interface.loadStyles("customstyles")
  ```

  The Load custom XML files section has more information about the Interface class. See Presentation Styles for additional information about styles and the format of a Presentation Styles file.

Example 2:

```
01 MAIN
02   DEFINE mywin ui.Window,
03          f1    ui.Form
04   CALL ui.interface.loadStyles("customstyles")
05   OPEN WINDOW w1 WITH FORM "testform"
06   LET mywin = ui.Window.getCurrent()
07     CALL mywin.setText("test")
08     LET f1 = mywin.getForm()
09   MENU
10     ON ACTION changes
11       CALL f1.setElementText("lb1", "goodbye")
12       CALL f1.setElementText("quit", "leave")
13       CALL f1.setElementImage("quit", "exit.png")
14       CALL f1.setElementStyle("lb1", "mystyle")
15     ON ACTION quit
16       EXIT MENU
17   END MENU
18 END MAIN
```

Display on Windows platform after the changes button has been clicked:



Hiding Form Items

You can use Form Class methods to change the value of the **hidden** property of form items, hiding parts of the form from the user. Interactive instructions such as INPUT or CONSTRUCT will automatically ignore a formfield that is hidden. The value can be:

- **0** - the form item is not hidden; it is visible
- **1** - the form item is hidden and cannot be made visible by the user
- **2** - the form item is hidden, but the user can make it visible, using the context menu for a table, for example

By default, all form items are visible.

Call the methods using the reference to the form object. Provide the name of the form item to the method and set the value for hidden.

- **setFieldHidden()** - this method can be used to hide formfields only. The prefix in the name of the formfield (*tablename.* or *formonly.*) is optional:

```
CALL f1.setFieldHidden("state_name",1)
```

- **setElementHidden()** - this method hides any form item, including formfields. If the item is a formfield, the name must include the prefix:

```
CALL f1.setElementHidden("lb1", 1)
CALL f1.setElementHidden("state.state_name",1)
CALL f1.setElementHidden("formonly.pflag",1)
```

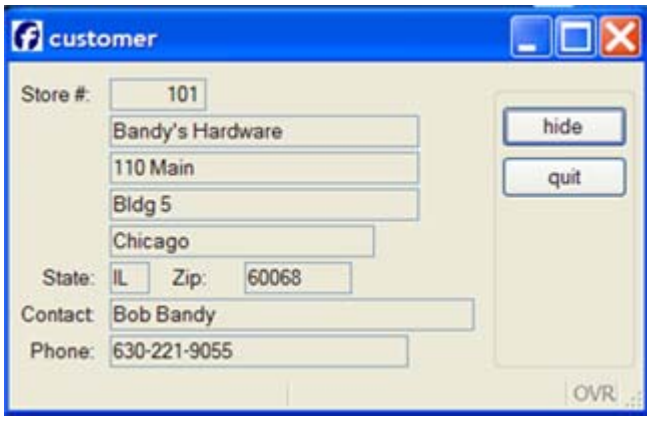
Genero adjusts the display of the form to eliminate blank spaces caused by hiding items, where possible.

Example 3:

```
01 SCHEMA custdemo
02 MAIN
03  DEFINE win ui.Window,
04         fm ui.Form,
```

```
05         mycust record like customer.*
06  CONNECT TO "custdemo"
07  OPEN WINDOW w1 WITH FORM "hidecust"
08  SELECT * INTO mycust.* FROM customer
09      WHERE store_num = 101
10  DISPLAY BY NAME mycust.*
11  LET win = ui.Window.getCurrent()
12  LET fm = win.getForm()
13  MENU
14   ON ACTION hide
15     CALL fm.setFieldHidden("contact_name",1)
16     CALL fm.setFieldHidden("addr2", 1)
17     -- hide the label for contact name
18     CALL fm.setElementHidden("lbl", 1)
19   ON ACTION quit
20     EXIT MENU
21  END MENU
22 END MAIN
```

Display on Windows platforms (before hiding):



After hiding:

Adding toolbars, topmenus, and action defaults

The Form Class provides methods that apply topmenus, toolbars, and action defaults to a form, to assist you in standardizing forms. The topmenus, toolbars, or action defaults are defined in external XML files having the following extensions:

- Action Defaults - **.4ad**
- Toolbar - **.4tb**
- Topmenu - **.4tm**

Call the methods using the reference to the form object and give the filename. Do not specify a path or file extension in the file name. If the file is not in the current directory and the path is not specified, Genero will search the directories indicated by the DBPATH environment variable.

- Action defaults file - default attributes for form items associated with actions; these action defaults are local to the form. See Action Defaults for information about the format and contents of the file.

  ```
  CALL f1.loadActionDefaults("mydefaults")
  ```

- Toolbar file - contains a toolbar definition to be used with the referenced form object. See Toolbars for information about the format and contents of the file.

  ```
  CALL f1.loadToolBar("mytoolbar")
  ```

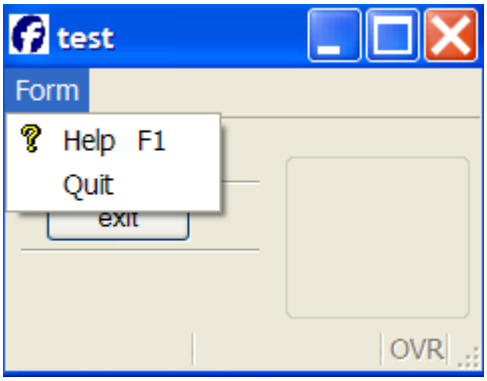
- Topmenu file - contains a topmenu definition to be used with the referenced form object. See Topmenus for information about the format and contents of the file.

  ```
  CALL f1.loadTopMenu("mytopmenu")
  ```

Example 4:

```
01 MAIN
02  DEFINE mywin ui.Window,
03         f1    ui.Form
04  OPEN WINDOW w1 WITH FORM "testform"
05  LET mywin = ui.Window.forName("w1")
06  CALL mywin.setText("test")
07  LET f1 = mywin.getForm()
08  CALL f1.loadTopMenu("mytopmenu")
09  MENU
10    ON ACTION quit
11      EXIT MENU
12  END MENU
13
14 END MAIN
```

Display on Windows platforms:



Specifying a Function to Initialize all Forms

To assist in standardizing forms, you can create an initializer function in your program that will be called automatically whenever any form is opened. A reference to the form object is passed by the runtime system to the function.

Example initializer function:

```
01 FUNCTION myforminit(f1)
02   DEFINE f1 ui.Form
03
04   IF f1 IS NOT NULL THEN
05     CALL f1.loadTopMenu("mytopmenu")
06     ...
07   END IF
07
08 END FUNCTION
```

The setDefaultInitializer method applies to all forms, rather than to a specific form object. It is a class method, and you call it using the class name as a prefix. Specify the name of the initializer function in lower-case letters:

```
CALL ui.Form.setDefaultInitializer("myforminit")
```

You can call the **myforminit** function in your program as part of a setup routine. The **myforminit** function can be in any module in the program.

Example 5:

```
01 MAIN
02  CALL ui.Form.setDefaultInitializer("myforminit")
03  OPEN WINDOW w1 WITH FORM "testform"
04  MENU
05   ON ACTION quit
06    EXIT MENU
07  END MENU
08  OPEN WINDOW w2 WITH FORM "testform2"
```

```
09  MENU
10   ON ACTION quit
11    EXIT MENU
12  END MENU
13 END MAIN
```

Display on Windows platforms:

Loading a ComboBox List

A ComboBox presents a list of values in a dropdown box on a form. The values are for the underlying formfield. For example, the following form specification file contains a ComboBox that represents the formfield **customer.state**:

```
01 SCHEMA custdemo
02 LAYOUT
03  GRID
04  {
05   Store #:[a0  ]
06      Name:[a1                ]
07     State:[a5      ]
08  }
09  END -- GRID
10 END
11 TABLES customer
12 ATTRIBUTES
13   EDIT a0=customer.store_num;
```

```
14   EDIT a1=customer.store_name;
15  COMBOBOX a5=customer.state;
16 END
```

During an INPUT, INPUT ARRAY or CONSTRUCT statement the ComboBox is active, and the user can select a value from the dropdown list. The value selected will be stored in the formfield named **customer.state**.

Getting a reference to the object

The ComboBox Class contains methods that manage the values for a ComboBox. In order to use these methods you must first obtain a reference to the ComboBox object:

- Define a variable for the reference to the ComboBox object. The data type for the variables is the class identifier (ui.ComboBox):

  ```
  DEFINE cb ui.ComboBox
  ```

- Open a form that contains a ComboBox using OPEN WINDOW ... WITH FORM ... :

  ```
  OPEN WINDOW w1 WITH FORM ("testcb")
  ```

- Next, get a reference to the ComboBox object using the method provided. As a "class method", this method is called using the class identifier. Provide the name of the formfield to the method:

  ```
  LET cb = ui.ComboBox.forName("customer.state")
  ```

Once you have a reference to the ComboBox object, you can call any of the methods defined in the class as "object methods":

- To add an item to a ComboBox list

  You can instruct the ComboBox to store a code (the "name") in the formfield that the ComboBox represents, but to display the description (the "text") in the list to help the user make his selection. For example, to store the value "IL" (*name*) in the formfield, but to display "Illinois" (*text*) to the user:

  ```
  CALL cb.additem("IL", "Illinois")
  ```

  If *text* is NULL, *name* will be displayed.

- To clear the list of all values

  ```
  CALL cb.clear()
  ```

- To remove an item from the list; provide the *name*

```
      CALL cb.removeitem("IL")
```

See the ComboBox Class documentation for a complete list of the methods.

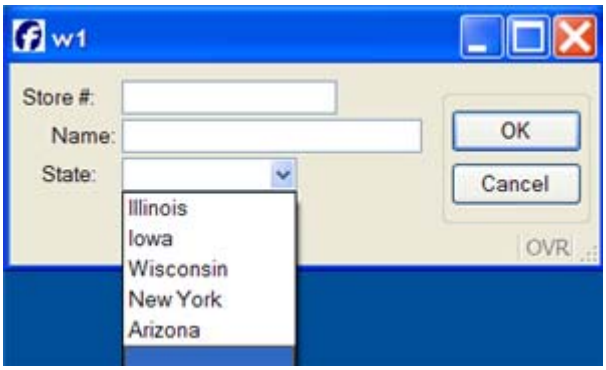
Adding values to the ComboBox from a Database Table

An example in Tutorial Chapter 5 GUI Options loads a ComboBox with static values. The following example retrieves the valid list of values from a database table (state) instead:

Example 6:

```
01 SCHEMA custdemo
02 MAIN
03  DEFINE cb ui.ComboBox
04  CONNECT TO "custdemo"
05  OPEN WINDOW w1 WITH FORM "testcb"
06  LET cb = ui.ComboBox.forName("customer.state")
07  IF cb IS NOT NULL THEN
08   CALL loadcb(cb)
09  END IF
10 ...
11 END MAIN
12
13 FUNCTION loadcb(cb)
12  DEFINE cb ui.ComboBox,
13         l_state_code LIKE state.state_code,
14         l_state_name LIKE state.state_name
15
18  DECLARE mycurs CURSOR FOR
19    SELECT state_code, state_name FROM state
20  CALL cb.clear()
21  FOREACH mycurs INTO l_state_code, l_state_name
22  -- provide name and text for the ComboBox item
23    CALL cb.addItem(l_state_code,l_state_name)
24  END FOREACH
26 END FUNCTION
```

Display on Windows platforms



As an alternative, this function can be specified as the initializer function for the ComboBox in the form specification file. When the form is opened, The initializer function is called automatically and a reference to the ComboBox object is passed to it. Provide the name of the initializer function in lowercase:

```
ATTRIBUTES
COMBOBOX a5=customer.state, INITIALIZER = loadcb;
```

Using the Dialog Class in Interactive Statements

The Dialog Class provides methods that can only be called from within an interactive instruction (dialog) such as MENU, INPUT, INPUT ARRAY, DISPLAY ARRAY and CONSTRUCT. The methods are called through the predefined variable DIALOG, which automatically provides a reference to the Dialog object.

Tutorial Chapter 5 Enhancing the Form illustrates the use of Dialog Class methods to disable/enable actions during a MENU interactive statement.

Hiding Default Action Views

To hide default action views (the buttons that appear on the form when there is no specific action view for an action), use the following Dialog Class method. Values for the hidden state of the action view can be:

- 0 - FALSE, the action is visible
- 1 - TRUE, the action is hidden

```
MENU
  BEFORE MENU
   CALL DIALOG.setActionHidden("next",1)
  ...
END MENU
```

This example hides the action that has the name **next**. The reference to the DIALOG object was provided by the runtime system.

Enabling and Disabling Fields

This method in the Dialog Class allows you to disable fields on a form during the interactive statement; the field is still visible, but the user cannot edit the value. Values for the active state of the field can be:

- 0 - FALSE, the field is disabled
- 1 - TRUE, the field is enabled

The reference to the DIALOG object is provided by the runtime system. Provide the name of the field and its state to the method.

The following example disables the **store_name** field during an INPUT statement:

```
INPUT BY NAME customer.*
 BEFORE INPUT
   CALL DIALOG.setFieldActive("customer.store_name",0)
 ...
END INPUT
```

See the Dialog Class documentation for a complete list of its methods.

Using the Interface Class

Methods in the Interface Class allow you interact with the user interface, as shown in the examples below.

You do not need to get an object reference to the Interface; call the methods in the Interface Class using the class identifier, ui.Interface.

Refresh the interface

The User Interface on the Client is synchronized with the DOM tree of the runtime system when an interactive statement is active. If you want to show something on the screen while the program is running in a batch procedure, you must force synchronization with the front end.

As shown in the Tutorial Chapter 9 Reports, the changes made in the program to the value of the progress bar are not displayed on the user's window, since the report is a batch process and no user interaction is required. To force the changes in the progress bar to be reflected on the screen, the following method from the Interface Class is used:

```
CALL ui.Interface.refresh()
```

Load custom XML files

- Start Menus, Toolbar icons, and Topmenus can each be defined in a unique XML file.

  Use the appropriate extension:

  - Start Menu - .4sm
  - Toolbar - .4tb
  - Topmenu - .4tm

  Use the corresponding method to load the file:

```
CALL ui.Interface.loadStartMenu("mystartmenu")
CALL ui.Interface.loadTopMenu("tmstandard")
```

```
CALL ui.Interface.loadToolbar("tbstandard")
```

Do not specify a path or file extension in the file name. The runtime system automatically searches for a file with the correct extension in the current directory and in the path list defined in the DBPATH environment variable.

See the Start Menu, Topmenu, or Toolbar documentation for details on the format and contents of the files.

- Custom Presentation Styles and global Action Defaults must each be defined in a unique file.

  Use the appropriate extension:

  - Presentation Styles - .4st
  - Action Defaults - .4ad

  Use the corresponding method to load the file:

```
CALL ui.Interface.loadStyles("mystyles")
CALL
ui.Interface.loadActionDefaults("mydefaults")
```

  You can provide an absolute path with the corresponding extension, or a simple file name without the extension. If you give the simple file name, the runtime system searches for the file in the current directory. If the file does not exist, it searches in the directories defined in the DBPATH environment variable.

  The action defaults are applied only once, to newly created elements. For example, if you first load a toolbar, then you load a global Action defaults file, the attribute of the toolbar items will not be updated with the last loaded Action defaults.

  See Presentation Styles and Action Defaults for details on the format and contents of the file.

Identify the Genero client

You can use methods in the Interface Class to identify the type and version of the Genero client currently being used by the program:

```
CALL ui.Interface.getFrontEndName() RETURNING typestring
CALL ui.Interface.getFrontEndVersion() RETURNING
versionstring
```

Each method returns a string. The type will be "Gdc" or "Console".

Some of the other methods in the ui.Interface class allow you to:

- Set and retrieve program names and titles
- call Front End functions that reside on the Genero client
- work with MDI windows

See the Interface Class documentation for a complete list of the methods.

ODI Adaptation Guide For Genero db 3.6x, 3.8x

Runtime configuration

- Install Genero db and create a database
- Prepare the runtime environment

Database concepts

- Database concepts
- Data consistency and concurrency management
- Transactions handling
- Defining database users
- Setting privileges

Data dictionary

- BOOLEAN data type
- CHARACTER data types
- NUMERIC data types
- DATE and DATETIME data types
- INTERVAL data type
- SERIAL data type
- ROWIDs
- Very large data types
- National character data types
- The ALTER TABLE instruction
- Constraints
- Triggers and Stored Procedures
- Name resolution of SQL objects
- Setup database statistics
- Data type conversion table

Data manipulation

- Reserved words
- Outer joins
- Transactions handling
- Temporary tables

Substrings in SQL
Name resolution of SQL objects
String delimiters and object names
Getting one row with SELECT
MATCHES and LIKE conditions
SQL functions and constants
Querying system catalog tables
Syntax of UPDATE statements
The USER constant

BDL programming

SERIAL data type
IBM®Informix®specific SQL statements in BDL
INSERT cursors
Cursors WITH HOLD
SELECT FOR UPDATE
UPDATE/DELETE WHERE CURRENT OF <cursor>
The LOAD and UNLOAD instructions
SQL Interruption

Runtime configuration

Install Genero db and create a database

1. Install Genero db on your computer.

By default, Genero db is configured to run alone on a networked machine, having client applications hosted on other machines. Here are some tips to setup Genero db , you should change some configuration parameters:

- If applications are co-located on the same machine as the database server: By default, Genero db uses nearly all the memory available on the computer. To share the memory with applications, you must change the **MEMORY_OVERRIDE** parameter in $ANTSHOME/Server/config.txt.

- You can also use the **networking=IPC** communication protocol in the data source definition to get better performance.
- For production use, a caching RAID controller on which to do logging is highly recommended. For maximum data integrity, **LOG_MODE** should be set to **DURABLE**, and **LOGPATH** should point to a different disk than the disk containing data. If **LOGPATH** does not point to a disk managed by a RAID controller, **LOG_MODE** should be set to **OSDURABLE**. See the database server documentation for more details.
- Genero db by default relies on local-area-network multicast to enable clients and servers to find each other. If you have a firewall, you must allow **UDP** connections on port **12345** for the multicast address **255.0.0.37**. If you want to disable multicast search: on the server side, set **MULTICAST_ENABLED=FALSE** in $ANTSHOME/Server/config.txt. On the client side, for Unix platforms, you must set **overridebroadcast=yes** in the ODBC data source definition. For Windows platforms, check the "**Override Multicast**" option in the ODBC data source configuration (click the "*Advanced*" button, then click "*Connection Method*" in the "*Networking*" section).

2. Set up an ODBC data source, called *mydb*.

   The defaults ODBC data source settings need to be adapted:
   - Set **overridebroadcast=yes** if you don't need multicast, as described in (1).
   - During the installation of Genero db, the data source might have been created with a login and password of the SYSTEM user. If you leave these default user and password entries, anyone can connect to the database as the SYSTEM super user. You must clean the **user** and **password** ODBC parameters. This will force client programs to specify a login and password to connect to the database.
   - By default, the Genero db client re-connects automatically to the server if the connection is lost. This is a useful feature as long as the SQL session is stateless . However, if

you create temporary tables or if you change session parameters with ALTER SESSION, the session context will be silently lost if a re-connect occurs. To get an SQL error when DB connection is lost, you can deny the client to re-connect by setting **automaticfailover=no** in the ODBC parameters. Note that the Genero db client does not re-connect automatically if a transaction was started. In such case the program gets and SQL error.

- Typical Genero FGL applications connect only once to the database server. By default, the Genero db client uses connection aggregators, which re-use connection resources and server bindings for new connections. This is useful when the same client process opens and closes many connections, but is unnecessary overhead in typical Genero FGL applications. Additionally, using aggregation implies multithreading. UNIX signal handling are not thread safe. Since the runtime system uses the UNIX signal() function, **you must disable aggregation by setting noaggregates=0 in the ODBC data source definition.** On Windows platforms, the noaggregators property can only be changed with the registry editor, under the ODBC.INI key.
- By default the ODBC client library uses a large buffer to pre-fetch rows from the server. This gives very good performance but consumes memory. If your application uses many cursors fetching large result sets, you can reduce the memory footprint with the **fetchsize** ODBC.INI parameter. On the other hand, the Genero db client maintains a statement pool by default. When you FREE a cursor or statement in your programs, the underlying ODBC statement handle structure goes to the statement pool for future reuse. You can disable the ODBC statement pool by setting the **disableStmtPool** ODBC.INI parameter to **Yes**. When this parameter is set, fetch buffers are freed as well.

3. Create a database user dedicated to your application. You can use the antscmd tool.

```
$ antscmd -d mydb -u SYSTEM -p SYSTEM
```

```
    mydb> CREATE USER appadmin IDENTIFIED
BY "password";
```

You must grant privileges to this user:

```
   mydb> GRANT CREATE TABLE TO appadmin;
   mydb> GRANT CREATE VIEW TO appadmin;
   mydb> GRANT CREATE SYNONYM TO
appadmin;
   mydb> GRANT CREATE PROCEDURE TO
appadmin;
   mydb> GRANT CREATE SEQUENCE TO
appadmin;
```

4. Create the application tables.

   Do not forget to convert Informix data types to Genero db data types. See issue ODIADS100 for more details.

   Check for reserved words in your table and column names.

Prepare the runtime environment

1. If you want to connect to a remote Genero db server from an application server, you must have ODBC properly configured on your application server.
2. Verify if the ODBC environment is correct.

   ```
   $ antscmd -d dns-name -u appadmin -p
   password
   ```

3. Verify the environment variable defining the search path for shared libraries. On UNIX platforms, the variable is specific to the operating system. For example, on Solaris and Linux systems, it is LD_LIBRARY_PATH.
4. Check the database client character set (**characterset ODBC parameter**) The DB locale must match the locale used by the runtime system (**LANG**).
5. Set up the fglprofile entries for database connections to your data source.
6. Create normal application users and define the schema to be used.

   With Genero db, a schema is created when creating a user. If the *APPADMIN* user creates the tables, the schema for application tables will be "APPADMIN".

In order to make application tables visible to normal DB users, you can specify a default schema for normal users by adding the DEFAULT SCHEMA clause in CREATE USER:

```
   mydb> CREATE USER username IDENTIFIED
BY password DEFAULT SCHEMA appadmin;
```

You can also use the following FGLPROFILE entry to make the database driver select a default schema after connection:

```
   dbi.database.dbname.ads.schema =
"name"
```

Here <**dbname**> identifies the database name used in the BDL program (DATABASE **dbname**) and <**name**> is the schema name to be used.

If needed, database users can be authenticated as *Operating System users*. In order to create a DB user authenticated by the operating system, use the IDENTIFIED EXTERNALLY clause in CREATE USER:

```
   mydb> CREATE USER username IDENTIFIED
EXTERNALLY;
```

The OS users will be able to connect to the database if the $ANTSHOME/Server/ants.rhosts file contains an entry to identify the OS user. See the Genero db documentation for more details.

Warning: Pay attention to the user name, which is case-sensitive. You must specify the user name in double quotes; otherwisem, the name defaults to uppercase letters.

7. Grant privileges to application users:

   By default the tables created by the *appadmin* user cannot be modified by the application users; you must first grant privileges:

```
   mydb> GRANT SELECT, INSERT, UPDATE,
DELETE ON tablename TO username;
```

   You can do this for all existing and future users by specifying PUBLIC as the grantee:

```
   mydb> GRANT SELECT, INSERT, UPDATE,
DELETE ON tablename TO PUBLIC;
```

   If the database has stored procedures, you must

also grant execute permission to application users:

```
   mydb> GRANT EXECUTE ON procname TO
username;
```

8. In order to connect to Genero db, you must have a database driver "dbmads*" installed.

ODIADS001 - DATE and DATETIME data types

Informix provides two data types to store date and time information:

- **DATE** = for year, month and day storage.
- **DATETIME** = for year to fraction of a second (1-5) storage.

Genero db provides four data types to store date and time information:

- **DATE** = for year, month, and day storage.
- **TIME** = for hour, minute, and second storage.
- **TIMESTAMP** = for year, month, day, hour, min, second, and fraction of second storage.
- **DATETIME** = Synonym for TIMESTAMP.

String representing date time information:

Informix is able to convert quoted strings to DATE / DATETIME data, if the string formatting matches the formatting set by environment parameters (i.e. DBDATE, GL_DATETIME).

Genero db can also convert quoted strings to DATE / TIME / TIMESTAMP; by default, date/time formats follow the ISO standard (2005-01-30). You can control date/time format with the ANTS locale settings (ANTS_DATE_FORMAT, ANTS_TIME_FORMAT, ANTS_TIMESTAMP_FORMAT).

Date arithmetic:

- Informix supports date arithmetic on DATE and DATETIME values. The result of an arithmetic expression involving dates or times is a number of days when only DATEs are used, and an INTERVAL value if a DATETIME is used in the expression.
- In Genero db, the result of an arithmetic expression involving DATE values is a number of days. You can subtract or add a integer to a DATE column.
- Informix automatically converts an integer to a date when the integer is used to set a value of a date column. Genero db does not do this conversion; review your code and change it to use a DATE type variable.
- Complex DATETIME expressions (involving INTERVAL values for example) are Informix-specific and have no equivalent in Genero db.

Solution:

The Genero db DATE type is used for Informix DATE data.

Informix DATETIME data with a precision from HOUR TO SECOND is stored in a Genero db TIME column. DATETIME data with any other precision is stored in Genero db TIMESTAMP columns. The database interface makes the conversion automatically. Missing date or time parts default to 1900-01-01 00:00:00. For example, when using a DATETIME HOUR TO MINUTE with the value of "11:45", the Genero db DATETIME value will be "1900-01-01 11:45:00".

Warning: Using integers (the number of days since 1899/12/31) as dates is supported by Genero db in a SELECT INTO statement but not in a WHERE clause. Check your code to detect the use of integers with DATE columns. Also note that SELECT TO_NUM('1900-01-01' AS INT) will return 0 and not 1. (With Informix & Genero db, a date of 1900/1/1 when selected into an INTEGER will return 1.)

Warning: Literal DATETIME expressions (i.e. DATETIME 1999-10-12 YEAR TO DAY) are not converted.

Warning: It is strongly recommended that you use BDL variables in dynamic SQL statements instead of quoted strings representing DATEs. For example:

```
   LET stmt = "SELECT ... FROM customer WHERE create_date >'",
adate,"'"
```

is not portable. Use a question mark place holder instead and OPEN the cursor by USING a date:

```
   LET stmt = "SELECT ... FROM customer WHERE create_date > ?"
```

Note: Most arithmetic expressions involving dates (for example, to add or remove a number of days from a date) will produce the same results with Genero db.

Warning: DATE arithmetic expressions using SQL parameters (USING variables) are not fully supported. For example: `"SELECT ... WHERE datecol < ? + 1"` generates an error at PREPARE time.

Warning: SQL Statements using expressions with EXTEND must be reviewed and adapted to the native syntax.

ODIADS003 - Reserved words

SQL object names, like table and column names, cannot be SQL reserved words in Genero db. An example of a common word which is part of the Genero db SQL grammar is **'level**'

Solution:

You must rename those table or column names that are Genero db reserved words. Genero db reserved keywords are listed in the Genero db documentation.
Another solution is to enclose the table/column name in double quotes. Double-quoted table/column names are case-sensitive. If this double-quoted syntax is used, all subsequent references to this table/column must be in the same double-quoted format.

ODIADS004 - ROWIDs

Genero db provides ROWIDs, but the data type is different from Informix. Informix ROWIDs are INTEGERs, while Genero db ROWIDs are BIGINT.

Warning: Genero db ROWIDs can be used to identify a unique row during the lifetime of the transaction. After the transaction is committed, the ROWID may change.

With Informix, SQLCA.SQLERRD[6] contains the ROWID of the last INSERTed or UPDATEd row. This is not currently supported with Genero db.

Solution:

Warning: Genero db ROWIDs are not fully compatible with Informix ROWIDs.

It is recommended that you review the code and remove any usage of ROWIDs, as their usage is not portable to other databases and may lead to problems when the code runs against any other databases. (For example, Oracle has ROWIDs, but they are CHARs instead of numeric.)

ODIADS005 - SERIAL data type

Informix SERIAL data type and automatic number production:

- The table column must be of type SERIAL.
- To generate a new serial, no value or a zero value is specified in the INSERT statement:
  ```
  INSERT INTO tab1 ( c ) VALUES ( 'aa' )
  INSERT INTO tab1 ( k, c ) VALUES ( 0, 'aa' )
  ```
- After INSERT, the new SERIAL value is provided in `SQLCA.SQLERRD[2]`.

Informix allows you to insert rows with a value other than zero for a serial column. Using an explicit value automatically increments the internal serial counter, to avoid conflicts with future INSERTs that are using a zero value:

```
CREATE TABLE tab ( k SERIAL );  --> internal counter = 0
INSERT INTO tab VALUES ( 0 );   --> internal counter = 1
INSERT INTO tab VALUES ( 10 );  --> internal counter = 10
INSERT INTO tab VALUES ( 0 );   --> internal counter = 11
```

```
DELETE FROM tab;                --> internal counter = 11
INSERT INTO tab VALUES ( 0 );   --> internal counter = 12
```

Genero db supports SERIAL the same as in Informix.

Solution:

When using Genero db, the SERIAL data type works the same as in Informix. After an insert, sqlca.sqlerrd[2] holds the last generated serial value.

CREATE [TEMP] TABLE with a SERIAL column works as in Informix.

ODIADS006 - Outer joins

The Genero db syntax for OUTER joins is different from Informix:

In Informix SQL, outer tables are defined in the FROM clause using the OUTER keyword:

```
SELECT ... FROM a, OUTER(b)
 WHERE a.key = b.akey
SELECT ... FROM a, OUTER(b,OUTER(c))
 WHERE a.key = b.akey
   AND b.key1 = c.bkey1
   AND b.key2 = c.bkey2
```

Genero db version 3.60 supports the same OUTER joins syntax as Informix.

Warning: Genero db version 3.4 does not support Informix-style OUTER joins.

Genero db also supports ANSI syntax joins:

```
SELECT ... FROM a, LEFT OUTER JOIN b ON a.key = b.key
SELECT ... FROM a
    LEFT OUTER JOIN  b LEFT OUTER JOIN c
    ON ( (b.key1 = c.bkey1) AND (b.key2 = c.bkey2) ) ON (
(a.key = b.akey) )
```

Solution:

None required.

ODIADS007a - Database concepts

Most BDL applications use only one database instance (in the meaning of Informix). But Informix servers can handle multiple database instances, while Genero db servers manage only one database instance. However, Genero db can manage multiple schemas.

```
SELECT * FROM stores.customer
```

Solution:

With Genero db, you can create as many users as database schemas are needed. You typically dedicate a database user to administer each occurrence of the application database (i.e. schema in Genero db).

Any user can select the current database schema with the following SQL command:

```
SET SCHEMA "<schema>"
```

Using this instruction, any user can access the tables without giving the owner prefix, as long as the table owner has granted privileges required to access the tables.

Genero db users can be associated to a default schema as follows:

```
CREATE USER "<username>" IDENTIFIED ...
       DEFAULT SCHEMA "<schema>"
```

This is the preferred way to assign a schema to DB users.

You can also make the database interface select the current schema automatically using the following FGLPROFILE entry:

```
dbi.database.<dbname>. ads. schema = "<schema>"
```

Warning: double-quoted schema/user names are case-sensitive.

ODIADS008a - Data consistency and concurrency management

Data consistency involves readers that want to access data currently being modified. Concurrency data access involves several writers accessing the same data for modification. Locking granularity defines the amount of data involved when a lock is set (row, page, table, and other groupings).

Informix:

Informix uses a locking mechanism to handle data consistency and concurrency. When a process changes database information with UPDATE, INSERT or DELETE, an exclusive lock is set on the touched rows. The lock remains active until the end of the transaction. Statements performed outside a transaction are treated as a transaction containing a single operation, and release the locks immediately after execution. SELECT statements can set shared locks according to the isolation level. In case of locking conflicts (for example, when two processes want to acquire an exclusive lock on the same row for modification, or when a writer is trying to modify data protected by a shared lock), the behavior of a process can be changed by setting the lock wait mode.

Control:

- Lock wait mode: `SET LOCK MODE TO ...`
- Isolation level: `SET ISOLATION TO ...`
- Locking granularity: `CREATE TABLE ... LOCK MODE {PAGE|ROW}`
- Explicit exclusive lock: `SELECT ... FOR UPDATE`

Defaults:

- The default isolation level is `READ COMMITTED.`
- The default lock wait mode is `NOT WAIT.`
- The default locking granularity is `PAGE.`

Genero db:

Genero db does not use the same locking mechanism as Informix to handle concurrency; however, it behaves the same way in terms of concurrency.

The following transaction control instructions have been implemented in Genero db:

- Lock wait mode: `SET LOCK MODE TO ...`
- Isolation level: `SET TRANSACTION ISOLATION LEVEL ...`
- Explicit exclusive lock: `SELECT ... FOR UPDATE`

Solution:

You can use the same transaction control instructions and update clauses as in Informix:

- `SET LOCK MODE ...`
- `SELECT ... FOR UPDATE`

Warning: The `SET ISOLATION TO ...` Informix syntax is replaced by `SET TRANSACTION ISOLATION LEVEL ...` in Genero db.

Warning: The `LOCK MODE {PAGE|ROW}` is not provided by Genero db. This is specific to data storage mechanisms and cannot be supported in the Genero db concurrency model.

ODIADS008b - SELECT FOR UPDATE

Many BDL programs implement pessimistic locking in order to prevent several users editing the same rows at the same time.

```
DECLARE cc CURSOR FOR
   SELECT ... FOR UPDATE [OF column-list]
OPEN cc
FETCH cc <-- lock is acquired
CLOSE cc <-- lock is released
```

- The row must be fetched in order to set the lock.
- If the cursor is local to a transaction, the lock is released when the transaction ends.
  If the cursor is declared "WITH HOLD", the lock is released when the cursor is closed.

Genero db allows individual and exclusive row locking with:

```
SELECT ... FOR UPDATE [OF column-list]
```

- A lock is acquired for each selected row when the cursor is opened (before the first fetch).
- The lock is only released when the transaction ends.
- FOR UPDATE cursors can only be OPENed inside a transaction.

Genero db locking granularity is at the row level.

To control the behavior of the program when locking rows, Informix provides a specific instruction to set the wait mode:

```
SET LOCK MODE TO { WAIT | NOT WAIT | WAIT seconds }
```

The default mode is NOT WAIT. This is an Informix-specific SQL statement.

Solution:

Genero db supports SELECT .. FOR UPDATE as in Informix, but the rows are locked when the cursor is opened, not when the first row is fetched.

Ensure that the use of 'FOR UPDATE' is always inside a transaction.

Ensure that you COMMIT the transaction as soon as possible to prevent rows being locked longer than necessary.

ODIADS009a - Transactions handling

Informix and Genero db handle transactions in a similar manner but with minor differences.

In both Informix and Genero db, transactions must be started with BEGIN WORK and finished with COMMIT WORK or ROLLBACK WORK.

Statements executed outside a transaction are automatically committed.

Warning: With Informix in native mode (non-ANSI), DDL statements can be executed (and cancelled) in transactions. Genero db does not support DDL statements inside transactions.

Solution:

Regarding transaction control instructions, existing applications do not have to be modified in order to work with Genero db.

Warning: You must extract the DDL statements from transaction blocks.

Transactions in stored procedures: avoid using transactions in stored procedures and allow the client applications to handle transactions, in accordance with the transaction model.

See also ODIADS008b

ODIADS010 - BOOLEAN data type

Informix provides the BOOLEAN data type, as a "Built-in Opaque Data Type". It is used to store Boolean values. You can use the character literals 't' for true and 'f' for false.

Genero db 3.4 does not have a BOOLEAN type.

Warning: Genero db 3.60 has implemented a BOOLEAN type which can store the following values (case-insensitive): TRUE/FALSE or 1/0. You can't use the 't' or 'f' character values instead.

Solution:

We don't recommend the use of the BOOLEAN Genero db datatype.

You must review the database creation scripts and the programs. Replace any BOOLEAN column by a CHAR(1).

ODIADS011a - CHARACTER data types

Informix provides the CHAR and VARCHAR data types to store characters. CHAR columns can store up to **32,767** characters; VARCHARs are limited to **255** characters. Starting with IDS 2000, Informix provides the LVARCHAR data type which is limited to 2K characters.

Genero db provides the CHAR and VARCHAR data types. Both data types support a length of **60000** bytes (or **3000** if the column is indexed).

String comparison semantics are equivalent in Informix and Genero db:

- Trailing blanks are ignored when comparing CHAR and VARCHAR values.
- Genero db treats empty strings as NOT NULL values (like Informix).

Solution:

The database interface supports character string variables in SQL statements for input (USING) and output (INTO).

ODIADS012 - Constraints

Constraint naming syntax:

Both INFORMIX and Genero db support primary key, unique, foreign key, default and check constraints, but the constraint naming syntax is different : Genero db expects the "CONSTRAINT" keyword **before** the constraint specification and INFORMIX expects it **after**.

UNIQUE constraint example:

INFORMIX	Genero db

`CREATE TABLE scott.emp (`
`...`
`empcode CHAR(10) UNIQUE`
`  [CONSTRAINT pk_emp],`
`...`
	`CREATE TABLE scott.emp (`
	`...`
	`empcode CHAR(10)`
	`  [CONSTRAINT pk_emp]`
	`UNIQUE,`
	`...`

Primary keys:

Like INFORMIX, Genero db creates an index to enforce PRIMARY KEY constraints (some RDBMS do not create indexes for constraints).

Unique constraints:

Like INFORMIX, Genero db creates an index to enforce UNIQUE constraints (some RDBMS do not create indexes for constraints).

Warning: Using CREATE UNIQUE INDEX is silently converted to a unique constraint. To drop an index created as CREATE UNIQUE INDEX, you must do an ALTER TABLE DROP CONSTRAINT.

Warning: When using a unique constraint, INFORMIX allows only one row with a NULL value, while Genero db allows several rows with NULL!

Foreign keys:

Both INFORMIX and Genero db support the ON DELETE CASCADE option. To defer constraint checking, INFORMIX provides the SET CONSTRAINT command while Genero db provides the DISABLE CONSTRAINTS hint.

Check constraints:

Warning: The check condition may be any valid expression that can be evaluated to TRUE or FALSE, including functions and literals. You must verify that the expression is not INFORMIX specific.

Null constraints:

INFORMIX and Genero db support not null constraints, but INFORMIX does not allow you to give a name to "NOT NULL" constraints.

Solution:

Constraint naming syntax:

The database interface does not convert constraint naming expressions when creating tables from BDL programs. Review the database creation scripts to adapt the constraint naming clauses for Genero db.

ODIADS013 - Triggers and Stored Procedures

Genero db supports the Informix trigger and stored procedure language.

See Genero db documentation for more details.

Solution:

None required.

ODIADS016a - Defining database users

Informix users are defined at the operating system level. They must be members of the 'Informix' group. The database administrator must grant CONNECT, RESOURCE or DBA privileges to those users.

Genero db users must be registered in the database. They are created by the database administrator with the following command:

```
CREATE USER <username> IDENTIFIED BY <pswd>
```

or for Operating System authentication:

```
CREATE USER <username> IDENTIFIED EXTERNALLY
```

Note: For defining database users, there is a file in the <install directory of the database>/Server/ants.rhosts . See the Genero db documentation for more information.

Solution:

For migration and testing purposes only, you can specify the user name and password in the FGLPROFILE.

For a live system, it is recommended that you use the CONNECT TO statement and supply the user name and password, or create database users IDENTIFIED EXTERNALLY.

ODIADS016b - Setting privileges

Informix and Genero db user privileges management are similar.

Genero db provides roles to group privileges which then can be assigned to users. Starting with version 7.20, Informix also provides roles.

Informix users must have at least the CONNECT privilege to access the database:

```
GRANT CONNECT TO (PUBLIC|user)
```

To be able to create tables, views or synonyms, Genero db users need:

```
GRANT CREATE TABLE TO <user>
GRANT CREATE VIEW TO <user>
GRANT CREATE SYNONYM TO <user>
```

Warning: Genero db does NOT provide the Informix CONNECT, RESOURCE and DBA roles.

Solution:

In Genero db, roles can be created with database privileges to simulate Informix system roles.

ODIADS017 - Temporary tables

Informix supports temporary tables with the following statements:

```
SELECT ... INTO TEMP tmpname
CREATE TEMP TABLE tmpname ( ... )
```

Genero db supports the same temporary table instructions as Informix.

Solution:

None required.

ODIADS018 - Substrings in SQL

Informix SQL statements can use substrings on columns defined with the character data type:

```
SELECT ... FROM tab1 WHERE col1[2,3] = 'RO'
SELECT ... FROM tab1 WHERE col1[10]  = 'R'   -- Same as col1[10,10]
UPDATE tab1 SET col1[2,3] = 'RO' WHERE ...
SELECT ... FROM tab1 ORDER BY col1[1,3]
```

Genero db provides the SUBSTRING( ) function, to extract a sub-string from a string expression:

```
SELECT .... FROM tab1 WHERE SUBSTRING(col1,2,2) = 'RO'
SELECT SUBSTRING('Some text' FROM 6 FOR 3)        -- Gives 'tex'
```

Genero db 3.60 has implemented the `col[x,y]` expression but not the `col[x]` one.

Solution:

The Genero db driver will convert SQL expressions containing Informix substring syntax for you. It is recommended, however, that you replace all Informix `col[x,y]` expressions with `SUBSTRING(col FROM x FOR y-x+1)`.

Warning: In UPDATE instructions, setting column values using subscripts will produce an error with Genero db:

```
UPDATE tab1 SET col1[2,3] = 'RO' WHERE ...
```

is converted to:

```
UPDATE tab1 SET SUBSTRING(col1 FROM 2 FOR 3-2+1) = 'RO' WHERE ...
```

ODIADS019 - Name resolution of SQL objects

Informix uses the following to identify an SQL object:

```
[database[@dbservername]:][{owner|"owner"}.]identifier
```

The ANSI convention is to use double-quotes for identifier delimiters (For example: "tabname"."colname").

Warning: When using double-quoted identifiers, both Informix and Genero db become case-sensitive. Unlike Informix, Genero db object names are stored in UPPERCASE in system catalogs. That means that SELECT "col1" FROM "tab1" will produce an error if those objects are created without double-quotes; they are identified by COL1 and TAB1 in Genero db system catalogs.

With Informix ANSI-compliant databases:

- The table name must include "owner", unless the connected user is the owner of the database object.
- The database server shifts the owner name to uppercase letters before the statement executes, unless the owner name is enclosed in double quotes.

With Genero db, an object name takes the following form:

```
[(schema|"schema").](identifier|"identifier")
```

Object names are limited to 128 chars in Genero db.

A Genero db schema is owned by a user (usually the application administrator).

Solution:

Check that you do not use single-quoted or double-quoted table names or column names in your static SQL statements. Those quotes must be removed because the database interface automatically converts double quotes to single quotes, and Genero db does not allow single quotes as database object name delimiters.

See also issue ODIADS007a

ODIADS020 - String delimiters and object names

The ANSI string delimiter character is the single quote ('string'). Double quotes are used to delimit database object names ("object-name").

Example: `WHERE "tabname"."colname" = 'a string value'`

Informix allows double quotes as string delimiters, but Genero db doesn't. This is an important distinction, as many BDL programs use double quotes to delimit the strings in SQL commands.

Remark: This problem concerns only double quotes within dynamic SQL statements. Double quotes used in pure BDL string expressions are not subject to SQL compatibility problems. Double-quoted string literals in static SQL statements are converted to single-quoted strings by compilers.

Genero db implements ANSI-compliant SQL syntax and therefore does not support double-quoted string literals; only database object names can be double-quoted.

Solution:

The Genero db driver can automatically replace all double quotes with single quotes.

Escaped string delimiters can be used inside strings like the following:

```
'This is a single quote: '''
'This is a single quote: \''
"This is a double quote: """
"This is a double quote: \""
```

Warning: Database object names cannot be delimited by double quotes, because the database interface cannot determine the difference between a database object name and a quoted string!

For example, if the program executes the SQL statement:
`WHERE "tabname"."colname" = "a string value"`
replacing all double quotes with single quotes would produce:
`WHERE 'tabname'.'colname' = 'a string value'`
This would produce an error since `'tabname'.'colname'` is not allowed by Genero db.

Although double quotes are replaced automatically in SQL statements, you should use only single quotes to enforce portability.

ODIADS021 - NUMERIC data types

Informix supports several data types to store numbers:

Informix data type	Synonym	Description
SMALLINT		16 bit integer (-2^15 to 2^15)
INTEGER	INT	32 bit integer (-2^31 to 2^31)
DECIMAL(p)	DEC(p)	Floating-point decimal number
DECIMAL(p,s)	DEC(p,s) /	Fixed-point decimal number

	NUMERIC(p,s)	
MONEY		Equivalent to DECIMAL(16,2)
MONEY(p)		Equivalent to DECIMAL(p,2)
MONEY(p,s)		Equivalent to DECIMAL(p,s)
SMALLFLOAT	REAL	approximate floating point (C float)
FLOAT(*n*)	DOUBLE PRECISION	approximate floating point (C double). *n* is ignored !

Genero db supports the following:

Genero db data type	Pseudotype	Description
SMALLINT	TINYINT / BIT	16 bit integer
INT / INTEGER		32 bit integer
BIGINT		64 bit integer
DECIMAL(p,s)		Fixed-point decimal number **(p<=15)**
NUMERIC(p,s)		Fixed-point decimal number **(p<=15)**
MONEY		Number with precision nearly 19 and scale 4
DOUBLE / DOUBLE PRECISION / REAL	SMALLFLOAT / FLOAT(n)	approximate floating point (C double)

Warning: The only difference between DECIMAL and NUMERIC is that NUMERIC guarantees the specified precision, whereas DECIMAL guarantees at least the specified precision.

Warning: A pseudotype is accepted anywhere a regular type name is, but is silently converted into another type which is supported by Genero db.

Solution:

We recommend that you use the following conversion rules:

Informix data type	Genero db data type
SMALLINT	SMALLINT
INTEGER	INTEGER
DECIMAL(p)	DOUBLE / DOUBLE PRECISION / REAL
DECIMAL(p,s)	DECIMAL(p,s)
MONEY(p,s)	DECIMAL(p,s)

SMALLFLOAT	DOUBLE / DOUBLE PRECISION / REAL
FLOAT(n)	DOUBLE / DOUBLE PRECISION / REAL

Warning: Genero db 3.60 DECIMAL can store up to **15 digits**, while Informix DECIMAL can store **32**. A future version of Genero db will support DECIMAL (p<=32,s).

ODIADS022 - Getting one row with SELECT

With Informix, you must use the SYSTABLES system table with a condition on the table id:

```
SELECT user FROM systables WHERE tabid=1
```

With Genero db some statements can be as follows:

```
PREPARE pre FROM "SELECT USER" EXECUTE pre INTO l_user
```

Solution:

Check the BDL sources for "FROM systables WHERE tabid=1" and use dynamic SQL to resolve this problem.

ODIADS024 - MATCHES and LIKE in SQL conditions

Informix and Genero db both support MATCHES and LIKE in SQL statements.

MATCHES allows you to use brackets to specify a set of matching characters at a given position:

```
( col MATCHES '[Pp]aris' )
( col MATCHES '[0-9][a-z]*' )
```

In this case, the LIKE statement has no equivalent feature.

Genero db implements the MATCHES operator.

Solution:

None required.

See also: MATCHES operator in SQL Programming.

ODIADS025 - Informix specific SQL statements in BDL

The BDL compiler supports several Informix-specific SQL statements that have no meaning when using Genero db:

- CREATE DATABASE
- DROP DATABASE
- START DATABASE (SE only)
- ROLLFORWARD DATABASE
- SET [BUFFERED] LOG
- CREATE TABLE with special options (storage, lock mode, etc.)

Solution:

Review your BDL source and remove all static SQL statements which are Informix-specific SQL statements.

ODIADS028 - INSERT cursors

Informix supports insert cursors. An "insert cursor" is a special BDL cursor declared with an INSERT statement instead of a SELECT statement. When this kind of cursor is open, you can use the PUT instruction to add rows and the FLUSH instruction to insert the records into the database.

For Informix database with transactions, OPEN, PUT and FLUSH instructions must be executed within a transaction.

Genero db does not support insert cursors.

Solution:

Insert cursors are emulated by the Genero db driver.

ODIADS029 - SQL functions and constants

Almost all Informix functions and SQL constants have a different name or behavior in Genero db.

Comparison list of functions and constants:

Informix	Genero db

today	current_date / today (synonyms)
current year to second	current_timestamp
day(value)	dayofmonth(d '2002-12-31')
extend(dtvalue, first to last)	to_date(dtvalue, '<format>')
mdy(m,d,y)	mdy(m,d,y)
month(date)	month(date)
weekday(date)	dayofweek(date '2002-12-31')
year(date)	year(date)
date("string" | integer)	to_date('string', '<format>') No equivalent with integer.
user	user *! Uppercase/lowercase: See* ODIADS047
trim([leading | trailing | both "char" FROM] "string")	trim([[leading | trailing | both] [*pad_character*] from] *string*)
length(c)	length(c)
pow(x,y)	pow(x,y)

Solution:

Warning: You must review the SQL statements using CURRENT / EXTEND expressions.

ODIADS030 - Very large data types

Informix uses the TEXT and BYTE data types to store very large texts or images.

Genero db 3.4 provides CLOB and BLOB data types, and provides TEXT/BYTE synonyms for Informix compatibility.

Solution:

None required.

TEXT & BYTE are supported by Genero db and by the ADS database driver.

ODIADS031 - Cursors WITH HOLD

Informix closes opened cursors automatically when a transaction ends unless the WITH HOLD option is used in the DECLARE instruction.

By default Genero db keeps cursors open when a transaction ends (however, FOR UPDATE locks are released at the end of a transaction).

Solution:

BDL cursors are automatically closed when a COMMIT WORK or ROLLBACK WORK is performed.

WITH HOLD cursors with a SELECT FOR UPDATE can be supported, if the table has a primary key or a unique index.

ODIADS032 - UPDATE/DELETE WHERE CURRENT OF <cursor>

Informix allows positioned UPDATEs and DELETEs with the "WHERE CURRENT OF <cursor>" clause, if the cursor has been DECLARED with a SELECT ... FOR UPDATE statement.

Warning: UPDATE/DELETE ... WHERE CURRENT OF <cursor> is supported by the Genero db API. However, the cursor must be OPENed and used inside a transaction.

```
DECLARE cur1 CURSOR FOR SELECT * FROM mytable WHERE 1=1 FOR UPDATE
BEGIN WORK
OPEN cur1
FETCH cur1 INTO x,chr
UPDATE mytable SET mycol2 = "updated" WHERE CURRENT OF cur1
CLOSE cur1
COMMIT WORK
```

Solution:

Check that your programs correctly put WHERE CURRENT OF <cursorname> inside a transaction.

ODIADS033 - Querying system catalog tables

Both Informix and Genero db provides system catalog tables, however the table names and structure are different.

Genero db provides the standard views for system catalog: INFORMATION_SCHEMA.TABLES, INFORMATION_SCHEMA.COLUMNS, and so on.

Solution:

Warning: No automatic conversion of Informix system tables is provided by the database interface.

ODIADS034 - Syntax of UPDATE statements

Informix allows a specific syntax for UPDATE statements:

```
UPDATE table SET ( <col-list> ) = ( <val-list> )
```

Genero db supports this syntax.

BDL programs can have the following type of statements:

```
UPDATE table SET table.* = myrecord.*
UPDATE table SET * = myrecord.*
```

Static UPDATE statements using the above syntax are converted by the compiler to the standard form:

```
UPDATE table SET column=value [,...]
```

Solution:

None required.

ODIADS036 - INTERVAL data type

Informix's INTERVAL data type stores a value that represents a span of time. INTERVAL types are divided into two classes: *year-month intervals* and *day-time intervals*.

Genero db does not provide a data type similar to Informix INTERVAL.

Solution:

Warning: INTERVAL data types are not supported by Genero db 3.4.

It is not recommended that you use the INTERVAL data type because Genero db has no equivalent native data type. This would cause problems when doing INTERVAL arithmetic on the database server side. However, INTERVAL values can be stored in CHAR columns.

Remark: Genero db will support INTERVALs in a future version.

ODIADS040 - National characters data types

Informix provides the NCHAR & NVARCHAR data types to store locale-dependent character data, using a specific collation order.

Genero db 3.6 has the same internationalization solution.

Solution:

None required with Genero db 3.6.

ODIADS046 - The LOAD and UNLOAD instructions

Informix provides SQL instructions to export data from a database table and import data into a database table: The UNLOAD instruction copies rows from a database table into a text file; the LOAD instructions insert rows from a text file into a database table.

Genero db does not provide LOAD and UNLOAD instructions.

Genero db provides an Import/Export Utility (impexp) that will convert a specified set of tables, or an entire database, to or from a Comma Separated Value (CSV) external format.

Solution:

In 4gl programs, the LOAD and UNLOAD instructions are supported with Genero db, with some limitations:

Warning: There is a difference when you use Genero db DATETIME columns. DATETIME columns created in Genero db are equivalent to Informix DATETIME YEAR TO SECOND columns. In LOAD and UNLOAD, all Genero db DATE columns are treated as Informix DATETIME YEAR TO SECOND columns and thus will be unloaded with the "YYYY-MM-DD hh:mm:ss" format.

Warning: When using an Informix database, simple dates are unloaded using the DBDATE format (ex: "23/12/1998"). As a result, unloading from an Informix database for loading into an Genero db is not supported.

ODIADS047 - The USER constant

Both Informix and Genero db provide the USER constant, which identifies the current user connected to the database server. However, there is a difference:

- Informix returns the user identifier as defined in the operating system, where it can be case-sensitive (UNIX) or not (NT).
- Genero db returns the user identifier that is stored in the database. By default, Genero db converts the user name to uppercase letters if you do not put the user name in double quotes when creating it.

This is important if your application stores user names in database records (for example, to audit data modifications). You can, for example, connect to Genero db with the name 'scott', and perform the following SQL operations:

```
(1) INSERT INTO mytab ( creator, comment )
        VALUES ( USER, 'example' );
(2) SELECT * FROM mytab
         WHERE creator = 'scott';
```

The first command inserts 'SCOTT' (in uppercase letters) in the author column. The second statement will not find the row.

Solution:

When creating a user in Genero db, you can put double quotes around the user name in order to force Genero db to store the given user identifier as is:

```
CREATE USER "scott" IDENTIFIED BY <pswd>
```

To verify the user names defined in Genero db, connect as SYSTEM and list the records of the ALL_USERS table as follows:

```
CREATE USER john IDENTIFIED BY <pswd>

SELECT user_name FROM table_of_users

USER_NAME
-----------
SYSTEM
JOHN
scott
```

ODIADS051 - Setup database statistics

Informix provides a special instruction to compute database statistics in order to help the optimizer determine the best query execution plan:

```
UPDATE STATISTICS ...
```

Genero db provides the following instruction to collect statistics:

```
SET GATHERSTATS tablename
```

Solution:

Replace the UPDATE STATISTICS by multiple SET GATHERSTATS statements (one for each table)

ODIADS053 - The ALTER TABLE instruction

Informix and Genero db have different implementations of the ALTER TABLE instruction. For example, Informix allows you to use multiple ADD clauses separated by commas; this is not supported by Genero db.

Informix:

```
     ALTER TABLE customer ADD(col1 INTEGER), ADD(col2 CHAR(20))
```

Genero db:

```
     ALTER TABLE customer ADD COLUMN col1 INTEGER ADD COLUMN col2
CHAR(20)
```

Solution:

Warning: No automatic conversion is done by the database interface. There is no real standard for this instruction (that is, no common syntax for all database servers). Read the SQL documentation and review the SQL scripts or the BDL programs in order to use the database server-specific syntax for ALTER TABLE.

ODIADS054 - SQL Interruption

With Informix, it is possible to interrupt a long running query if the SQL INTERRUPT ON option is set by the Genero program. The database server returns SQLCODE -213, which can be trapped to detect a user interruption.

```
MAIN
  DEFINE n INTEGER
  DEFER INTERRUPT
  OPTIONS SQL INTERRUPT ON
  DATABASE test1
  WHENEVER ERROR CONTINUE
  -- Start long query (self join takes time)
  -- From now on, user can hit CTRL-C in TUI mode to stop the query
  SELECT COUNT(*) INTO n FROM customers a, customers b
       WHERE a.cust_id <> b.cust_id
  IF SQLCA.SQLCODE == -213 THEN
     DISPLAY "Statement was interrupted by user..."
     EXIT PROGRAM 1
  END IF
  WHENEVER ERROR STOP
```

```
  ...
END MAIN
```

Genero db 3.80 supports SQL Interruption in a similar way as Informix. The db client must issue an SQLCancel() ODBC call to interrupt a query.

Solution:

The Genero db database driver supports SQL interruption and converts the native SQL error code -30005 to the Informix error code -213.

Warning: Make sure you have Genero db 3.80 or higher installed. Older versions do not support SQL interruption.

Note that when writing these lines, Genero db 3.80 does not support interruption of a DDL statement or an SQL statement waiting for a lock to be released (such as SELECT FOR UPDATE). Those limitations should be removed in a later Genero db version.

ODIADS100 - Data type conversion table

Informix Data Types	Genero db Data Types
BIGINT	BIGINT
BLOB	BLOB
BOOLEAN	**CHAR(1)**
BYTE	BYTE (= BLOB)
CHAR(n)	CHAR(n)
CHARACTER VARYING(n,m)	VARCHAR(n,m) (= VARCHAR (n))
CHARACTER(n)	CHARACTER(n) (= CHAR (n))
CLOB	CLOB
DATE	DATE
DATETIME HOUR TO SECOND	**TIME**
DATETIME x TO y (not HOUR TO SECOND)	**TIMESTAMP**
DEC	DECIMAL
DECIMAL(p)	**DOUBLE**
DECIMAL(p,s)	DECIMAL(p,s)**! p<=15**
DOUBLE	DOUBLE
DOUBLE PRECISION	DOUBLE PRECISION (= DOUBLE)
FLOAT(n)	FLOAT(n) (= DOUBLE)
INT	INT
INT8	INT8 (= BIGINT)

INTEGER	INTEGER
INTERVAL x TO y	**CHAR(50)**
MONEY(p,s)	**DECIMAL(p,s)! p<=15**
NCHAR(n)	NCHAR(n)
NUMERIC(p,s)	NUMERIC(p,s)
NVARCHAR(n)	NVARCHAR(n)
REAL	REAL (= DOUBLE)
SERIAL	SERIAL
SERIAL8	SERIAL8
SMALLFLOAT	SMALLFLOAT (= DOUBLE)
SMALLINT	SMALLINT
TEXT	TEXT (= CLOB)
VARCHAR(n,m)	VARCHAR(n,m) (= VARCHAR(n))
VARCHAR(n)	VARCHAR(n)

ODIADS101 - GeneroDB Sql Error management

For a general idea on how Four J's ODI handles SqlErrors in the BDL language, check the following links:

Error Handling
SQL Errors
STATUS
SQLSTATE
SQLERRMESSAGE
SQLCA Record
Portability: SQLCA

GeneroDB has some specific rules that need to be highlighted:

1. Normally, you should use SQLSTATE, because this is the ANSI/ISO standard. SQLSTATE defines unified codes so that you can write programs for different RDBMS. However, only some RDBMS - and only in their recent versions - support SQLSTATE. GeneroDB is not one of them. So we would suggest that you keep using SQLCA.SQLCODE (which gives the Informix error code).

    - The native Genero db error code is stored in **SQLCA.SQLERRD[2]** register
    - The native Genero db error message is stored in **SQLERRMESSAGE** operator
    - The generic Informix error code is stored in **SQLCA.SQLCODE** register

So far, we convert the following native Genero db errors:

Genero db SQLCA.SQLERRD[2]	Informix SQLCA.SQLCODE	Error description (SQLERRMESSAGE)
-1	-201	syntax error
-3	-206	table not found
-4	-201	syntax error
-6	-217	column not found
-17	-743	object exists
-24	-201	syntax error
-35	-236	cols/vals mismatch
-10012	-768	internal / untranslatable error
-10013	-768	internal / untranslatable error
-10014	-213	SQL interrupted
-30004	-263	*Cannot wait on another session.*
-30005	-213	SQL interrupted
-60001	-268	*Uniqueness constraint violation.*
-80002	-387	no connect permission

2. There's no one-to-one error conversion even from Informix to GeneroDB. If you take a look at the exhaustive list of Informix error:s http://www-306.ibm.com/software/data/informix/pubs/library/ierrors.htm

   Many different formerly Informix errors can be returned by the ODI driver; For example, fetch on open cursor, commit on unopened transaction, ...

   If an unexpected problem happens on the ODBC driver end (could not create temporary table, ...), the driver will return:

SQLCA.SQLCODE	SQLERRMESSAGE
-768	Internal error in routine routine-name

List of known errors the GeneroDB ODBC driver can return:

SQLCA.SQLERRD[2]	SQLERRMESSAGE	Cause
-213	Query canceled	Long running query interrupted by user
-254	Too many or too few host variables given.	PREPARE s FROM "insert into t values (?,?)" EXECUTE s USING x,y,z (z en trop)
-255	Not in transaction.	OPEN insert cursor

		without BEGIN WORK
-284	A subquery has not returned exactly one row.	SELECT * INTO ... FROM tab, returns more than one row
-400	Fetch attempted on unopened cursor.	FETCH on cursor not opened
-404	The cursor or statement is not available.	OPEN cursor after a FREE
-410	Prepare statement failed or was not executed.	EXECUTE a statement where PREPARE has failed
-413	Insert attempted on unopened cursor.	PUT on insert cursor not opened
-481	Invalid statement name or statement was not prepared.	EXECUTE a statement without PREPARE
-482	Invalid operation on a non-SCROLL cursor.	FETCH LAST/PREV/... on non SCROLL cursor
-526	Updates are not allowed on a scroll cursor.	SELECT FOR UPDATE with SCROLL cursor
-535	Already in transaction.	BEGIN WORK x2
-6370	Unsupported SQL feature.	CREATE DATABASE, SET CONNECTION DORMANT, CREATE PROCEDURE FROM, DATABASE IN EXCLUSIVE MODE, CONNECT TO @server, ...

3. If an unknown error comes from the DB Server and therefore is not mapped as an Informix error, you'll get:

SQLCA.SQLCODE	SQLERRMESSAGE
-6372	General SQL error, check SQLCA.SQLERRD[2]

As said previously, you can always check the native SQL error in the SQLCA.SQLERRD[2] register or in the FGLSQLDEBUG output.

Example:

Here is a suggestion to trap an unknown error (-768 or -6372):

```
MAIN
  WHENEVER ERROR CONTINUE
  CONNECT TO dsn_connectstring
  IF STATUS <> 0 THEN
```

```
         DISPLAY "ERROR: Connection to the database failed."
         DISPLAY SQLCA.SQLCODE, ": ", SQLCA.SQLERRD[2], "-" ,
SQLERRMESSAGE
         EXIT PROGRAM 1
      END IF
      WHENEVER ERROR STOP
    END MAIN
```

ODI Adaptation Guide For DB2 UDB 7.x, 8.x, 9x

Runtime configuration

- Install DB2 and create a database
- Prepare the runtime environment

Database concepts

- Database concepts
- Data storage concepts
- Data consistency and concurrency management
- Transactions handling
- Defining database users
- Setting privileges

Data dictionary

- CHARACTER data types
- NUMERIC data types
- DATE and DATETIME data types
- INTERVAL data type
- SERIAL data type
- ROWIDs
- Very large data types
- National character data types
- Constraints
- Triggers
- Stored procedures
- Name resolution of SQL objects
- Setup database statistics
- The ALTER TABLE instruction
- Data type conversion table

Data manipulation

- Reserved words
- Outer joins
- Transactions handling

- Temporary tables
- Substrings in SQL
- Name resolution of SQL objects
- String delimiters and object name delimiters
- Getting one row with SELECT
- MATCHES and LIKE conditions
- SQL functions and constants
- Querying system catalog tables
- The GROUP BY clause
- The star in SELECT statements
- The LENGTH() function

BDL programming

- SERIAL data type
- INFORMIX specific SQL statements in BDL
- INSERT cursors
- Cursors WITH HOLD
- SELECT FOR UPDATE
- SQL parameters limitation
- The LOAD and UNLOAD instructions
- SQL Interruption

Connecting to DB2 OS/400

- DB2 Architecture on OS/400
- Log in to the AS/400 server

- Collection (Schema) Creation
- Source Physical File Creation
- Trigger Creation
- Permission Definition
- Relational DB Directory Entry Creation
- DB2 Client Configuration on Windows
- Differences Between DB2 UNIX & DB2 OS/400
- Naming Conventions

Runtime configuration

Install DB2 and create a database

1. Install the IBM DB2 Universal Server on your database server.
2. Create a DB2 database entity: *dbname*
3. Declare a database user dedicated to your application: the **application administrator**. This user will manage the database schema of the application (all tables will be owned by it).
4. Give all requested database administrator privileges to the **application administrator**.
5. If you plan to use temporary table emulation, you must setup the database for DB2 global temporary tables (create a user temporary tablespace and grant privileges to all users).

   See issue ODIDB2017 for more details.
6. Connect as the application administrator:

   ```
   $ db2 "CONNECT TO dbname USER appadmin USING password"
   ```
7. Create the **application tables**. Do not forget to convert Informix data types to DB2 data types. See issue ODIDB2100 for more details.
8. If you plan to use SERIAL column emulation, you must prepare the database. See issue ODIDB2005 for more details.

Prepare the runtime environment

1. If you want to connect to a remote DB2 server, the **IBM DB2 Client Application Enabler** must be installed and configured on the computer running the BDL applications. You must declare the data source set up as follows:

   1. Login as root.
   2. Create a user dedicated to the db2 client instance environment, for example, "db2cli1".
   3. Create a client instance environment with the **db2icrt** tool as in following example:

      ```
      # db2dir/instance/db2icrt -a server -s client instance-user
      ```
   4. Login as the instance user (environment should be set automatically, verify DB2DIR).
   5. Catalog the remote server node:

      ```
      # db2 "catalog tcpip node db2node remote hostname server tcp-service"
      ```

6. Catalog the remote database:
```
# db2 "catalog database datasource
at node db2node authentication server"
```
7. Test the connection to the remote database:
```
# db2 "connect to datasource user
dbuser using password"
```
(where *dbuser* is a database user declared on the remote database server).

See IBM DB2 documentation for more details.

2. **IMPORTANT WARNING**: You may need to set the PATCH2=15 configuration parameter in the **DB2CLI.INI** file, if you have a non-English environment; otherwise DECIMAL values will not be interpreted as expected:
```
[datasource]
PATCH2=15
```
For more details, see the DB2 README.TXT file in the SQLLIB directory.
3. Check the database locale settings (**DB2CODEPAGE**, etc). The DB locale must match the locale used by the runtime system (**LANG**).
4. Setup the **fglprofile** entries for database connections.
5. Define the database schema selection if needed. Use the following entry to define the database schema to be used by the application. The database interface will automatically perform a "SET SCHEMA <name>" instruction to switch to a specific schema:

```
dbi.database.dbname.schema = 'name'
```

Here *dbname* identifies the database name used in the BDL program (DATABASE *dbname*) and *name* is the schema name to be used in the SET SCHEMA instruction. If this entry is not defined, no "SET SCHEMA" instruction is executed and the current schema defaults to the user's name.
6. In order to connect to IBM DB2, you must have a database driver "**dbmdb2***" installed.

ODIDB2001 - DATE and DATETIME data types

INFORMIX provides two data types to store date and time information:

- **DATE** = for year, month and day storage.
- **DATETIME** = for year to fraction(1-5) storage.

IBM DB2 provides only one data type to store dates :

- **DATE** = for year, month, day storage.
- **TIME** = for hour, minute, second storage.
- **TIMESTAMP** = for year, month, day, hour, minute, second, fraction storage.

String representing date time information:

INFORMIX is able to convert quoted strings to DATE / DATETIME data if the string content matches environment parameters (i.e. DBDATE, GL_DATETIME). As INFORMIX, IBM DB2 can convert quoted strings to dates, times or timestamps. Only one format is possible: 'yyyy-mm-dd' for dates, 'hh:mm:ss' for times and 'yyyy-mm-dd hh:mm:ss:f' for timestamps.

Date time arithmetic:

- INFORMIX supports date arithmetic on DATE and DATETIME values. The result of an arithmetic expression involving dates/times is a number of days when only DATEs are used and an INTERVAL value if a DATETIME is used in the expression.
- In IBM DB2, the result of an arithmetic expression involving DATE values is a NUMBER of days, the decimal part is the fraction of the day (0.5 = 12H00, 2.00694444 = (2 + (10/1440)) = 2 days and 10 minutes)).
- INFORMIX automatically converts an integer to a date when the integer is used to set a value of a date column. IBM DB2 does not support this automatic conversion.
- Complex DATETIME expressions (involving INTERVAL values for example) are INFORMIX specific and have no equivalent in IBM DB2.

Solution:

DB2 has the same **DATE** data type as INFORMIX (year, month, day). So you can use DB2 DATE data type for Informix DATE columns.

DB2 **TIME** data type can be used to store INFORMIX DATETIME HOUR TO SECOND values. The database interface makes the conversion automatically.

INFORMIX DATETIME values with any precision from YEAR to FRACTION(5) can be stored in DB2 **TIMESTAMP** columns. The database interface makes the conversion automatically. Missing date or time parts default to 1900-01-01 00:00:00.0. For example, when using a DATETIME HOUR TO MINUTE with the value of "11:45", the DB2 TIMESTAMP value will be "1900-01-01 11:45:00.0".

Warning: Using integers as a number of days in an expression with dates is not supported by IBM DB2. Check your code to detect where you are using integers with DATE columns.

Warning: Literal DATETIME and INTERVAL expressions (i.e. DATETIME (1999-10-12) YEAR TO DAY) are not converted.

Warning: It is strongly recommended that you use BDL variables in dynamic SQL statements instead of quoted strings representing DATEs. For example :

```
   LET stmt = "SELECT ... FROM customer WHERE creat_date >'", adate,"'"
```

is not portable, use a question mark place holder instead and OPEN the cursor USING adate:

```
   LET stmt = "SELECT ... FROM customer WHERE creat_date > ?"
```

Warning: DATE arithmetic expressions using SQL parameters (USING variables) are not fully supported. For example: `"SELECT ... WHERE datecol < ? + 1"` generate an error at PREPARE time.

Warning: SQL Statements using expressions with TODAY / CURRENT / EXTEND must be reviewed and adapted to the native syntax.

ODIDB2003 - Reserved words

Even if IBM DB2 allows SQL reserved keywords as SQL object names (`"create table table ( column int )"`), you should take care in your existing database schema and check that you do not use DB2 SQL words. An example of a common word which is part of DB2 SQL grammar is '**alias**'.

Solution:

See IBM DB2 documentation for reserved keywords.

ODIDB2004 - ROWIDs

INFORMIX rowids are implicit INTEGER columns managed by the database server.

IBM DB2 tables have no ROWIDs.

Solution:

If the BDL application uses ROWIDs, the program logic should be reviewed in order to use the real primary keys (usually, serials which can be supported).

However, if your existing INFORMIX application depends on using ROWID values, you can use the DB2 GENERATE_UNIQUE built-in function, or the IDENTITY attribute of the INTEGER or BIGINT data types, to simulate this functionality.

All references to SQLCA.SQLERRD[6] must be removed because this variable will not hold the ROWID of the last INSERTed or UPDATEd row when using the IBM DB2 interface.

ODIDB2005 - SERIAL data type

INFORMIX SERIAL data type and automatic number production:

- The table column must be of type SERIAL.
- To generate a new serial, **no value** or **zero** value is given to the INSERT statement:
  ```
  INSERT INTO tab1 ( c ) VALUES ( 'aa' )
  INSERT INTO tab1 ( k, c ) VALUES ( 0, 'aa' )
  ```
- After INSERT, the new SERIAL value is provided in `SQLCA.SQLERRD[2]`.

INFORMIX allows you to insert rows with a value different from zero for a serial column. Using an explicit value will automatically increment the internal serial counter, to avoid conflicts with future INSERTs that are using a zero value:

```
CREATE TABLE tab ( k SERIAL ); --> internal counter = 0
INSERT INTO tab VALUES ( 0 );  --> internal counter = 1
INSERT INTO tab VALUES ( 10 ); --> internal counter = 10
INSERT INTO tab VALUES ( 0 );  --> internal counter = 11
DELETE FROM tab;               --> internal counter = 11
INSERT INTO tab VALUES ( 0 );  --> internal counter = 12
```

IBM DB2 has no equivalent for INFORMIX SERIAL columns.

DB2 version 7.1 supports IDENTITY columns:

```
CREATE TABLE tab ( k INTEGER GENERATED ALWAYS AS IDENTITY);
```

To get the last generated IDENTITY value after an INSERT, DB2 provides the following function:

```
IDENTITY_VAL_LOCAL( )
```

DB2 version 8.1 supports SEQUENCES:

```
CREATE SEQUENCE sq1 START WITH 100;
```

To create a new sequence number, you must use the "NEXTVAL FOR" operator:

```
INSERT INTO table VALUES ( NEXTVAL FOR sq1, ... )
```

To get the last generated sequence number, you must use the "PREVVAL FOR" operator:

```
SELECT PREVVAL FOR sq1 ...
```

Solution:

You are free to use **IDENTITY columns** (1) or **insert triggers using SEQUENCES** (2). The first solution is faster, but does not allow explicit serial value specification in insert statements; the second solution is slower but allows explicit serial value specification. You can start to use the second solution to make unmodified 4gl programs work on DB2, but you should update your code to use native IDENTITY columns for performance.

Warning: The second method (trigseq) works only with DB2 version 8 and higher.

The serial emulation type is defined by the following FGLPROFILE entry:

```
   dbi.database.<dbname>.ifxemul.datatype.serial.emulation =
{"native"|"trigseq"}
```

The '**native**' value defines the IDENTITY column technique and the '**trigseq**' defines the trigger technique.

This entry must be used with:

```
   dbi.database.<dbname>.ifxemul.datatype.serial = {true|false}
```

If the `datatype.serial` entry is set to false, the emulation method specification entry is ignored.

Warning: When no entry is specified, the default is SERIAL emulation enabled with '**native**' method (IDENTITY-based).

1. Using IDENTITY columns

In database creation scripts, all SERIAL[(n)] data types must be converted by hand to INTEGER GENERATED ALWAYS AS IDENTITY[(START WITH n, INCREMENT BY 1)].

Tables created from the BDL programs can use the SERIAL data type : When a BDL program executes a CREATE [TEMP] TABLE with a SERIAL column, the database interface automatically converts the "SERIAL[(n)]" data type to an IDENTITY specification.

In BDL, the new generated SERIAL value is available from the SQLCA.SQLERRD[2] variable. This is supported by the database interface which performs a call to the IDENTITY_VAL_LOCAL() function.

Warning: Since IBM DB2 does not allow you to specify the value of IDENTITY columns, it is mandatory to convert all INSERT statements to remove the SERIAL column from the list.
For example, the following statement:

```
   INSERT INTO tab (col1,col2) VALUES (0, p_value)
```

must be converted to:

```
   INSERT INTO tab (col2) VALUES (p_value)
```

2. Using triggers with the SEQUENCE

In database creation scripts, all SERIAL[(n)] data types must be converted to INTEGER data types and you must create a sequence and a trigger for each table using a SERIAL. To know how to write those triggers, you can create a small Genero program that creates a table with a SERIAL column. Set the FGLSQLDEBUG environment variable and run the program. The debug output will show you the native SQL commands to create the sequence and the trigger.

Tables created from the BDL programs can use the SERIAL data type : When a BDL program executes a CREATE [TEMP] TABLE with a SERIAL column, the database interface automatically converts the "SERIAL[(n)]" data type to "INTEGER" and creates the sequence and the insert trigger.

Warning: IBM DB2 performs NOT NULL data controls before the execution of triggers. If the serial column must be NOT NULL (for example, because it is part of the primary key), you cannot specify a NULL value for that column in INSERT statements.
For example, the following statement :

```
INSERT INTO tab VALUES (NULL,p_value)
```

must be converted to :

```
INSERT INTO tab (col2) VALUES (p_value)
```

Warning: IBM DB2 triggers are not automatically dropped when the corresponding table is dropped. They become ***inoperative*** instead. Database administrators must take care of this behavior when managing schemas.

Warning: With IBM DB2, INSERT statements using NULL for the SERIAL column will produce a new serial value, not a NULL like INFORMIX does :

```
INSERT INTO tab ( col_serial, col_data ) VALUES ( NULL, 'data' )
```

This behavior is mandatory in order to support INSERT statements which do not use the serial column :

```
INSERT INTO tab (col_data) VALUES ('data')
```

Check if your application uses tables with a SERIAL column that can contain a NULL value.

Warning: With DB2, trigger creation is not allowed on temporary tables. Therefore, the "**trigseq**" method cannot work with temporary tables using serials.

ODIDB2006 - Outer joins

The syntax of OUTER joins is very different in INFORMIX and IBM DB2:

In INFORMIX SQL, outer tables are defined in the FROM clause with the **OUTER** keyword:

```
SELECT ... FROM cust, OUTER(order)
 WHERE cust.key = order.custno
SELECT ... FROM cust, OUTER(order,OUTER(item))
 WHERE cust.key = order.custno
   AND order.key = item.ordno
   AND order.accepted = 1
```

IBM DB2 supports the ANSI outer join syntax:

```
SELECT ... FROM cust LEFT OUTER JOIN order
                    ON cust.key = order.custno
```

```
SELECT ...
  FROM cust LEFT OUTER JOIN order
                LEFT OUTER JOIN item
                ON order.key = item.ordno
            ON cust.key = order.custno
 WHERE order.accepted = 1
```

See the IBM DB2 SQL reference for a complete description of the syntax.

Solution:

The IBM DB2 interface can convert most INFORMIX OUTER specifications to IBM DB2 outer joins.

Prerequisites :

1. In the FROM clause, the main table must be the first item and the outer tables must figure from left to right in the order of outer levels.
   Example which does not work : "FROM OUTER(tab2), tab1".
2. The outer join in the WHERE clause must use the table name as prefix.
   Example : "WHERE tab1.col1 = tab2.col2".

Restrictions :

1. Additional conditions on outer table columns cannot be detected and therefore are not supported :
   Example : "... FROM tab1, OUTER(tab2) WHERE tab1.col1 = tab2.col2 AND tab2.colx > 10".
2. Statements composed by 2 or more SELECT instructions using OUTERs are not supported.
   Example : "SELECT ... UNION SELECT" or "SELECT ... WHERE col IN (SELECT...)"

Remarks :

1. Table aliases are detected in OUTER expressions.
   OUTER example with table alias : "OUTER(tab1 alias1)".
2. In the outer join, <outer table>.<col> can be placed on both right or left sides of the equal sign.
   OUTER join example with table on the left : "WHERE outertab.col1 = maintab.col2 ".
3. Table names detection is not case-sensitive.
   Example : "SELECT ... FROM tab1, TAB2 WHERE tab1.col1 = tab2.col2".
4. Temporary tables are supported in OUTER specifications.

ODIDB2007a - Database concepts

As INFORMIX, an IBM DB2 database server can handle more than one database entity. INFORMIX servers have an ID (INFORMIXSERVER) and databases are identified by name. IBM DB2 instances are identified by the DB2INSTANCE environment variable and databases have to be cataloged as data sources (see IBM DB2 documentation for more details).

ODIDB2008a - Data consistency and concurrency management

Data consistency involves readers that want to access data currently modified by writers and **concurrency data access** involves several writers accessing the same data for modification. **Locking granularity** defines the amount of data concerned when a lock is set (row, page, table, ...).

INFORMIX

INFORMIX uses a locking mechanism to manage data consistency and concurrency. When a process modifies data with UPDATE, INSERT or DELETE, an exclusive lock is set on the affected rows. The lock is held until the end of the transaction. Statements performed outside a transaction are treated as a transaction containing a single operation and therefore release the locks immediately after execution. SELECT statements can set shared locks according the isolation level. In case of locking conflicts (for example, when two processes want to acquire an exclusive lock on the same row for modification or when a writer is trying to modify data protected by a shared lock), the behavior of a process can be changed by setting the lock wait mode.

Control :

- Isolation level : SET ISOLATION TO ...
- Lock wait mode : SET LOCK MODE TO ...
- Locking granularity : CREATE TABLE ... LOCK MODE {PAGE|ROW}
- Explicit locking : SELECT ... FOR UPDATE

Defaults :

- The default isolation level is **read committed**.
- The default lock wait mode is "not wait".
- The default locking granularity is on per page.

IBM DB2

As in INFORMIX, IBM DB2 uses locks to manage data consistency and concurrency. The database manager sets exclusive locks on the modified rows and shared locks when data is read, according to the isolation level. The locks are held until the end of the

transaction. When multiple processes want to access the same data, the latest processes must wait until the first finishes its transaction. The lock granularity is at the row or table level. For more details, see DB2's Administration Guide, "Application Consideration".

Control :

- Lock wait mode : Always WAIT. Only the Lock Timeout can be changed, but this is a global database parameter.
- Isolation level : Can be set through an API function call or with a database client configuration parameter.
- Locking granularity : Row level or Table level.
- Explicit locking : SELECT ... FOR UPDATE

Defaults :

- The default isolation level is Cursor Stability (readers cannot see uncommitted data, no shared lock is set when reading data).

Solution:

For portability, it is recommended that you work with INFORMIX in the read committed isolation level, to make processes wait for each other (lock mode wait) and to create tables with the "lock mode row" option.

See INFORMIX and IBM DB2 documentation for more details about data consistency, concurrency and locking mechanisms.

ODIDB2008b - SELECT FOR UPDATE

A lot of BDL programs use pessimistic locking in order to prevent several users editing the same rows at the same time.

```
DECLARE cc CURSOR FOR
      SELECT ... FROM tab WHERE ... FOR UPDATE
OPEN cc
FETCH cc <-- lock is acquired
...
CLOSE cc <-- lock is released
```

In both INFORMIX and DB2, locks are released when closing the cursor or when the transaction ends.

DB2's locking granularity is at the row level.

To control the behavior of the program when locking rows, INFORMIX provides a specific instruction to set the wait mode :

```
SET LOCK MODE TO { WAIT | NOT WAIT | WAIT seconds }
```

The default mode is NOT WAIT. This as an INFORMIX specific SQL statement.

Warning: DB2 has no equivalent for "SET LOCK MODE TO NOT WAIT". The "**Lock timeout**" can be changed but this is a database parameter (global to all processes)!

Solution:

Warning : The database interface is based on an emulation of an Informix engine using transaction logging. Therefore, opening a SELECT ... FOR UPDATE cursor declared outside a transaction will raise an SQL error -255 (not in transaction).

You must review the program logic if you use pessimistic locking because it is based on the NOT WAIT mode which is not supported by IBM DB2.

ODIDB2009a - Transactions handling

INFORMIX and IBM DB2 handle transactions differently. The differences in the transactional models can affect the program logic.

INFORMIX native mode (non ANSI) :

- DDL statements can be executed (and canceled) in transactions.
- Transactions must be started with BEGIN WORK. Statements executed outside of a transaction are automatically committed.

IBM DB2 :

- DDL statements can be executed (and canceled) in transactions.
- Beginning of transactions are implicit; two transactions are delimited by COMMIT or ROLLBACK.

Transactions in stored procedures : avoid using transactions in stored procedures to allow the client applications to handle transactions, in accordance with the transaction model.

Solution:

The INFORMIX behavior is simulated with an auto-commit mode in the IBM DB2 interface. A switch to the explicit commit mode is done when a BEGIN WORK is performed by the BDL program.

Regarding the transaction control instructions, the BDL applications do not have to be modified in order to work with IBM DB2.

See also ODIDB2008b

ODIDB2011a - CHARACTER data types

As in INFORMIX, IBM DB2 provides the CHAR and VARCHAR data types to store character data.

INFORMIX CHAR type can store up to **32767** characters and the VARCHAR data type is limited to **255** characters.

IBM DB2 CHAR are limited to **254** characters and VARCHAR can be **32672** characters in size.

Solution:

The database interface supports character string variables in SQL statements for input (BDL USING) and output (BDL INTO).

Warning: Check that your database schema does not use CHAR or VARCHAR types with a length exceeding the DB2 limits.

ODIDB2012 - Constraints

Constraint naming syntax:

Both INFORMIX and BD2 support primary key, unique, foreign key, default and check constraints. But the constraint naming syntax is different : DB2 expects the "CONSTRAINT" keyword **before** the constraint specification, and INFORMIX expects it **after**.

UNIQUE constraint example:

INFORMIX	IBM DB2

`CREATE TABLE scott.emp (`
`...`
`empcode CHAR(10) UNIQUE`
`  [CONSTRAINT pk_emp],`
`...`
	`CREATE TABLE scott.emp (`
	`...`
	`empcode CHAR(10)`
	`  [CONSTRAINT pk_emp]`
	`UNIQUE,`
	`...`

Primary keys:

Like INFORMIX, DB2 creates an index to enforce PRIMARY KEY constraints (some RDBMS do not create indexes for constraints). Using "CREATE UNIQUE INDEX" to define unique constraints is obsolete (use primary keys or a secondary key instead).

Warning: DB2 primary key constraints do not allow NULLs; make sure your tables do not contain NULLs in the primary key columns.

Unique constraints:

Like INFORMIX, DB2 creates an index to enforce UNIQUE constraints (some RDBMS do not create indexes for constraints).

Warning: DB2 unique constraints do not allow NULLs; make sure your tables do not contain NULLs in the unique columns.

Foreign keys:

Both INFORMIX and DB2 support the ON DELETE CASCADE option.

Check constraints:

Warning : The check condition may be any valid expression that can be evaluated to TRUE or FALSE, including functions and literals. You must verify that the expression is not INFORMIX specific.

Null constraints:

INFORMIX and DB2 support not null constraints, but INFORMIX does not allow you to give a name to "NOT NULL" constraints.

Solution:

Constraint naming syntax:

The database interface does not convert constraint naming expressions when creating tables from BDL programs. Review the database creation scripts to adapt the constraint naming clauses for DB2.

ODIDB2013 - Triggers

INFORMIX and IBM DB2 provide triggers with similar features, but the trigger creation syntax and the programming languages are totally different.

INFORMIX triggers define which stored procedures must be called when a database event occurs (before | after insert | update | delete ...), while IBM DB2 triggers can hold a procedural block.

IBM DB2 provides specific syntax to define triggers. See documentation for more details.

Solution:

INFORMIX triggers must be converted to IBM DB2 triggers "by hand".

ODIDB2014 - Stored procedures

Both INFORMIX and IBM DB2 support stored procedures and user functions, but the programming languages are totally different.

INFORMIX implements the **SPL** language, while DB2 allows you to write stored procedures or user defined functions in the **DB2 SQL** or with an external language, such as **JAVA**, **C** or **C++**.

Solution:

INFORMIX stored procedures must be converted to IBM DB2 "by hand".

ODIDB2016a - Defining database users

INFORMIX users are defined at the operating system level, they must be members of the 'informix' group, and the database administrator must grant CONNECT, RESOURCE or DBA privileges to those users.

IBM DB2 users are operating system users with a specific DB2 environment. The database administrator must grant the CONNECT authority to these users.

Database *authorities* involve actions on a database as a whole. When a database is created, some authorities are automatically granted to anyone who accesses the database. For example, CONNECT, CREATETAB, BINDADD and IMPLICIT_SCHEMA authorities are granted to all users. Database *privileges* involve actions on specific objects within the database. When a database is created, some privileges are automatically granted to anyone who accesses the database. For example, SELECT privilege is granted on catalog views and EXECUTE and BIND privilege on each successfully bound utility is granted to all users.

Together, privileges and authorities act to control access to an instance and its database objects. Users can access only those objects for which they have the appropriate authorization, that is, the required privilege or authority.

Warning: As in INFORMIX, DB2 user names that connect to the database server must be a maximum of **eight** characters long.

Solution:

Setup the IBM DB2 environment for each user as described in the documentation.

ODIDB2016b - Setting privileges

INFORMIX and IBM DB2 user privileges management is quite similar.

IBM DB2 provides user groups to define.

INFORMIX users must have at least the CONNECT privilege to access the database:

```
GRANT CONNECT TO (PUBLIC|user)
```

IBM DB2 users must have at least the CONNECT authority to access the database.
font face="Courier New"> GRANT **CONNECT** ON DATABASE TO (PUBLIC|user|group)

Solution:

Make sure DB2 users have the right privileges to access the database.

See also Temporary Tables

ODIDB2017 - Temporary tables

INFORMIX temporary tables are created through the **`CREATE TEMP TABLE`** DDL instruction or through a **`SELECT ... INTO TEMP`** statement. Temporary tables are automatically dropped when the SQL session ends, but they can also be dropped with the DROP TABLE command. There is no name conflict when several users create temporary tables with the same name.

Remark : BDL reports create a temporary table when the rows are not sorted externally (by the source SQL statement).

INFORMIX allows you to create indexes on temporary tables. No name conflict occurs when several users create an index on a temporary table by using the same index identifier.

IBM DB2 7 supports the DECLARE GLOBAL TEMPORARY TABLE instruction. Native DB2 temporary tables are quite similar to Informix temporary tables with some exceptions:

- A 'user temporary table space' must exist for the database.
- Users must have 'USE' privilege on a 'user temporary table space'.

- For usage, the temporary table name must be prefixed by 'SESSION'.
- No constraints or indexes can be created on temporary tables.

For more details, see the DB2 documentation.

Solution:

In accordance with some prerequisites, temporary tables creation in BDL programs can be supported by the database interface.

How does it work ?

- INFORMIX specific statements involving temporary table creation are automatically converted to IBM DB2 "DECLARE GLOBAL TEMPORARY TABLE" statements.
- Once the temporary table has been created, all other SQL statements performed in the current SQL session are parsed to add the SESSION prefix to the table name automatically.

Prerequisites:

- DB2 prerequisites to create global temporary tables. See DB2 documentation for more details.

Limitations:

- Tokens matching the original table names are converted to unique names in all SQL statements. Make sure you are not using a temp table name for other database objects, like columns. The following example illustrates this limitation :

```
CREATE TEMP TABLE tmp1 ( col1 INTEGER, col2 CHAR(20) )
SELECT tmp1 FROM table_x WHERE ...
```

- **Warning:** Only the 'native' serial emulation mode is supported with temporary tables. See the issue about SERIALs for more details.

ODIDB2018 - Substrings in SQL

INFORMIX SQL statements can use subscripts on columns defined with the character data type:

```
    SELECT ... FROM tab1 WHERE col1[2,3] = 'RO'
    SELECT ... FROM tab1 WHERE col1[10]  = 'R'   -- Same as
col1[10,10]
    UPDATE tab1 SET col1[2,3] = 'RO' WHERE ...
    SELECT ... FROM tab1 ORDER BY col1[1,3]
```

.. while IBM DB2 provides the SUBSTR( ) function, to extract a substring from a string expression:

```
SELECT .... FROM tab1 WHERE SUBSTR(col1,2,2) = 'RO'
SELECT SUBSTR('Some text',6,3) FROM DUAL       -- Gives 'tex'
```

Solution:

You must replace all Informix col[x,y] expressions by SUBSTR(col,x,y-x+1).

Warning : In UPDATE instructions, setting column values through subscripts will produce an error with IBM DB2:

```
UPDATE tab1 SET col1[2,3] = 'RO' WHERE ...
```

is converted to :

```
UPDATE tab1 SET SUBSTR(col1,2,3-2+1) = 'RO' WHERE ...
```

Warning: Column subscripts in ORDER BY expressions produce an error with IBM DB2:

```
SELECT ... FROM tab1 ORDER BY col1[1,3]
```

is converted to :

```
SELECT ... FROM tab1 ORDER BY SUBSTR(col1,1,3-1+1)
```

ODIDB2019 - Name resolution of database objects

Case sensitivity in object names:

INFORMIX database object names are not **case-sensitive** in non-ANSI databases.

```
CREATE TABLE Tab1 ( Key INT, Col1 CHAR(20) )
SELECT COL1 FROM TAB1
```

IBM DB2 database object names are case-sensitive. When a name is used without double quotes, it is automatically converted to uppercase letters. When using double quotes, the names are not converted:

```
CREATE TABLE tab1 ( Key INT, Col1 CHAR(20) )
   => Table name is "TAB1", column names are "KEY" and "COL1"

CREATE TABLE "Tab1" ( "Key" INT, "Col1" CHAR(20) )
   => Table name is "Tab1", column names are "Key" and "Col1"
```

The DB2 schema concept:

With non-ANSI INFORMIX databases, you do not have to give a schema name before the tables when executing an SQL statement.

```
SELECT ... FROM <table> WHERE ...
```

In an IBM DB2 database, tables always belong to a database **schema**. When executing a SQL statement, a schema name must be used as the high-order part of a two-part object name, unless the current schema corresponds to the table's schema.

The default (implicit) schema is the current user's name but it can be changed with the "**SET SCHEMA**" instruction.

Example: The table "TAB1" belongs to the schema "SCH1". User "MARK" (implicit schema is "MARK") wants to access "TAB1" in a SELECT statement :

```
SELECT ... FROM TAB1 WHERE ...
    => Error "MARK"."TAB1" is an undefined name. SQLSTATE=42704
SELECT ... FROM SCH1.TAB1 WHERE ...
    => OK.
SET SCHEMA SCH1
    => Changes the current schema to SCH1.
SELECT ... FROM TAB1 WHERE ...
    => OK.
```

DB2 provides "**aliases**", but they cannot be used to make a database object name public because aliases belong to schemas also.

Solution:

Case sensitivity in object names:

Avoid the usage of double quotes around the database object names. All names will be converted to uppercase letters.

The DB2 schema concept:

After a connection, the database interface can automatically execute a "SET SCHEMA <**name**>" instruction if the following FGLPROFILE entry is defined:

```
dbi.database.<dbname>.schema = "<name>"
```

Here <**dbname**> identifies the database name used in the BDL program (DATABASE **dbname**) and <**name**> is the schema name to be used in the SET SCHEMA instruction. If this entry is not defined, no "SET SCHEMA" instruction is executed and the current schema defaults to the user's name.

Examples:

```
dbi.database.stores.schema = "STORES1"
dbi.database.accnts.schema = "ACCSCH"
```

Warning: DB2 does not check the schema name when the SET SCHEMA instruction is executed. Setting a wrong schema name results in "undefined name" errors when performing subsequent SQL instructions like SELECT, UPDATE, INSERT.

In accordance with this automatic schema selection, you must create a DB2 schema for your application :

1. Connect as a user with the DBADM authority.
2. Create an administrator user dedicated to your application. For example, "STORESADM". Make sure this user has the IMPLICIT_SCHEMA privilege (this is the default in DB2).
3. Connect as the application administrator "STORESADM" to create all database objects (tables, indexes, ...). In our example, a "STORESADM" schema will be created implicitly and all database objects will belong to this schema.

As a second option you can create a specific schema with the following SQL command :
`CREATE SCHEMA "<`**name**`>" AUTHORIZATION "<`**appadmin**`>"`
See IBM DB2 manuals for more details about schemas.

Warning: Case sensitivity: When executing the "SET SCHEMA" instruction, the database interface does not use double quotes around the schema name (= name is converted to uppercase letters). Make sure that the schema name is created with uppercase letters in the database.

ODIDB2020 - String delimiters

The ANSI string delimiter character is the single quote ('string'). Double quotes are used to delimit database object names ("object-name").

Example: `WHERE "tabname"."colname" = 'a string value'`

INFORMIX allows double quotes as string delimiters, but IBM DB2 doesn't. This is important since many BDL programs use that character to delimit the strings in SQL commands.

Remark : This problem concerns only double quotes within SQL statements. Double quotes used in pure BDL string expressions are not subject of SQL compatibility problems.

Solution:

The IBM DB2 database interface can automatically replace all double quotes by single quotes.

Escaped string delimiters can be used inside strings like the following :

```
'This is a single quote : '''
'This is a single quote : \''
"This is a double quote : """
"This is a double quote : \""
```

Warning: Database object names cannot be delimited by double quotes because the database interface cannot determine the difference between a database object name and a quoted string!

For example, if the program executes the SQL statement:
```
   WHERE "tabname"."colname" = "a string value"
```
replacing all double quotes by single quotes would produce:
```
   WHERE 'tabname'.'colname' = 'a string value'
```
This would produce an error since `'tabname'.'colname'` is not allowed by IBM DB2.

Although double quotes are automatically replaced in SQL statements, you should use only single quotes to enforce portability.

ODIDB2021a - NUMERIC data types

INFORMIX provides several data types to store numbers :

INFORMIX Data Type	Description
SMALLINT	16 bit integer (-2^15 to 2^15)
INT/INTEGER	32 bit integer (-2^31 to 2^31)
DEC/DECIMAL(p)	Floating-point decimal number
DEC/DECIMAL(p,s)	Fixed-point decimal number
MONEY	Equivalent to DECIMAL(16,2)
MONEY(p)	Equivalent to DECIMAL(p,2)
MONEY(p,s)	Equivalent to DECIMAL(p,s)
REAL/SMALLFLOAT	approx floating point (C float)
DOUBLE PREC./FLOAT	approx floating point (C double)

IBM DB2 numeric data types are compatible with INFORMIX numeric data types, except for the following:

INFORMIX Data Type	IBM DB2 equivalent
DECIMAL(p)	Floating point decimals are not supported in DB2!
DECIMAL(32[,s])	DB2 decimals maximum precision is 31 digits!
MONEY	DECIMAL(16,2)
MONEY(p)	DECIMAL(p,2)
MONEY(p,s)	DECIMAL(p,s)

Solution:

SQL scripts to create databases must be converted manually. Tables created from BDL programs do not have to be converted; the database interface detects the MONEY data type and uses the DECIMAL type for DB2.

Warning: Floating point decimals (DECIMAL(P)) are not supported with DB2.

Warning: The maximum precision for DB2 decimals is 31 digits, while Informix supports 32 digits.

ODIDB2022 - Getting one row with SELECT

With INFORMIX, you must use the system table with a condition on the table id :

```
SELECT user FROM systables WHERE tabid=1
```

With IBM DB2, you have to do the following :

```
SELECT user FROM SYSIBM.SYSTABLES WHERE NAME='SYSTABLE'
```

Solution:

Check the BDL sources for "`FROM systables WHERE tabid=1`" and use dynamic SQL to resolve this problem.

ODIDB2024 - MATCHES and LIKE in SQL conditions

INFORMIX supports MATCHES and LIKE in SQL statements, while IBM DB2 supports the LIKE statement only.

MATCHES allows you to use brackets to specify a set of matching characters at a given position :

```
( col MATCHES '[Pp]aris' ).
( col MATCHES '[0-9][a-z]*' ).
```

In this case, the LIKE statement has no equivalent feature.

The following substitutions must be made to convert a MATCHES condition to a LIKE condition :

- MATCHES keyword must be replaced by LIKE.
- All '`*`' characters must be replaced by '`%`'.
- All '`?`' characters must be replaced by '_'.
- Remove all brackets expressions.

Solution:

Warning: SQL statements using MATCHES expressions must be reviewed in order to use LIKE expressions.

See also: MATCHES operator in SQL Programming.

ODIDB2025 - INFORMIX specific SQL statements in BDL

The BDL compiler supports several INFORMIX specific SQL statements that have no meaning when using IBM DB2:

- CREATE DATABASE
- DROP DATABASE
- START DATABASE (SE only)
- ROLLFORWARD DATABASE
- SET [BUFFERED] LOG
- CREATE TABLE with special options (storage, lock mode, etc.)

Solution:

Review your BDL source and remove all static SQL statements that are INFORMIX specific.

ODIDB2028 - INSERT cursors

INFORMIX supports insert cursors. An "insert cursor" is a special BDL cursor declared with an INSERT statement instead of a SELECT statement. When this kind of cursor is open, you can use the PUT instruction to add rows and the FLUSH instruction to insert the records into the database.

For INFORMIX databases with transactions, OPEN, PUT and FLUSH instructions must be executed within a transaction.

IBM DB2 does not support insert cursors.

Solution:

Insert cursors are emulated by the IBM DB2 database interface.

ODIDB2029 - SQL functions and constants

Both INFORMIX and DB2 provide numerous built-in SQL functions. Most INFORMIX SQL functions have the same name and purpose in DB2 (DAY(), MONTH(), YEAR(), UPPER(), LOWER(), LENGTH()).

INFORMIX	IBM DB2
today	current date
current hour to second	current time
current year to fraction(5)	current timestamp
trim([leading | trailing | both "char" FROM] "string")	ltrim( ) and rtrim( )
pow(x,y)	power(x,y)

Solution:

Warning: You must review the SQL statements using TODAY / CURRENT / EXTEND expressions.

You can create user defined functions (UFs) in the DB2 database.

ODIDB2030 - Very large data types

Both INFORMIX and IBM DB2 provide special data types to store very large texts or images.

IBM DB2 recommends the following conversion rules :

INFORMIX Data Type	IBM DB2 Data Type
TEXT	LONG VARCHAR or CLOB
BYTE	BLOB, VARGRAPHIC or DBCLOB

Solution:

The DB2 database interface can convert BDL TEXT data to CLOB and BYTE data to BLOB.

Warning: DB2 CLOB and BLOB columns are created with a size of 500K.

ODIDB2031 - Cursors WITH HOLD

INFORMIX provides the WITH HOLD option to prevent cursors being closed when a transaction ends.

Warning: This feature is well supported when using the DB2 interface, except when a transaction is canceled with a ROLLBACK, because DB2 automatically closes all cursors when you rollback a transaction.

Solution:

Check that your source code does not use WITH HOLD cursors after transactions canceled with ROLLBACK.

ODIDB2033 - Querying system catalog tables

As in INFORMIX, IBM DB2 provides system catalog tables (systables,syscolumns,etc.) in each database, but the table names and their structures are quite different.

Solution:

Warning: No automatic conversion of INFORMIX system tables is provided by the database interface.

ODIDB2036 - INTERVAL data type

INFORMIX INTERVAL data type stores a value that represents a span of time. INTERVAL types are divided into two classes : *year-month intervals* and *day-time intervals*.

DB2 does not provide a data type corresponding the INFORMIX INTERVAL data type.

Solution:

Warning: The INTERVAL data type is not well supported because the database server has no equivalent native data type. However, BDL INTERVAL values can be stored into and retrieved from CHAR columns.

ODIDB2039 - Data storage concepts

An attempt should be made to preserve as much of the storage information as possible when converting from INFORMIX to IBM DB2. Most important storage decisions made for INFORMIX database objects (like initial sizes and physical placement) can be reused for the IBM DB2 database.

Storage concepts are quite similar in INFORMIX and in IBM DB2, but the names are different.

The following table compares INFORMIX storage concepts to IBM DB2 storage concepts :

INFORMIX	IBM DB2
Physical units of storage	
The largest unit of physical disk space is a "**chunk**", which can be allocated either as a cooked file (I/O is controlled by the OS) or as raw device (=UNIX partition, I/O is controlled by the database engine). A "dbspace" uses at least one "chunk" for storage. You must add "chunks" to "dbspaces" in order to increase the size of the logical unit of storage.	One or more "**containers**" are created for each "tablespace" to physically store the data of all logical structures. Like INFORMIX "chunks", "containers" can be an OS file or a raw device. You can add "containers" to a "tablespace" in order to increase the size of the logical unit of storage or you can define EXTEND options.
A "**page**" is the smallest physical unit of disk storage that the engine uses to read from and write to databases. A "chunk" contains a certain number of "pages". The size of a "page" must be equal to the operating system's block size.	At the finest level of granularity, IBM DB2 stores data in "**data blocks**" with size corresponding to a multiple of the operating system's block size. You set the "data block" size when creating the database.
An "**extent**" consists of a collection of contiguous "pages" that the engine uses to allocate both initial and subsequent storage space for database tables. When creating a table, you can specify the first extent size and the size of future extents with the EXTENT SIZE and NEXT EXTENT options. For a single table, "extents" can be located in different "chunks" of the same "dbspace".	An "**extent**" is a specific number of contiguous "data blocks", obtained in a single allocation. When creating a table, you can specify the first extent size and the size of future extents with the STORAGE() option. For a single table, "extents" can be located in different "data files" of the same "tablespace".
Logical units of storage	
A "**table**" is a logical unit of storage that	Same concept as INFORMIX.

contains rows of data values.	
A "**database**" is a logical unit of storage that contains table and index data. Each database also contains a system catalog that tracks information about database elements like tables, indexes, stored procedures, integrity constraints and user privileges.	Same concept as INFORMIX. An IBM DB2 instance can manage several databases.
Database tables are created in a specific "**dbspace**", which defines a logical place to store data. If no dbspace is given when creating the table, INFORMIX defaults to the current database dbspace.	Database tables are created in a specific "**tablespace**", which defines a logical place to store data. The main difference with Informix "dbspaces", is that IBM DB2 tablespaces belong to a "database", while Informix "dbspaces" are external to a database.
Other concepts	
When initializing an INFORMIX engine, a "**root dbspace**" is created to store information about all databases, including storage information (chunks used, other dbspaces, etc.).	Each IBM DB2 database uses a set of "**control files**" to store internal information. These files are located in a dedicated directory : ".../$DB2INSTANCE/NODEnnnn"
The "**physical log**" is a set of continuous disk pages where the engine stores "before-images" of data that has been modified during processing. The "**logical log**" is a set of "**logical-log files**" used to record logical operations during on-line processing. All transaction information is stored in the logical log files if a database has been created with transaction log. INFORMIX combines "physical log" and "logical log" information when doing fast recovery. Saved "logical logs" can be used to restore a database from tape.	DB2 uses "**database log files**" to record SQL transactions.

ODIDB2040 - National characters data types

INFORMIX : NCHAR & NVARCHAR
IBM DB2 : ?

Solution:

1374

Warning: National character data types are not supported yet.

ODIDB2043 - SQL parameters limitation

The IBM DB2 SQL parser does not allow some uses of the '?' SQL parameter marker.

The following SQL expressions are not supported :

```
? IS [NOT] NULL
? <operator> ?
<function>( ? )
```

SQL instructions containing these expressions raise an error during the statement preparation.

Solution:

Check that your BDL programs do not use these kinds of conditional expressions.

If you really need to test a BDL variable during the execution of a SQL statement, you must use the CAST() function for DB2 only :

```
WHERE CAST( ? AS INTEGER ) IS NULL
```

See the DB2 documentation for more details.

ODIDB2046 - The LOAD and UNLOAD instructions

INFORMIX provides two SQL instructions to export / import data from / into a database table: The UNLOAD instruction copies rows from a database table into an text file, and the LOAD instruction inserts rows from an text file into a database table.

IBM DB2 does not provide LOAD and UNLOAD instructions.

Solution:

LOAD and UNLOAD instructions are supported.

Warning: There is a difference when using DB2 TIME and TIMESTAMP columns: TIME columns created in the IBM DB2 database are similar to INFORMIX DATETIME HOUR TO SECOND columns. In LOAD and UNLOAD, all DB2 TIME columns are treated as INFORMIX DATETIME HOUR TO SECOND columns and thus will be unloaded with the "hh:mm:ss" format.

Warning: When using an INFORMIX database, simple dates are unloaded with the DBDATE format (ex: "23/12/1998"). Therefore, unloading from an INFORMIX database for loading into an DB2 database is not supported.

ODIDB2051 - Setup database statistics

INFORMIX provides a special instruction to compute database statistics in order to improve query optimization plans :

```
UPDATE STATISTICS [options]
```

IBM DB2 provides the following equivalent:

```
RUNSTATS ON TABLE full-qualified-table-name [options]
```

Warning: RUNSTATS is not a SQL instruction, it is a DB2 command and therefore cannot be executed from a BDL program.

Solution:

You must execute the RUNSTATS command manually from a DB2 Command Center.

ODIDB2052 - The GROUP BY clause

INFORMIX allows you to use column numbers in the GROUP BY clause

```
SELECT ord_date, sum(ord_amount) FROM order GROUP BY 1
```

IBM DB2 does not support column numbers in the GROUP BY clause.

Solution:

Use column names instead:

```
SELECT ord_date, sum(ord_amount) FROM order GROUP BY
ord_date
```

ODIDB2053 - The ALTER TABLE instruction

INFORMIX and IBM DB2 use different implementations of the ALTER TABLE instruction.

For example:

INFORMIX allows you to use multiple ADD clauses separated by commas. DB2 does not expect braces and the comma separator :

INFORMIX:
```
ALTER TABLE customer ADD(col1 INTEGER), ADD(col2 CHAR(20))
```
IBM DB2:
```
ALTER TABLE customer ADD col1 INTEGER  ADD col2 CHAR(20)
```

Depending on the values currently stored, INFORMIX can change the data type of a column, while DB2 only supports changing the size of CHAR and VARCHAR columns :

INFORMIX:
```
ALTER TABLE customer MODIFY ( col1 INTEGER )
```
IBM DB2:
```
ALTER TABLE customer ALTER COLUMN col1 SET DATA TYPE VARCHAR(200)
```

Solution:

Warning: No automatic conversion is done by the database interface. Read the SQL documentation and review the SQL scripts or the BDL programs in order to use the database server specific syntax for ALTER TABLE.

ODIDB2054 - The star (asterisk) in SELECT statements

Informix allows you to use the star character in the select list along with other expressions :

```
SELECT col1, * FROM tab1 ...
```

IBM DB2 does not support this. You must use the table name as a prefix to the star :

```
SELECT col1, tab1.* FROM tab1 ...
```

Solution:

Always use the table name with stars.

ODIDB2055 - The LENGTH() function

INFORMIX provides the LENGTH() function:

```
SELECT LENGTH("aaa"), LENGTH(col1) FROM table
```

IBM DB2 has a equivalent function with the same name, but there is some difference:

Informix does not count the trailing blanks neither for CHAR not for VARCHAR expressions, while IBM DB2 counts the trailing blanks.

With the IBM DB2 LENGTH function, when using a CHAR column, values are always blank padded, so the function returns the size of the CHAR column. When using a VARCHAR column, trailing blanks are significant, and the function returns the number of characters, including trailing blanks.

Solution:

You must check if the trailing blanks are significant when using the LENGTH() function.

If you want to count the number of character by ignoring the trailing blanks, you must use the RTRIM() function:

```
SELECT LENGTH(RTRIM(col1)) FROM table
```

ODIDB2056 - SQL Interruption

With Informix, it is possible to interrupt a long running query if the SQL INTERRUPT ON option is set by the Genero program. The database server returns SQLCODE -213, which can be trapped to detect a user interruption.

```
MAIN
  DEFINE n INTEGER
  DEFER INTERRUPT
  OPTIONS SQL INTERRUPT ON
  DATABASE test1
  WHENEVER ERROR CONTINUE
  -- Start long query (self join takes time)
  -- From now on, user can hit CTRL-C in TUI mode to stop the query
  SELECT COUNT(*) INTO n FROM customers a, customers b
       WHERE a.cust_id <> b.cust_id
  IF SQLCA.SQLCODE == -213 THEN
     DISPLAY "Statement was interrupted by user..."
     EXIT PROGRAM 1
  END IF
  WHENEVER ERROR STOP
  ...
END MAIN
```

DB2 UDB 9 supports SQL Interruption in a similar way as Informix. The db client must issue an SQLCancel() ODBC call to interrupt a query.

Solution:

The DB2 database driver supports SQL interruption and converts the native SQL error code -952 to the Informix error code -213.

ODIDB2100 - Data type conversion table

INFORMIX Data Types	DB2 Data Types
CHAR(n)	CHAR(n) (limit = 254c!)
VARCHAR(n)	VARCHAR(n) (limit = 32672c!)
INTEGER	INTEGER
SMALLINT	SMALLINT
FLOAT[(n)]	FLOAT(n)
SMALLFLOAT	SMALLFLOAT
DECIMAL(p)	No floating point equivalent!
DECIMAL(p,s)	DECIMAL(p,s) (limit = 31 digits)
MONEY(p,s)	DECIMAL(p,s) (limit = 31 digits)
DATE	DATE
DATETIME HOUR TO SECOND	TIME
DATETIME q1 TO q2	TIMESTAMP
INTERVAL q1 TO q2	CHAR(n)

Connecting to DB2 OS/400

Note : some of the following actions can be taken via the OS/400 Operations Navigator.

DB2 Architecture on OS/400

On OS/400 machines, the DB2 Universal Database is integrated to the operating system. Therefore, some concepts change. For example, the physical organization of the database is quite different from UNIX or Windows platforms.

Common terms:

SQL Terms	DB2 OS/400 Terms
Table	Physical file

Row	Record
Column	Field
Index	Keyed logical file, access path
View	Non keyed logical file
Schema	Library, Collection, Schema (OS/400 V5R1 only)
Log	Journal
Isolation Level	Commitment control level

A Collection is a library containing a Journal, Journal Receivers, Views on the database catalogues.

Login to the AS/400 server

First, login to the AS/400 machine with a 5250 display emulation. All the commands are executed in the 5250 display emulation (or telnet connection).

Collection (Schema) Creation

A collection or library in DB2 for OS/400 is equivalent to a schema in DB2 for UNIX.

1. Launch "Interactive SQL"

```
STRSQL COMMIT(*NONE)
```

2. Create a Collection

```
CREATE COLLECTION
Press F4
Enter field values:
   LIBRARY : name of the collection (Schema)
   ASP : 1
   WITH DATA DICTIONARY : Y
Press ENTER
Press F3 to quit ( choose Option 1 (save and exit) ).
```

Note: The name of the Schema should not begin with “Q”; libraries beginning with “Q” are system libraries.

This procedure creates:

- A library for your new database,
- A catalog with a data dictionary,
- A journal (QSQJRN),
- A journal receiver (QSQJRN0001).

Source Physical File Creation

Each table in the database is stored in a Physical file. They can be created in the control center with SQL scripts (CREATE TABLE), or with OS/400 commands.

The table creation script file must be copied in the library in the form:
library/sourcefile.member

Creation of a physical file:

Type:
  CRTSRCPF
Enter field values:
  FILE = name of the table (10 characters max).
  LIBRARY = name of the library in which the table is created (schema).
  RECORD LENGTH = length of the script creation file (in bytes)
  MEMBER = *FILE

Execution of the SQL creation script:

Type
  RUNSQLSTM
Press F10 for additional parameters
Enter field values:
  SOURCE FILE = name of the source file of the script creation file
  LIBRARY = name of the library (schema)
  SOURCE MEMBER = name of the member of the script creation file
  NAMING FIELD = *SQL (SQL Naming convention library.table)
  COMMITMENT CONTROL = *NONE
  IBM SQL FLAGGING FIELD = *FLAG

If errors occur, you can use WRKSPLF to display error information saved in the spool file. Use option 5 in the Opt Field on the line of the script file you tried to execute.

Trigger Creation

With DB2 on OS/400, triggers need to be external programs written in a high level language such as C, COBOL, RPG, or PL/I.

To create a trigger, use the following steps:

1. Create an OS/400 Source file for the trigger programs

Create a source physical file on your AS/400 for the trigger programs. Each trigger program will be stored in a separate member within this source file.

Type:
  CRTSRCPF FILE(*library*/*file*)
where:

- *library* : name of the library you created for your new database
- *file* : name you want to call the trigger source physical file

The file name should be ten characters or fewer.

2. Create a member for each trigger program

Create a source file member for each trigger program. After the creation of trigger programs (in the next step), the programs will be forwarded to these members.

Type:
ADDPFM
Enter field values:
FILE = name of the source file you just created
LIBRARY = name of the library you created for your database
MEMBER = name you want to give the trigger source member

Repeat this operation for each trigger.

3. Create trigger programs in an OS/400 supported high level language

The OS/400-compatible languages include: ILE C/400, ILE COBOL, ILE RPG, COBOL, PL/I, and RPG.
The script creation file of the trigger should be send via FTP into *library/sourcefile.member,* where *sourcefile* and member are the values specified in the previous step.

4. Compile the trigger programs

Once the trigger programs are in AS/400 members, you can compile them. Use whichever compiler is appropriate for the language you used to create the trigger program.

5. Bind the trigger programs

After you compile the trigger programs, "bind" each compiled program file. Binding will establish a relationship between the program and any tables or views the program specifies.

Type:
CRTPGM PGM (*library*/*program*) ACTGRP(*CALLER)
where:
library is the name of the library you created for your new database
program is the name of the compiled trigger program

Repeat this operation for each trigger.

6. Add the trigger programs to physical files

The final step for migrating triggers is to add each program to a physical file. This will tie the trigger program to the table that calls it.

Type:
ADDPFTRG
Enter field values:
PHYSICAL FILE = name of the table you want to attach the trigger to
PHYSICAL FILE LIBRARY = name of the database library
TRIGGER TIME = either *BEFORE or *AFTER.
TRIGGER EVENT = *INSERT, *DELETE, or *UPDATE.
PROGRAM = name of the compiled program file
PROGRAM LIBRARY = name of the database library.
REPLACE TRIGGER = *YES.
ALLOW REPEATED CHANGES = *YES.

Note:: The trigger program should be in the same library as the database.

The trigger program is now tied to the table specified in the *Physical File* field and will be called each time the database action you specified above occurs. The trigger program may be called from interactive SQL, another AS/400 program, or an ODBC insert, delete, update, or procedure call.

Permission Definition

On OS/400, database security is managed at the operating system level, not at the database level. When you set up permissions for the database, you determine the degree of access (read, add, delete, etc.) individual users, groups, and authorization lists may have. This operation can easily be done via Operation Navigator.

The privileges must include the following system authorities:

- *USE to the Create Physical File (CRTPF) command.
- *EXECUTE and *ADD to the library into which the table is created.
- *OBJOPR and *OBJMGT to the journal.
- *CHANGE to the data dictionary if the library into which the table is created is an SQL collection with a data dictionary.

To define a foreign key, the privileges must include the following on the parent table:

- The REFERENCES privilege or object management authority for the table.
- The REFERENCES privilege on each column of the specified parent key.
- Ownership of the table.

The REFERENCES privilege on a table consists of:

- Being the owner of the table.
- Having the REFERENCES privilege to the table.
- Having the system authorities of either *OBJREF or *OBJMGT to the table.

The REFERENCES privilege on a column consists of:

- Being the owner of the table.
- Having the REFERENCES privilege to the column.
- Having the system authority of *OBJREF to the column or the system authority of *OBJMGT to the table.

To EXECUTE a user-defined function, the privilege consists of:

- Being owner of the user-defined function.
- Having EXECUTE privilege to the user-defined function.
- Having the system authorities of *OBJOPR and *EXECUTE to the user-defined function.

Relational DB Directory Entry Creation

The relational database directory is equivalent to the database directory of the DB2 client. This is necessary to access the database with DRDA clients (Distributed Relational Database Architecture) like DB2 client.

Use the WRKRDBDIRE tool to add the entry in the database directory:

- Type
  WRKDBDIRE
- Type Option 1 (add)
- Enter field values:
  ADDRESS = *LOCAL
  TYPE = *IP

Start the DDM server on the OS/400 which listens on the DRDA 446 port:

- Type STRTCPSVR *DDM

Start the database server:

- Type STRHOSTSVR
- Enter field values:
  SERVER TYPE = *DATABASE
  REQUIRED PROTOCOL : *ANY

The DDM/DRDA server that listens on TCP/IP port 446 handles requests from a DRDA client (examples are DB2 Connect or another AS/400).

The database server is not needed for DRDA clients, but it is needed for Client Access.

If a TCP/IP connection is desired, then your AS/400 server cannot have a release prior to V4R2 installed.

To manually configure the connection via the DB2 command line, you will need to enter catalog commands:

```
> db2 catalog tcpip node <node-name> remote <as400-adress> server
446
> db2 catalog db <db-name-alias> at node <node-name>
authentication dcs
> db2 catalog dcs db <db-name-alias> as <local-RDB-name-of-AS400>
```

If you catalogue the DB2 UDB for iSeries server incorrectly, you may get an SQL5048N error message. SQL7008N is another common error in that the DB2 UDB for iSeries tables being accessed on the server are not being journaled. To correct the SQL7008N error, you need to start journaling your tables or change the isolation level to No Commit.

The proper CCSID value (normally 37 for US English customers) is needed for any tables on the iSeries accessed via DB2 Connect. You can view the CCSID value with the DSPFD CL command or Operations Navigator. CCSID values can be changed with the ALTER TABLE statement or CHGPF CL command. Furthermore, to successfully connect, you may need to change one of the following: the CCSID of the job, the CCSID of the user profile used, or the system CCSID value (QCCSID) if it's the default 65535.

DB2 Client Configuration on Windows

To configure a DB2 client on Windows platforms, use the Client Configuration Assistant. This tool is available only under Microsoft Windows. Under Unix, you have to use the command line as described in the previous chapter.

1. Source:

    - Select “Manually configure a connection to a database”.

2. Protocol:

    - Select “TCP/IP”.

    - Check “The database physically resides on a host or AS/400 System”.

3. TCP/IP:

    - Host Name : AS/400 system name.

    - Port Number : Port where DDM/DRDA server is listening (default : 446).

4. Database:

    - Database name : name defined in the relational database directory entries (with WRKRDBDIRE).

5. ODBC:

- You can register the database as an ODBC data source. Not needed for DRDA connection used by ODI.

6. Node Options:

- Optional, but needed to access the database via the control center.

- System name : AS/400 system name.

- Instance name : not used for a connection to AS400 (because only one instance is running on an AS/400).

- Operating System : OS/400.

7. Security Options:

- Optional.

8. Host or AS400 Options:

- Optional.

Differences Between DB2 UNIX & DB2 OS/400

Some of the differences between DB2 for Unix/Windows and DB2 for OS/400 are:

- There is only one database on a system; you can not create two instances on the same database server. The database is a single system-wide database. The database name used for the connect statement is the name of the system. Schemas (Collections) can be used to manage different logical databases on the same OS/400 machine.
- There is no TABLESPACE concept on DB2 for iSeries. All the storage is controlled by the database manager and operating system.
- The identity column is not supported (for serial emulation).
- The SET SCHEMA SQL command is not supported.
- NUMERIC data type is defined as zoned decimal on DB2 for iSeries and packed decimal on other platforms.
- The FLOAT data type does not use the same storage. For portability across platforms, do not use FLOAT(n).
- Not all features of the CREATE FUNCTION statement are supported on each platform (see documentation).
- iSeries prior to V5R1 requires the statement to be processed by a special schema processor. iSeries as of V5R1 would require this only if the statement includes other DDL statements.
- OS/400 supports “SET DEFAULT” clause ON DELETE.
- OS/400 supports DROP statement with CASCADE behavior.
- Syntaxes of CREATE, ALTER and RENAME TABLE are different on the two systems.

Naming Conventions

The naming convention defines how database tables are identified.

DB2 OS/400 can use two kinds of naming conventions:

- The ***SQL** naming convention.
  The table has to be qualified with the name of the collection (schema) which must be the same name as the user connected to the database. All tables have to be in the same database.
- The ***SYS** naming convention.
  If a table is unqualified, it will be searched for in the *CURLIB collection. You can change the library list with the ADDLIBLE command. You may create a small CL program attached to the profile that will change the library list on sign on. You can also globally change the user portion of the library list using the QUSRLIBL system variable, but this would affect all users on the system.

ODI Adaptation Guide For Oracle 8.x, 9.x, 10.x, 11.x

Runtime configuration

- Install ORACLE and create a database
- Prepare the runtime environment

Database concepts

- Database concepts
- Data storage concepts
- Data consistency and concurrency management
- Transactions handling
- Defining database users
- Setting privileges

Data dictionary

- CHARACTER data types
- NUMERIC data types
- DATE and DATETIME data types
- INTERVAL data type
- SERIAL data type
- ROWIDs
- Very large data types
- National character data types
- The ALTER TABLE instruction
- Constraints
- Triggers
- Stored procedures
- Name resolution of SQL objects
- Setup database statistics
- NULLs in indexed columns
- Data type conversion table

Data manipulation

Reserved words
Outer joins
Transactions handling
Temporary tables
Substrings in SQL
The LENGTH( ) function
Empty character strings
Name resolution of SQL objects
String delimiters and object names
Getting one row with SELECT
MATCHES and LIKE conditions
SQL functions and constants
Querying system catalog tables
Syntax of UPDATE statements
The USER constant
The GROUP BY clause
The star in SELECT statements

BDL programming

SERIAL data type
Handling SQL errors when preparing statements
INFORMIX specific SQL statements in BDL
INSERT cursors
Cursors WITH HOLD
SELECT FOR UPDATE
UPDATE/DELETE WHERE CURRENT OF <cursor>
The LOAD and UNLOAD instructions
SQL Interruption

Runtime configuration

Install Oracle and create a database

1. Install the ORACLE Server on your computer.
2. Create and setup the Oracle **instance**.
3. Set up and start a **listener** if you plan to use a client / server architecture.

4. Create a database user dedicated to your application, the **application administrator** which will manage the database tables of the application:

```
$ sqlplus / AS SYSDBA
...
sqlplus> CREATE USER appadmin
IDENTIFIED BY password;
```

   You must grant privileges to this user:

```
sqlplus> GRANT CONNECT, RESOURCE TO
appadmin;
```

5. If you plan to use the default temporary table emulation, you must create the **TEMPTABS** tablespace. Note that this tablespace must be created as **permanent** tablespace. See issue ODIORA017 for more details:

```
sqlplus> CREATE TABLESPACE TEMPTABS
         DATAFILE 'file'
         SIZE 1M AUTOEXTEND ON NEXT
1K;
```

6. Connect as the application administrator:

```
sqlplus> CONNECT appadmin/password
```

7. Create the **application tables**. Do not forget to convert Informix data types to Oracle data types. See issue ODIORA100 for more details. Check for reserved words in your table and column names: Oracle 8i provides the V$RESERVED_WORDS view to track Oracle reserved words.
8. If you plan to use SERIAL emulation, you must choose an emulation method. You are free to use a technique based on SEQUENCES or based on the SERIALREG registration table. If you want to use the registration table technique, you must create the **SERIALREG** table and create a INSERT TRIGGER for each table using a SERIAL. See issue ODIORA005 for more details.

Prepare the runtime environment

1. If you want to connect to a remote Oracle server from an application server, you must install the **ORACLE Client Software** on your application server and configure this part.
2. Verify if the ORACLE environment variables are correct (**ORACLE_HOME**,**ORACLE_SID**). If you

are using the TNS protocol, verify if the **ORACLE listener** is started on the server. For testing, you can make a connection test with the SQL*Plus tool:

```
$ sqlplus username/password@service
```

3. Verify the environment variable defining the search path for shared libraries. On UNIX platforms, the variable is specific to the operating system, it can be **LIBPATH** (AIX), **LD_LIBRARY_PATH** (SOLARIS) or **SHLIB_PATH** (HP). On Windows platforms, the **OCI.DLL** must exist in **%ORACLE_HOME%\bin** and the **PATH** environment variable must contain this directory.
4. Check the database locale settings (**NLS_LANG**, **NLS_DATE_FORMAT**, etc). The DB locale must match the locale used by the runtime system (**LANG**).
5. Set up the **fglprofile** entries for database connections.
6. Set up fglprofile for the SERIAL emulation method. The following entry defines the SERIAL emulation method. You can either use the SEQUENCE based trigger or the SERIALREG based trigger method:

   ```
   dbi.database.dbname.ifxemul.datatype.serial.emulation = "(native|regtable)"
   ```

   The value 'native' selects the SEQUENCE based method and the value 'regtable' selects the SERIALREG based method. This entry has no effect if **dbi.database.<dbname>.ifxemul.datatype.serial** is set to 'false'.
   The default is SERIAL emulation enabled with native method (SEQUENCE-based). See issue ODIORA005 for more details.
7. Define the database schema selection if needed. Warning : This is only supported in Oracle 8i (8.1.5) and higher. The following entry defines the database schema to be used by the application. The database interface automatically executes an "ALTER SESSION SET CURRENT_SCHEMA <owner>" instruction to switch to a specific schema:

   ```
   dbi.database.dbname.ora.schema = "name>"
   ```

   Here *dbname* identifies the database name used in the BDL program (DATABASE *dbname*) and *name* is the schema name to be used in the ALTER SESSION instruction. If this entry is not defined, no

"ALTER SESSION" instruction is executed and the current schema defaults to the user's name.

8. Define pre-fetch parameters. Oracle offers high performance by pre-fetching rows in memory. The pre-fetching parameters can be tuned with the following entries:

   ```
   dbi.database.dbname.ora.prefetch.rows
   = integer

   dbi.database.dbname.ora.prefetch.memor
   y = integer # in bytes
   ```

   Note: These values will be applied to all application cursors.

   The interface pre-fetches rows up to the prefetch.rows limit unless the prefetch.memory limit is reached, in which case the interface returns as many rows as will fit in a buffer of size prefetch.memory. By default, pre-fetching is on and defaults to 10 rows, the memory parameter is set to zero, which means that memory size is not included in computing the number of rows to prefetch.

9. If needed, define a specific command to generate session identifiers with this FGLPROFILE setting:

   ```
      dbi.database.dbname.ora.sid.command
   = "SELECT ..."
   ```

   This unique session identifier will be used to create table names for temporary table emulation.
   By default, the database driver will use "SELECT USERENV('SESSIONID') FROM DUAL".

10. The default temporary table emulation uses regular permanent tables. If this does not fit you needs, you can use GLOBAL TEMPORARY TABLES with this FGLPROFILE setting:

    ```
    dbi.database.dbname.ifxemul.temptables
    .emulation = "global"
    ```

11. In order to connect to ORACLE, you must have a database driver "**dbmora***" installed.

ODIORA001 - DATE and DATETIME data types

INFORMIX provides two data types to store date and time information:

- **DATE** = for year, month and day storage.
- **DATETIME** = for year to fraction(1-5) storage.

ORACLE provides only the following data types to store date and time data:

- **DATE** = for year, month, day, hour, min, second storage.
- **TIMESTAMP** (Oracle 9i) = for year, month, day, hour, min, second, fraction storage.

String representing date time information:

INFORMIX is able to convert quoted strings to DATE / DATETIME data if the string contains matching environment parameters (i.e. DBDATE, GL_DATETIME).

As in INFORMIX, ORACLE can convert quoted strings to DATE or TIMESTAMP data if the contents of the string matches the NLS date format parameters (NLS_DATE_FORMAT, NLS_TIMESTAMP_FORMAT). The TO_DATE( ) and TO_TIMESTAMP() SQL functions convert strings to dates or timestamps, according to a given format. The TO_CHAR( ) SQL function allows you to convert dates or timestamps to strings, according to a given format.

Date arithmetic:

- INFORMIX supports date arithmetic on DATE and DATETIME values. The result of an arithmetic expression involving dates/times is a number of days when only DATEs are used, and an INTERVAL value if a DATETIME is used in the expression. In ORACLE, the result of an arithmetic expression involving DATE values is a NUMBER of days; the decimal part is the fraction of the day (0.5 = 12H00, 2.00694444 = (2 + (10/1440)) = 2 days and 10 minutes). The result of an expression involving Oracle TIMESTAMP data is of type INTERVAL. See Oracle documentation for more details.
- INFORMIX automatically converts an integer to a date when the integer is used to set a value of a date column. ORACLE does not support this automatic conversion.
- Complex DATETIME expressions (involving INTERVAL values for example) are INFORMIX specific and have no equivalent in ORACLE.
- To compare dates that have time data in ORACLE, you can use the ROUND() or TRUNC() SQL functions.

Solution:

The ORACLE DATE type is used for INFORMIX DATE data. The database interface automatically sets the time part to midnight (00:00:00) during input/output operations. You must be very careful since manual modifications of the database might set the time part, for example :

UPDATE table SET date_col = SYSDATE
(SYSDATE is equivalent to CURRENT YEAR TO SECOND in INFORMIX).
After this kind of update, when columns have date values with a time part different from midnight, some SELECT statements might not return all the expected rows.

INFORMIX DATETIME data with any precision from YEAR to SECOND is stored in ORACLE DATE columns. The database interface makes the conversion automatically. Missing date or time parts default to 1900-01-01 00:00:00. For example, when using a DATETIME HOUR TO MINUTE with the value of "11:45", the ORACLE DATE value will be "1900-01-01 11:45:00".

When using ORACLE 9i, INFORMIX DATETIME YEAR TO FRACTION(n) data is stored in ORACLE TIMESTAMP columns. The TIMESTAMP data type can store up to 9 digits in the fractional part, and therefore can store all precisions of INFORMIX DATETIME.

Warning: Using integers (number of days since 1899/12/31) as dates is not supported by ORACLE. Check your code to detect where you are using integers with DATE columns.

Warning: Literal DATETIME and INTERVAL expressions (i.e. DATETIME (1999-10-12) YEAR TO DAY) are not converted.

Warning: It is strongly recommended that you use BDL variables in dynamic SQL statements instead of quoted strings representing DATEs. For example :

```
LET stmt = "SELECT ... FROM customer WHERE creat_date >'", adate,"'"
```

is not portable. Use a question mark place holder instead and OPEN the cursor by USING adate :

```
LET stmt = "SELECT ... FROM customer WHERE creat_date > ?"
```

Warning: Most arithmetic expressions involving dates (for example, to add or remove a number of days from a date) will produce the same result with ORACLE. But keep in mind that ORACLE evaluates date arithmetic expressions to NUMBERs (<days>.<fraction>) while INFORMIX evaluates to INTEGERs when only DATEs are used in the expression, or to INTERVALs if at least one DATETIME is used in the expression.

Warning: DATE arithmetic expressions using SQL parameters (USING variables) are not fully supported. For example: `" SELECT ... WHERE datecol < ? + 1"` generates an error at PREPARE time.

Warning: SQL Statements using expressions with TODAY / CURRENT / EXTEND must be reviewed and adapted to the native syntax.

ODIORA003 - Reserved words

SQL object names like table and column names cannot be SQL reserved words in ORACLE.

An example of a common word which is part of the ORACLE SQL grammar is '**level**'.

Solution:

Table or column names which are ORACLE reserved words must be renamed.

ORACLE reserved keywords are listed in the ORACLE documentation, or Oracle 8i provides the V$RESERVED_WORDS view to track Oracle reserved words. All BDL application sources must be verified. To check if a given keyword is used in a source, you can use UNIX 'grep' or 'awk' tools. Most modifications can be done automatically with UNIX tools like 'sed' or 'awk'.

ODIORA004 - ROWIDs

ORACLE provides ROWIDs but the data type is different: INFORMIX rowids are INTEGERs while ORACLE rowids are CHAR(18).

ORACLE ROWIDs are physical addresses of the rows. For example :
`AAAA8mAALAAAAQkAAA`

Since ORACLE rowids are physical addresses, they cannot be used as permanent row identifiers (After a DELETE, an INSERT statement might reuse the physical place of the deleted row, to store the new row).

With INFORMIX, SQLCA.SQLERRD[6] contains the ROWID of the last INSERTed or UPDATEd row. This is not supported with ORACLE because ORACLE ROWID are not INTEGERs.

Solution:

If the BDL application uses INFORMIX rowids as primary keys, the program logic should be reviewed in order to use the real primary keys (usually, serials which can be supported) or ORACLE rowids as CHAR(18) (INFORMIX rowids will fit in this char data type).

If you cannot avoid the use of rowids, you must change the type of the variables which hold ROWID values. Instead of using INTEGER, you must use CHAR(18). INFORMIX rowids (INTEGERs) will automatically fit into a CHAR(18) variable.

All references to SQLCA.SQLERRD[6] must be removed because this variable will not contain the ROWID of the last INSERTed or UPDATEd row when using the ORACLE interface.

ODIORA005 - SERIAL data type

INFORMIX SERIAL data type and automatic number production :

- The table column must be of type SERIAL.
- To generate a new serial, **no value** or **zero** value is given to the INSERT statement :
```
INSERT INTO tab1 ( c ) VALUES ( 'aa' )
INSERT INTO tab1 ( k, c ) VALUES ( 0, 'aa' )
```
- After INSERT, the new SERIAL value is provided in **`SQLCA.SQLERRD[2]`**.

ORACLE sequences :

- Sequences are totally detached from tables.
- The purpose of sequences is to provide unique integer numbers.
- Sequences are identified by a sequence name.
- To create a sequence, you must use the CREATE SEQUENCE statement. Once a sequence is created, it is permanent (like a table).
- To get a new sequence value, you must use the **nextval** keyword, preceded by the name of the sequence.
  The **<seqname>.nextval** expression can be used in INSERT statements :
```
INSERT INTO tab1 VALUES ( tab1_seq.nextval, ... )
```
- To get the last generated number, ORACLE provides the **currval** keyword :
```
SELECT <seqname>.currval FROM DUAL
```

Remark: In order to improve performance, ORACLE can handle a set of sequences in the cache (See CREATE SEQUENCE syntax in the ORACLE documentation).

INFORMIX allows you to insert rows with a value different from zero for a serial column. Using an explicit value will automatically increment the internal serial counter, to avoid conflicts with future INSERTs that are using a zero value:

```
CREATE TABLE tab ( k SERIAL );  --> internal counter = 0
INSERT INTO tab VALUES ( 0 );   --> internal counter = 1
INSERT INTO tab VALUES ( 10 );  --> internal counter = 10
INSERT INTO tab VALUES ( 0 );   --> internal counter = 11
DELETE FROM tab;                --> internal counter = 11
INSERT INTO tab VALUES ( 0 );   --> internal counter = 12
```

Solution:

When using Oracle, the SERIAL data type can be emulated with INSERT TRIGGERs. In BDL programs, the SQLCA structure is filled as expected (after an insert, **sqlca.sqlerrd[2]** holds the last generated serial value).

The triggers can be created manually during the database creation procedure, or **automatically** from a BDL program: When a BDL program executes a CREATE [TEMP] TABLE with a SERIAL column, the Oracle interface automatically converts the SERIAL data type to NUMBER(10,0) and dynamically creates the trigger. For temporary tables, the trigger is dropped automatically after a "DROP TABLE temptab" or when the program disconnects from the database.

Warning: Users executing programs which create tables with SERIAL columns must have the **CONNECT** and **RESOURCE** roles assigned to create triggers and sequences.

In database creation scripts, all SERIAL[(n)] data types must be converted to NUMBER(10,0) data types and you must create the triggers (and the sequences when using sequence-based triggers). See below for more details.

Warning : With Oracle, INSERT statements using NULL for the SERIAL column will produce a new serial value, not a NULL as INFORMIX does:

```
INSERT INTO tab (col1,col2) VALUES (NULL,'data')
```

This behavior is mandatory in order to support INSERT statements which do not use the serial column :

```
INSERT INTO tab (col2) VALUES ('data')
```

Check whether your application uses tables with a SERIAL column that can contain a NULL value.

Warning: Since the INFORMIX SERIAL data type simulation is implemented in the ORACLE database, inserting rows with ORACLE tools like SQL*Plus or SQL*Loader will raise the INSERT triggers. When loading big tables, you can disable triggers with ALTER TRIGGER [ENABLE | DISABLE] (see ORACLE documentation for more details). After re-activation of the serial triggers, the SERIAL sequences must be re-initialized (use serialpkg.create_sequence('tab','col') or re-execute the PL/SQL script containing the sequence and trigger creation.

You are free to use **SEQUENCE based insert triggers** (1) or **SERIALREG based insert triggers** (2). The second solution needs the **SERIALREG** table to register serials.

With the following fglprofile entry, you define the technique to be used for SERIAL emulation:

```
dbi.database.<dbname>.ifxemul.datatype.serial.emulation =
{"native"|"regtable"}
```

The '**native**' value defines the SEQUENCE-based technique and the '**regtable**' defines the SERIALREG-based technique.

This entry must be used with :

```
dbi.database.<dbname>.ifxemul.serial = {true|false}
```

If this entry is set to false, the emulation method specification entry is ignored.

When no entry is specified, the default is SERIAL emulation enabled with 'native' method (SEQUENCE-based).

1. Using SEQUENCES based triggers

Each table having a SERIAL column needs an INSERT TRIGGER and a SEQUENCE dedicated to SERIAL generation.

To know how to write those sequences and triggers, you can create a small Genero program that creates a table with a SERIAL column. Set the FGLSQLDEBUG environment variable and run the program. The debug output will show you the native SQL commands to create the sequence and the trigger.

2. Using SERIALREG based triggers

Each table having a SERIAL column needs an INSERT TRIGGER which uses the SERIALREG table dedicated to SERIAL registration.

First, you must prepare the database and create the SERIALREG table as follows:

```
CREATE TABLE SERIALREG (
      TABLENAME VARCHAR2(50) NOT NULL,
      LASTSERIAL NUMBER(10,0) NOT NULL,
      PRIMARY KEY ( TABLENAME )
)
```

Warning: This table must exist in the database before creating the serial triggers.

In database creation scripts, all SERIAL[(n)] data types must be converted to INTEGER data types and you must create one trigger for each table. To know how to write those triggers, you can create a small Genero program that creates a table with a SERIAL column. Set the FGLSQLDEBUG environment variable and run the program. The debug output will show you the native trigger creation command.

Warning : The serial production is based on the SERIALREG table which registers the last generated number for each table. If you delete rows of this table, sequences will restart at start values and you might get duplicated values.

ODIORA006 - Outer joins

The syntax of OUTER joins is very different in INFORMIX and ORACLE :

In INFORMIX SQL, outer tables are defined in the FROM clause with the **OUTER** keyword :

```
SELECT ... FROM a, OUTER(b) WHERE a.key = b.akey
```

```
SELECT ... FROM a, OUTER(b,OUTER(c)) WHERE a.key = b.akey
AND b.key1 = c.bkey1 AND b.key2 = c.bkey2
```

ORACLE expects the **(+)** operator in the join condition. You must set a (+) after columns of the tables which must have NULL values when no record matches the condition:

```
SELECT ... FROM a, b WHERE a.key = b.key (+)
SELECT ... FROM a, b, c WHERE a.key = b.akey (+)
   AND b.key1 = c.bkey1 (+)
   AND b.key2 = c.bkey2 (+)
```

When using additional conditions on outer tables, the (+) operator also has to be used. For example :

```
SELECT ... FROM a, OUTER(b) WHERE a.key = b.akey AND b.colx
> 10
```

Must be converted to :

```
SELECT ... FROM a, b WHERE a.key = b.akey (+)
   AND b.colx (+) > 10
```

The ORACLE outer joins restriction :

In a query that performs outer joins of more than two pairs of tables, a single table can only be the NULL generated table for one other table. The following case is not allowed : `WHERE a.col = b.col (+) AND b.col (+) = c.col`

***Solution*:**

The Oracle interface can convert most INFORMIX OUTER specifications to Oracle outer joins.

Prerequisites :

1. In the FROM clause, the main table must be the first item and the outer tables must figure from left to right in the order of outer levels.
   Example which does not work : "`FROM OUTER(tab2), tab1` ".
2. The outer join in the WHERE part must use the table name as prefix.
   Example : "`WHERE tab1.col1 = tab2.col2` ".

Restrictions :

1. Statements composed by 2 or more SELECT instructions are not supported.
   Example : "`SELECT ... UNION SELECT`" or "`SELECT ... WHERE col IN (SELECT...)`"

Notes::

1. Table aliases are detected in OUTER expressions.
   OUTER example with table alias : "`OUTER( tab1 alias1)`".
2. In the outer join, <outer table>.<col> can be placed on both right or left sides of the equal sign.
   OUTER join example with table on the left : "`WHERE outertab.col1 = maintab.col2` ".
3. Table names detection is not case-sensitive.
   Example : "`SELECT ... FROM tab1, TAB2 WHERE tab1.col1 = tab2.col2`".
4. Temporary tables are supported in OUTER specifications.

ODIORA007a - Database concepts

Most of BDL applications use only one database entity (in the meaning of INFORMIX). But the same BDL application can connect to different occurrences of the same database schema, allowing several users to connect to those different databases.

INFORMIX servers can handle multiple database entities, while ORACLE servers manage only one database. ORACLE can manage multiple schemas, but by default other users must give the owner name as prefix to the table name:

```
SELECT * FROM stores.customer
```

Solution:

In an ORACLE database, each user can manage his own database schema. You can dedicate a database user to administer each occurrence of the application database.

Starting with version 8.1.5, any user can select the current database schema with the following SQL command:

```
ALTER SESSION SET CURRENT_SCHEMA = "<schema>"
```

Using this instruction, any user can access the tables without giving the owner prefix as long as the table owner has granted the privileges to access the tables.

You can make the database interface select the current schema automatically with the following fglprofile entry :

```
dbi.database.<dbname>.schema = "<schname>"
```

When using multiple database schemas, it is recommended that you create them in separated tablespaces to enable independent backups and keep logical sets of tables together. The simplest way is to define a default tablespace when creating the schema owner :

```
CREATE USER <user> IDENTIFIED BY <pswd>
       DEFAULT TABLESPACE <tabspacename>
       TEMPORARY TABLESPACE <tmptabspace>
```

ODIORA008a - Data consistency and concurrency management

Data consistency involves readers that want to access data currently modified by writers and **concurrency data access** involves several writers accessing the same data for modification. **Locking granularity** defines the amount of data concerned when a lock is set (row, page, table, ...).

INFORMIX

INFORMIX uses a locking mechanism to handle data consistency and concurrency. When a process changes database information with UPDATE, INSERT or DELETE, an **exclusive lock** is set on the touched rows. The lock remains active until the end of the transaction. Statements performed outside a transaction are treated as a transaction containing a single operation and therefore release the locks immediately after execution. SELECT statements can set **shared locks** according to the **isolation level**. In case of locking conflicts (for example, when two processes want to acquire an exclusive lock on the same row for modification or when a writer is trying to modify data protected by a shared lock), the behavior of a process can be changed by setting the **lock wait mode**.

Control:

- Lock wait mode : SET LOCK MODE TO ...
- Isolation level : SET ISOLATION TO ...
- Locking granularity : CREATE TABLE ... LOCK MODE {PAGE|ROW}
- Explicit exclusive lock : SELECT ... FOR UPDATE

Defaults:

- The default isolation level is read committed.
- The default lock wait mode is "not wait".
- The default locking granularity is page.

ORACLE

When data is modified, **exclusive locks** are set and held until the end of the transaction. For data consistency, ORACLE uses a **multi-version consistency model**: a copy of the original row is kept for readers before performing writer modifications. Readers do not have to wait for writers as in INFORMIX. The simplest way to think of Oracle's implementation of read consistency is to imagine each user accessing a private copy of the database, hence the multi-version consistency model. The **lock wait mode** cannot be changed session wide as in INFORMIX; the waiting behavior can be controlled with a

SELECT FOR UPDATE NOWAIT only. Locks are set at the **row level** in ORACLE, and this cannot be changed.

Control :

- Lock wait mode (on SELECT only): SELECT ... FOR UPDATE NOWAIT
- Isolation level : SET TRANSACTION ISOLATION LEVEL TO ...
- Explicit exclusive lock : SELECT ... FOR UPDATE [NOWAIT]

Defaults :

- The default isolation level is Read Committed (readers cannot see uncommitted data, no shared lock is set when reading data).

The main difference between INFORMIX and ORACLE is that readers do not have to wait for writers in ORACLE.

Solution:

ORACLE does not provide a dirty read mode, the (session wide) lock wait mode cannot be changed and the locking precision is always at the row level. Based on this, it is recommended that you work with INFORMIX in the read committed isolation level (default), make processes wait for each other (lock mode wait), and use the default page-level locking granularity.

See INFORMIX and ORACLE documentation for more details about data consistency, concurrency and locking mechanisms.

ODIORA008b - SELECT FOR UPDATE

A lot of BDL programs use pessimistic locking in order to prevent several users editing the same rows at the same time.

```
DECLARE cc CURSOR FOR
   SELECT ... FOR UPDATE [OF col-list]
OPEN cc
FETCH cc <-- lock is acquired
CLOSE cc <-- lock is released
```

- The row must be fetched in order to set the lock.
- If the cursor is local to a transaction, the lock is released when the transaction ends.
  If the cursor is declared "WITH HOLD", the lock is released when the cursor is closed.

ORACLE allows individual and exclusive row locking with :

```
SELECT ... FOR UPDATE [OF col-list]
```

- A lock is acquired for each selected row when the cursor is opened, before the first fetch.
- Cursors using SELECT ... FOR UPDATE are automatically closed when the transaction ends;
  Warning : Locks are **not** released **when a cursor is closed**.

ORACLE's locking granularity is at the row level.

To control the behavior of the program when locking rows, INFORMIX provides a specific instruction to set the wait mode :

```
SET LOCK MODE TO { WAIT | NOT WAIT | WAIT seconds }
```

The default mode is NOT WAIT. This as an INFORMIX specific SQL statement.

In order to simulate the same behavior in ORACLE, your can use the NOWAIT keyword in the SELECT ... FOR UPDATE statement, as follows:

```
SELECT ... FOR UPDATE [OF col-list] NOWAIT
```

With this option, ORACLE immediately returns an SQL error if the row is locked by another user.

Solution:

Warning: The database interface is based on an emulation of an Informix engine using transaction logging. Therefore, opening a SELECT ... FOR UPDATE cursor declared outside a transaction will raise an SQL error -255 (not in transaction).

Warning : Cursors declared with SELECT ... FOR UPDATE using the "WITH HOLD" clause cannot be supported with ORACLE. See ODIORA031 and ODIORA032 for more details.

If your BDL application is using pessimistic locking with SELECT ... FOR UPDATE, you must review the program logic to OPEN cursor and CLOSE cursor statements inside transactions (BEGIN WORK + COMMIT WORK / ROLLBACK WORK).

ODIORA009a - Transactions handling

INFORMIX and ORACLE handle transactions differently. The differences in the transactional models can affect the program logic.

INFORMIX native mode (non ANSI):

- DDL statements can be executed (and canceled) in transactions.

- Transactions must be started with BEGIN WORK. Statements executed outside of a transaction are automatically committed.

ORACLE :

- Beginnings of transactions are implicit; two transactions are delimited by COMMIT or ROLLBACK.
- The current transaction is automatically committed when a DDL statement is executed.

Transactions in stored procedures: avoid using transactions in stored procedures to allow the client applications to handle transactions, in accordance with the transaction model.

Solution:

Regarding transaction control instructions, BDL applications do not have to be modified in order to work with ORACLE. The INFORMIX behavior is simulated with an auto-commit mode in the ORACLE interface. A switch to the explicit commit mode is done when a BEGIN WORK is performed by the BDL program.

Warning : When executing a DDL statement inside a transaction, ORACLE automatically commits the transaction. Therefore, you must extract the DDL statements from transaction blocks.

See also ODIORA008b

ODIORA010 - Handling SQL errors when preparing statements

The ORACLE interface is implemented with the ORACLE Call Interface (OCI). This library does not provide a way to send SQL statements to the database server during the BDL PREPARE instruction, as in the INFORMIX interface. The statement is sent to the server only when opening the cursors or when executing the statement.

Therefore, when preparing an SQL statement with the BDL PREPARE instruction, no SQL errors can be returned if the statement has syntax errors, or if a column or a table name does not exist in the database. However, an SQL error will occur after the OPEN or EXECUTE instructions.

Solution:

Make sure your BDL programs do not test the STATUS or SQLCA.SQLCODE variable just after PREPARE instructions.

Change the program logic in order to handle the SQL errors when opening the cursors (OPEN) or when executing SQL statements (EXECUTE).

ODIORA011a - CHARACTER data types

INFORMIX provides the CHAR and VARCHAR data types to store characters. CHAR columns can store up to **32767** chars, and VARCHARs are limited to **255**. Starting with IDS 2000, INFORMIX provides the LVARCHAR data type which is limited to 2K.

ORACLE provides the CHAR and VARCHAR2 data types. CHAR columns can have a length of **2000** and VARCHAR2 can have a length of **4000**. VARCHAR is a synonym to VARCHAR2, but you should not use VARCHAR because the behavior may change in future server versions. See the ORACLE documentation for more details.

Warning : When comparing VARCHAR2 values in ORACLE, the trailing blanks are significant; this is not the case when using INFORMIX VARCHARs. But comparison with columns of type CHAR is similar to INFORMIX. See blank-padded and non-padded comparison semantics in ORACLE documentation.

Data type	INFORMIX	ORACLE
CHAR	'aaa ' = 'aaa'	'aaa ' = 'aaa'
VARCHAR	'aaa ' = 'aaa'	'aaa ' <> 'aaa'

Warning: ORACLE treats empty strings like NULL values; INFORMIX doesn't. See issue ODIORA011c for more details.

Solution:

The database interface supports character string variables in SQL statements for input (BDL USING) and output (BDL INTO).

Warning: Based on the comparison semantics, we recommend that you use ORACLE CHARs for INFORMIX CHARs. Take care if you want to use ORACLE VARCHAR2, since the comparison of values having trailing blanks is different.

Warning: Check that your database schema does not use CHAR or VARCHAR types with a length exceeding the ORACLE limit.

ODIORA011b - The LENGTH( ) function

INFORMIX provides the LENGTH() function:

```
SELECT LENGTH("aaa"), LENGTH(col1) FROM table
```

Oracle has a equivalent function with the same name, but there is some difference:

Informix does not count the trailing blanks neither for CHAR not for VARCHAR expressions, while Oracle counts the trailing blanks.

With the Oracle LENGTH function, when using a CHAR column, values are always blank padded, so the function returns the size of the CHAR column. When using a VAR CHAR column, trailing blanks are significant, and the function returns the number of characters, including trailing blanks.

The INFORMIX LENGTH() function returns 0 when the given string is empty. That means, `LENGTH('')` is 0.

Since ORACLE handles empty strings (`''`) as NULL values, writing "`LENGTH('')`" is equivalent to "`LENGTH(NULL)`". In this case, the function returns NULL.

Solution:

The ORACLE database interface cannot simulate the behavior of the INFORMIX LENGTH() function.

You must check if the trailing blanks are significant when using the LENGTH() function.

If you want to count the number of character by ignoring the trailing blanks, you must use the RTRIM() function:

```
SELECT LENGTH(RTRIM(col1)) FROM table
```

SQL conditions which verify that the result of LENGTH( ) is greater that a given number do not have to be changed, because the expression evaluates to false if the given string is empty (`NULL>n`) :

```
SELECT * FROM x WHERE LENGTH(col)>0
```

Only SQL conditions that compare the result of LENGTH() to zero will not work if the column is NULL. You must check your BDL code for such conditions :

```
SELECT * FROM x WHERE LENGTH(col)=0
```

In this case, you must add a test to verify if the column is null:

```
SELECT * FROM x WHERE ( LENGTH(col)=0 OR col IS NULL )
```

In addition, when retrieving the result of a LENGTH( ) expression into a BDL variable, you must check that the variable is not NULL.

In ORACLE, you can use the NVL( ) function in order to get a non-null value :

```
SELECT * FROM x WHERE NVL(LENGTH(c),0)=0
```

INFORMIX Dynamic Server 7.30 supports the NVL() function, as in ORACLE. So you can write the same SQL for both INFORMIX 7.30 and ORACLE 8, as shown in the above example.

If the INFORMIX version supports stored procedures, you can create the following stored procedure in the INFORMIX database in order to use NVL( ) expressions :

```
create procedure nvl( val char(512), def char(512) )
       returning char(512);
   if val is null then
      return def;
   else
      return val;
   end if;
end procedure;
```

With this stored procedure, you can write NVL( ) expressions like **`NVL(LENGTH(c),0)`**. This should work in almost all cases and provides upward compatibility with INFORMIX Dynamic Server 7.30.

ODIORA011c - Empty character strings

INFORMIX SQL and ORACLE SQL handle empty quoted strings differently. ORACLE SQL does not distinguish between `''` and NULL, while INFORMIX SQL treats`''` (or `""`) as a string with a length of zero.

Warning: Using literal string values which are empty (`''`) for INSERT or UPDATE statements will result in the storage of NULLs with ORACLE, while INFORMIX would store the value as a string with a length of zero:

```
insert into tab1 ( col1, col2 ) values ( NULL, '' )
```

Warning: Using the comparison expression (`col=''`) with ORACLE has no meaning because an empty string is equivalent to NULL; (`col=NULL`) expressions will always evaluate to FALSE because this is not a correct expression: The expression should be (`col IS NULL`).

```
select * from tab1 where col2 IS NULL
```

In Informix 4GL, when setting a variable with an empty string constant, it is automatically set to a NULL value. When using one or more space characters, the value is set to one space character:

```
define x char(10)
let x = ""
if x is null then -- evaluates to TRUE
let x = "    "
if x = " " then   -- evaluates to TRUE
```

Solution:

The ORACLE database interface cannot automatically convert comparison expressions like (`col=""`) to (`col IS NULL`) because this would require an SQL grammar parser. The interface could convert expressions like (`col=""`), but it would do this for the whole SQL statement:

```
UPDATE tab1 SET col1 = "" WHERE col2 = ""
```

Would be converted to an incorrect SQL statement:

```
UPDATE tab1 SET col1 IS NULL WHERE col2 IS NULL
```

To increase portability, you should avoid the usage of literal string values with a length of zero in SQL statements; this would resolve storage and Boolean expressions evaluation differences between INFORMIX and ORACLE.

NULL or program variables can be used instead. Program variables set with empty strings (`let x=""`) are automatically converted to NULL by BDL and therefore are stored as NULL when using both INFORMIX or ORACLE databases.

ODIORA012 - Constraints

Constraint naming syntax:

Both INFORMIX and ORACLE support primary key, unique, foreign key, default and check constraints, but the constraint naming syntax is different : ORACLE expects the "CONSTRAINT" keyword **before** the constraint specification and INFORMIX expects it **after**.

UNIQUE constraint example:

INFORMIX	ORACLE

`CREATE TABLE scott.emp (`
`...`
`empcode CHAR(10) UNIQUE`
`  [CONSTRAINT pk_emp],`
`...`
	`CREATE TABLE scott.emp (`
	`...`
	`empcode CHAR(10)`
	`  [CONSTRAINT pk_emp]`
	`UNIQUE,`
	`...`

Primary keys:

Like INFORMIX, ORACLE creates an index to enforce PRIMARY KEY constraints (some RDBMS do not create indexes for constraints). Using "CREATE UNIQUE INDEX" to define unique constraints is obsolete (use primary keys or a secondary key instead).

Unique constraints:

Like INFORMIX, ORACLE creates an index to enforce UNIQUE constraints (some RDBMS do not create indexes for constraints).

Warning: When using a unique constraint, INFORMIX allows only one row with a NULL value, while ORACLE allows several rows with NULL! Using CREATE UNIQUE INDEX is obsolete.

Foreign keys:

Both INFORMIX and ORACLE support the ON DELETE CASCADE option. To defer constraint checking, INFORMIX provides the SET CONSTRAINT command while ORACLE provides the ENABLE and DISABLE clauses.

Check constraints:

Warning: The check condition may be any valid expression that can be evaluated to TRUE or FALSE, including functions and literals. You must verify that the expression is not INFORMIX specific.

Null constraints:

INFORMIX and ORACLE support not null constraints, but INFORMIX does not allow you to give a name to "NOT NULL" constraints.

Solution:

Constraint naming syntax:

The database interface does not convert constraint naming expressions when creating tables from BDL programs. Review the database creation scripts to adapt the constraint naming clauses for ORACLE.

ODIORA013 - Triggers

INFORMIX and ORACLE provide triggers with similar features, but the trigger creation syntax and the programming languages are totally different.

INFORMIX triggers define the stored procedures to be called when a database event occurs (before | after insert | update | delete ...), while ORACLE triggers can hold a procedural block.

In ORACLE, triggers can be created with 'CREATE OR REPLACE' to keep privileges settings. With INFORMIX, you must drop and create again.

ORACLE V8 provides an 'INSTEAD OF' option to completely replace the INSERT, UPDATE or DELETE statement. This is provided to implement complex storage operations, for example on views that are usually read-only (you can attach triggers to views).

Warning: ORACLE allows you to create multiple triggers on the same table for the same trigger event, but it does not guarantee the execution order.

Solution:

INFORMIX triggers must be converted to ORACLE triggers "by hand".

ODIORA014 - Stored procedures

Both INFORMIX and ORACLE support stored procedures, but the programming languages are totally different : **SPL** for INFORMIX versus **PL/SQL** for ORACLE.

In Oracle, stored procedures and functions can be implemented in packages (similar to BDL modules). This is a powerful feature which enables structured procedural programming in the database. ORACLE itself implements system tools with packages (dbms_sql, dbms_output, dbms_lock). Procedures, functions and packages can be created with 'CREATE OR REPLACE' to keep privileges settings.
With INFORMIX, you must drop and create again.

Warning: ORACLE uses a different privilege context when using dynamic SQL in PL/SQL; roles are not effective. Users must have direct privileges settings in order to perform DDL or DML operations inside dynamic SQL.

Solution:

INFORMIX stored procedures must be converted to ORACLE "by hand".

Try to use ORACLE packages in order to group stored procedures into modules.

ODIORA016a - Defining database users

INFORMIX users are defined at the operating system level, they must be members of the 'informix' group, and the database administrator must grant CONNECT, RESOURCE or DBA privileges to those users.

ORACLE users must be createed in the database with a CREATE USER command. Oracle supports different sort of user authentications.
Following command defines a user authenticated by the database server (must give username and password to connect):

```
CREATE USER <username> IDENTIFIED BY <pswd>
```

Users defined at the operating system level can be declared as ORACLE users with the "IDENTIFIED EXTERNALLY" clause :

```
CREATE USER OPS$<username> IDENTIFIED EXTERNALLY
```

In this case, ORACLE trusts the operating system, and users can connect to the database without giving any user name and password.

Warning: By default, database users authenticated by the operating systems have a name with the "OPS$" prefix. The 'OPS$' prefix can be changed with the OS_AUTHENT_PREFIX server parameter. You can set this parameter to blank ("") in order to use the same user names in the system and in the ORACLE database. See ORACLE documentation ("Server Administrators Guide", "User authentication") for more details.

Warning: When creating a user with OS authentication, the user name in the database must be in uppercase letters, even if the OS user name is lowercase.

Warning: For Windows NT operating system authentication to work, the SQLNET.AUTHENTICATION_SERVICES parameter must be set as follows in %ORACLE_HOME%\NETWORK\ADMIN\SQLNET.ORA :

```
SQLNET.AUTHENTICATION_SERVICES = (NTS)
```

Solution:

Based on the application logic (is it a multi-user application ?), you must create one or several ORACLE users. As INFORMIX users are operating system users, we recommend that you use the OS authentication services offered by ORACLE.

ODIORA016b - Setting privileges

INFORMIX and ORACLE user privileges management are quite similar.

ORACLE provides roles to group privileges which then can be assigned to users. Starting with version 7.20, INFORMIX provides roles too. But users must execute the SET ROLE statement in order to enable a role. ORACLE users do not have to explicitly set a role, they are assigned to a default privilege domain (set of roles). More than one role can be enabled at a time with ORACLE.

INFORMIX users must have at least the CONNECT privilege to access the database:

```
GRANT CONNECT TO (PUBLIC|user)
```

ORACLE users must have at least the CREATE SESSION privilege to access the database. This privilege is part of the CONNECT role.

```
GRANT CONNECT TO (PUBLIC|user)
```

Warning: INFORMIX database privileges do NOT correspond exactly to ORACLE CONNECT, RESOURCE and DBA roles. However, roles can be created with equivalent privileges.

Solution:

Create a role which groups INFORMIX CONNECT privileges, and assign this role to the application users :

```
CREATE ROLE ifx_connect IDENTIFIED BY oracle;
GRANT CREATE SESSION, ALTER SESSION, CREATE ANY VIEW, ...
TO ifx_connect;
GRANT ifx_connect TO user1;
```

ODIORA017 - Temporary tables

INFORMIX temporary tables are created through the **`CREATE TEMP TABLE`** DDL instruction or through a **`SELECT ... INTO TEMP`** statement. Temporary tables are automatically dropped when the SQL session ends, but they can also be dropped with the DROP TABLE command. There is no name conflict when several users create temporary tables with the same name.

Remark: BDL reports create a temporary table when the rows are not sorted externally (by the source SQL statement).

INFORMIX allows you to create indexes on temporary tables. No name conflict occurs when several users create an index on a temporary table by using the same index identifier.

ORACLE does not support temporary tables as Informix does. ORACLE 8.1 provides GLOBAL TEMPORARY TABLEs which are shared among processes (only data is temporary and local to a SQL process). INFORMIX does not shared temp tables among SQL processes; each process can create its own temp table without table name conflicts.

Solution:

In accordance with some prerequisites, temporary tables creation in BDL programs can be supported by the database interface.

The temporary table emulation can use regular tables or GLOBAL TEMPORARY tables. The way the driver converts Informix temp table statements to Oracle regular tables or global temporary tables is driven by the following FGLPROFILE entry:

```
dbi.database.<dbname>.ifxemul.temptables.emulation = { "default"
| "global" }
```

By default, the database driver uses regular tables (*default* emulation). This default emulation provides maximum compatibility with Informix temporary tables, but requires real table creation which can be a significant overhead with Oracle. The *global* emulation uses native Oracle Global Temporary Tables, requiring only one initial table creation and thus making programs run faster. However, the *global* emulation mode has to be used carefully because of some limitations and constraints.

Warning: When creating a temporary table, you perform a Data Definition Language statement. Oracle automatically commits the current transaction when executing a DDL statement. Therefore, you must avoid temp table creation/destruction in transactions.

Using the *default* temporary table emulation

How does the *default* emulation work?

- INFORMIX CREATE TEMP TABLE and SELECT INTO TEMP statements are automatically converted to ORACLE "CREATE TABLE". The name of the temporary table is converted to a unique table name.
- Tables are created in the current schema.
- Temporary tables are created with the option TABLESPACE TEMPTABS so that data is stored in a dedicated tablespace named "**TEMPTABS**". Storing temporary table data in a separated tablespace allows you to use a physical device which can be different from the disk drive used for real data storage. Additionally, backups can be performed without the data of temporary tables. Of course this tablespace must exist otherwise temporary table creation will fail. This tablespace is not needed when using GLOBAL TEMPORARY tables.
- Once the temporary table has been created, all other SQL statements performed in the current SQL session are parsed to convert the original table name to the corresponding unique table name.
- When the BDL program disconnects from the database (for example, when it ends or when a CLOSE DATABASE instruction is executed), the tables which have not been removed with an explicit "DROP TABLE" are automatically removed by the database interface. However, if the program crashes, the tables will remain in the database, so you may need to cleanup the database from time to time.

Prerequisites when using the *default* emulation:

- Application users must have sufficient **privileges** to create database tables in their own schema (usually, "CONNECT" and "RESOURCE" roles).
- When using the default emulation based on permanent tables, you must create a dedicated tablespace named "**TEMPTABS**".
  Warning: The TEMPTABS tablespace must be of type "**permanent**", as it will hold permanent tables used to emulate Informix temp tables.
  Make sure it is big enough to hold all the data, and check for automatic extension.
  For more details, see "CREATE TABLESPACE" in the Oracle documentation.

Limitations of the *default* emulation:

- **Warning:** When using the default emulation, the real name of an emulated temporary table will get the following format:
  ```
  tt<number>_<original_name>
  ```
  Where <number> is the Oracle AUDSID session id returned by:
  ```
  SELECT USERENV('SESSIONID') FROM DUAL
  ```
  As Oracle 9i and 10g table names can't exceed 30 characters in length, and since session ids are persistent over server shutdown, you must pay attention to the names of your temporary tables. For example, if you create a temp table with the name TEMP_CUSTOMER_INVOICES (22c) it leaves 30 - (3 + 22) = 5 characters left for the session id, which gives a limit of 99999 sessions.
  To workaround this limitation, you can provide your own SQL command to generate a unique session id with the following FGLPROFILE entry:
  ```
  dbi.database.<dbname>.ora.sid.command = "select ..."
  ```
  As an example, you can use the SID column value from V$SESSION:
  ```
  SELECT SID FROM V$SESSION WHERE AUDSID =
  USERENV('SESSIONID')
  ```
- You are not allowed to use the unique table name format in your own database schema. Make sure you are not using table or column names with the following format:
  ```
  tt<number>_<original_name>
  ```
- Tokens matching the original table names are converted to unique names in all SQL statements. Make sure you are not using the temp table name for other database objects, like columns. The following example illustrates this limitation :
  ```
  CREATE TABLE tab1 ( key INTEGER, tmp1 CHAR(20) )
  CREATE TEMP TABLE tmp1 ( col1 INTEGER, col2 CHAR(20) )
  SELECT tmp1 FROM tab1 WHERE ...
  ```

Maintenance of *default* emulation:

- If you want to list the tables created by specific user, do the following:
  ```
  SELECT * FROM ALL_TABLES WHERE OWNER = '<user_name>'
  ```
  Remark: as with other database object names, the user name is stored in uppercase letters if it has been created without using double quotes (`create user scott` ... = stored name is "SCOTT").

Creating indexes on temporary tables with *default* emulation:

- Indexes created on temporary tables must have unique names too. The database interface detects CREATE INDEX statements which are using temporary tables and converts the index name to unique names.

- DROP INDEX statements are also detected to replace the original index name by the real name.

SERIALs in temporary table creation with *default* emulation:

- You can use the SERIAL data type when creating a temporary table.
  Sequences and triggers will be created in the current schema.
  See issue about SERIALs for more details.

Using the *global* temporary table emulation

Warning: The *global* temporary table emulation is provided to get benefit of the Oracle GLOBAL TEMPORARY TABLES, by sharing the same table structure with multiple SQL sessions, reducing the cost of the CREATE TABLE statement execution. However, this emulation does not provide the same level of Informix compatibility as the *default* emulation, and must be used carefully. See below for more details about the limitations and constraints.

How does the *global* emulation work?

- INFORMIX CREATE TEMP TABLE and SELECT INTO TEMP statements are automatically converted to ORACLE "CREATE GLOBAL TEMPORARY TABLE". The original table name is kept, but it gets a "**TEMPTABS**" schema prefix, to share the underlying table structure with other database users.
- The Global Temporary Tables are created with the "ON COMMIT PRESERVE ROWS" option, to keep the rows in the table when a transaction ends.
- The Global Temporary Tables are created in a specific schema called "**TEMPTABS**". If the table exists already, error ORA-00955 will just be ignored by the database driver. This allows to do several CREATE TEMP TABLE statements in your programs with no SQL error, to emulate the Informix behavior. This works fine as long as the table name is unique for a given structure (column count and data types must match).
- Once the Global Temporary Table has been created, all other SQL statements performed in the current SQL session are parsed to convert the original table name to TEMPTABS.*original-tablename*.
- When doing a DROP TABLE *temp-table* statement in the program, the database driver converts it to a DELETE statement, to remove all data added by the current session. A next CREATE TEMP TABLE or SELECT INTO TEMP will fail with error ORA-00955 but since this error is ignored, it will be transparent for the program. Note that we can't use TRUNCATE TABLE because that would required at least DROP ANY TABLE privileges for all users.
- When the BDL program disconnects from the database (for example, when it ends or when a CLOSE DATABASE instruction is executed), the tables which have not been dropped by the program with an explicit DROP TABLE statement will be automatically cleaned by Oracle.

Prerequisites when using the *global* emulation:

- You must create a database user (schema) dedicated to this emulation, with the name "**TEMPTABS**".
- All database users must have sufficient privileges to use Global Temporary Tables in the **TEMPTABS** schema: If you want programs to create Global Temporary Table on the fly, you must grant a CREATE ANY TABLE + CREATE ANY INDEX system privilege to all users. But this means that all users will be able to create/drop tables in any schema (Here Oracle (10g) is missing some fine-grained system privilege to create/drop tables in a particular schema). You better "prepare" the database by creating the Global Temporary Table with the TEMPTABS user (do not forget to specify ON COMMIT PRESERVE ROWS option), and give INSERT, UPDATE, DELETE and SELECT object privileges to

PUBLIC, for example:

```
  CREATE GLOBAL TEMPORARY TABLE temptabs.mytable
        ( k INT PRIMARY KEY, c CHAR(10) ) ON COMMIT
PRESERVE ROWS;
  CREATE UNIQUE INDEX temptabs.ix1 ON temptabs.mytable ( C
)
  GRANT SELECT, UPDATE, INSERT, DELETE ON temptabs.mytable
TO PUBLIC;
```

Limitations of the *global* emulation:

- **Warning:** Global Temporary Tables are shared by multiple users/sessions. In order to have the *global* emulation working properly with your application, each temporary table name must be unique for a given table structure, for all programs. You must for example avoid to use generic names such as **tmp1**. It is recommended to use table names as follows:

```
  CREATE TEMP TABLE custinfo_1 ( cust_id INTEGER, cust_name
VARCHAR(50) )
  CREATE TEMP TABLE custinfo_2 ( cust_id INTEGER, cust_name
VARCHAR(50), cust_addr VARCHAR(200) )
```

- Tokens matching the original table names are converted to unique names in all SQL statements. Make sure you are not using the temp table name for other database objects, like columns. The following example illustrates this limitation :

```
  CREATE TABLE tab1 ( key INTEGER, tmp1 CHAR(20) )
  CREATE TEMP TABLE tmp1 ( col1 INTEGER, col2 CHAR(20) )
  SELECT tmp1 FROM tab1 WHERE ...
```

Creating indexes on temporary tables with *global* emulation:

- Indexes created on temporary tables get also the **TEMPTABS** schema prefix.
- When executing a DROP INDEX statement on a temporary table in a program, the database driver just ignores the statement.

SERIALs in temporary table creation with *global* emulation :

- You can use the SERIAL data type when creating a temporary table. Sequences and triggers will be created in the **TEMPTABS** schema too. See issue about SERIALs for more details.

ODIORA018 - Substrings in SQL

INFORMIX SQL statements can use subscripts on columns defined with the character data type :

```
    SELECT ... FROM tab1 WHERE col1[2,3] = 'RO'
    SELECT ... FROM tab1 WHERE col1[10]  = 'R'   -- Same as
col1[10,10]
    UPDATE tab1 SET col1[2,3] = 'RO' WHERE ...
    SELECT ... FROM tab1 ORDER BY col1[1,3]
```

.. while ORACLE provides the SUBSTR( ) function, to extract a sub-string from a string expression :

```
SELECT .... FROM tab1 WHERE SUBSTR(col1,2,2) = 'RO'
SELECT SUBSTR('Some text',6,3) FROM DUAL        -- Gives 'tex'
```

Solution:

You must replace all Informix col[x,y] expressions by SUBSTR(col,x,y-x+1).

Warning: In UPDATE instructions, setting column values through subscripts will produce an error with ORACLE :

```
UPDATE tab1 SET col1[2,3] = 'RO' WHERE ...
```

is converted to:

```
UPDATE tab1 SET SUBSTR(col1,2,3-2+1) = 'RO' WHERE ...
```

ODIORA019 - Name resolution of SQL objects

INFORMIX uses the following form to identify an SQL object:

```
[database[@dbservername]:][{owner|"owner"}.]identifier
```

The ANSI convention is to use double quotes for identifier delimiters (For example : "tabname"."colname").

Warning: When using double-quoted identifiers, both INFORMIX and ORACLE become case sensitive. Unlike INFORMIX, ORACLE database object names are stored in UPPERCASE in system catalogs. That means that SELECT "col1" FROM "tab1" will produce an error because those objects are identified by "COL1" and "TAB1" in ORACLE system catalogs.

Remark: in INFORMIX ANSI compliant databases:

- The table name must include "owner", unless the connected user is the owner of the database object.
- The database server shifts the owner name to uppercase letters before the statement executes, unless the owner name is enclosed in double quotes.

With ORACLE, an object name takes the following form:

```
[(schema|"schema").](identifier|"identifier")[@database-link]
```

Remark: ORACLE has separate namespaces for different classes of objects (tables, views, triggers, indexes, clusters).

Object names are limited to 30 chars in ORACLE.

An ORACLE database schema is owned by a user (usually, the application administrator) and this user must create PUBLIC SYNONYMS to provide a global scope

for his table names. PUBLIC SYNONYMS can have the same name as the schema objects they point to.

Solution:

Check that you do not use singl-quoted or double-quoted table names or column names in your source. Those quotes must be removed because the database interface automatically converts double quotes to single quotes, and ORACLE does not allow single quotes as database object name delimiters.

See also issue ODIORA007a

ODIORA020 - String delimiters and object names

The ANSI string delimiter character is the single quote ('string'). Double quotes are used to delimit database object names ("object-name").

Example: `WHERE "tabname"."colname" = 'a string value'`

INFORMIX allows double quotes as string delimiters, but ORACLE doesn't. This is important, since many BDL programs use that character to delimit the strings in SQL commands.

Remark: this problem concerns only double quotes within SQL statements. Double quotes used in pure BDL string expressions are not subject to SQL compatibility problems.

Solution:

The ORACLE database interface can automatically replace all double quotes by single quotes.

Escaped string delimiters can be used inside strings like the following :

```
'This is a single quote : '''
'This is a single quote : \''
"This is a double quote : """
"This is a double quote : \""
```

Warning: Database object names cannot be delimited by double quotes because the database interface cannot determine the difference between a database object name and a quoted string !

For example, if the program executes the SQL statement:
`WHERE "tabname"."colname" = "a string value"`
replacing all double quotes by single quotes would produce :

```
WHERE 'tabname'.'colname' = 'a string value'
```

This would produce an error since `'tabname'.'colname'` is not allowed by ORACLE.

Although double quotes are replaced automatically in SQL statements, you should use only single quotes to enforce portability.

ODIORA021 - NUMERIC data types

INFORMIX supports several data types to store numbers:

INFORMIX data type	Description
SMALLINT	16 bit integer (-2^15 to 2^15)
INT/INTEGER	32 bit integer (-2^31 to 2^31)
DEC/DECIMAL(p)	Floating-point decimal number
DEC/DECIMAL(p,s)	Fixed-point decimal number
MONEY	Equivalent to DECIMAL(16,2)
MONEY(p)	Equivalent to DECIMAL(p,2)
MONEY(p,s)	Equivalent to DECIMAL(p,s)
REAL/SMALLFLOAT	approx floating point (C float)
DOUBLE PREC./FLOAT	approx floating point (C double)

ORACLE supports only one data type to store numbers:

ORACLE data type	Description
NUMBER(p,s) (1<=p<= 38, -84<=s<=127)	Fixed point decimal numbers.
NUMBER(p) (1<=p<= 38)	Integer numbers with a precision of p.
NUMBER	Floating point decimals with a precision of 38.
FLOAT(b) (1<=b<= 126)	Floating point numbers with a binary precision b.
FLOAT	Floating point numbers with a binary precision of 126.

ANSI types like SMALLINT, INTEGER, FLOAT are supported by ORACLE but will be respectively converted to ORACLE native data types NUMBER(38), NUMBER(38) and NUMBER.

Warning: When dividing INTEGERs or SMALLINTs, INFORMIX rounds the result (7 / 2 = 3), while ORACLE doesn't, because it does not have a native integer data type (7 / 2 = 3.5)

Solution:

We recommend that you use the following conversion rules:

INFORMIX data type	ORACLE data type
DECIMAL(p,s), MONEY(p,s)	NUMBER(p,s)
DECIMAL(p)	FLOAT(b)
SMALLINT	NUMBER(5,0)
INTEGER	NUMBER(10,0)
SMALLFLOAT	NUMBER
FLOAT	NUMBER

Avoid dividing integers in SQL statements. If you do divide an integer, use the TRUNC() function with ORACLE.

ODIORA022 - Getting one row with SELECT

With INFORMIX, you must use the system table with a condition on the table id :

```
SELECT user FROM systables WHERE tabid=1
```

Oracle provides the **DUAL** table to generate one row only.

```
SELECT user FROM DUAL
```

Solution:

Check the BDL sources for "FROM systables WHERE tabid=1" and use dynamic SQL to resolve this problem.

ODIORA024 - MATCHES and LIKE in SQL conditions

INFORMIX supports MATCHES and LIKE in SQL statements, while ORACLE supports the LIKE statement only.

MATCHES allows you to use brackets to specify a set of matching characters at a given position :

```
   ( col MATCHES '[Pp]aris' ).
   ( col MATCHES '[0-9][a-z]*' ).
```

In this case, the LIKE statement has no equivalent feature.

The following substitutions must be done to convert a MATCHES condition to a LIKE condition:

- MATCHES keyword must be replaced by LIKE.
- All '`*`' characters must be replaced by '`%`'.
- All '`?`' characters must be replaced by '_'.
- Remove all brackets expressions.

Solution:

Warning: SQL statements using MATCHES expressions must be reviewed in order to use LIKE expressions.

ORACLE provides the TRANSLATE function which can be used to replace MATCHES in specific cases. The TRANSLATE function replaces all occurrences of characters listed in a '*from*' set, with the corresponding character defined in a '*to*' set.

```
INFORMIX : WHERE col MATCHES '[0-9][0-9][0-9]'
ORACLE   : WHERE TRANSLATE(col,'0123456789','9999999999')='999'
```

See also: MATCHES operator in SQL Programming.

ODIORA025 - INFORMIX specific SQL statements in BDL

The BDL compiler supports several INFORMIX specific SQL statements that have no meaning when using ORACLE:

deleted the next sentence as not necessary

- CREATE DATABASE
- DROP DATABASE
- START DATABASE (SE only)
- ROLLFORWARD DATABASE
- SET [BUFFERED] LOG
- CREATE TABLE with special options (storage, lock mode, etc.)

Solution:

Review your BDL source and remove all static SQL statements which are INFORMIX specific.

ODIORA028 - INSERT cursors

INFORMIX supports insert cursors. An "insert cursor" is a special BDL cursor declared with an INSERT statement instead of a SELECT statement. When this kind of cursor is open, you can use the PUT instruction to add rows and the FLUSH instruction to insert the records into the database.

For INFORMIX database with transactions, OPEN, PUT and FLUSH instructions must be executed within a transaction.

ORACLE does not support insert cursors.

Solution:

Insert cursors are emulated by the ORACLE database interface.

ODIORA029 - SQL functions and constants

Almost all INFORMIX functions and SQL constants have a different name or behavior in ORACLE.

Here is a comparison list of functions and constants:

INFORMIX	ORACLE
today	trunc(sysdate)
current year to second	sysdate
day(value)	to_number(to_char(value, 'dd'))
extend(dtvalue, first to last)	to_date(nvl(to_char(dtvalue, 'fmt-mask'), '19000101000000'), 'fmt-mask')
mdy(m,d,y)	to_date(to_char(m,'09') || to_char(d,'09') || to_char(y,'0009'), 'MMDDYYYY')
month(date)	to_number(to_char(date, 'mm'))
weekday(date)	to_number(to_char(date, 'd')) -1
year(date)	to_number(to_char(date, 'yyyy'))
date("string" | integer)	No equivalent - Depends from DBDATE in IFX
user	user *! Uppercase/lowercase: See* ODIORA047
trim([leading | trailing | both "char" FROM] "string")	ltrim( ) and rtrim( )

length(c)	length(c) *! Different behavior: See* ODIORA011b
pow(x,y)	power(x,y)

Solution:

Warning: You must review the SQL statements using TODAY / CURRENT / EXTEND expressions.

You can define stored functions in the ORACLE database, to simulate INFORMIX functions. This works only for functions that are not already provided by ORACLE:

```
create or replace function month( adate in date )
return number
is
  v_month number;
begin
  v_month := to_number( to_char( adate, 'mm' ) );
  return (v_month);
end month;
```

You can find the scripts to create those stored functions in the **ifxf_ora.sql** script provided in the Adaptation Kit.

ODIORA030 - Very large data types

INFORMIX uses the TEXT and BYTE data types to store very large texts or images.ORACLE 8 provides CLOB, BLOB, and BFILE data types. Columns of these types store a kind of pointer (lob locator). This technique allows you to use more than one CLOB / BLOB / BFILE column per a table.

Solution:

The ORACLE database interface can convert BDL TEXT data to CLOB and BYTE data to BLOB.

Warning : ORACLE BFILEs are not supported.

ODIORA031 - Cursors WITH HOLD

INFORMIX closes opened cursors automatically when a transaction ends unless the WITH HOLD option is used in the DECLARE instruction. In ORACLE, opened cursors using SELECT statements without a FOR UPDATE clause are not closed when a

transaction ends. Actually, all ORACLE cursors are 'WITH HOLD' cursors unless the FOR UPDATE clause is used in the SELECT statement.

Solution:

BDL cursors that are not declared "WITH HOLD" are automatically closed by the database interface when a COMMIT WORK or ROLLBACK WORK is performed.

Warning: Since ORACLE automatically closes FOR UPDATE cursors when the transaction ends, opening cursors declared FOR UPDATE and WITH HOLD results in an SQL error that does not normally appear with INFORMIX, in the same conditions. Review the program logic in order to find another way to set locks.

ODIORA032 - UPDATE/DELETE WHERE CURRENT OF <cursor>

INFORMIX allows positioned UPDATEs and DELETEs with the "WHERE CURRENT OF <cursor>" clause, if the cursor has been DECLARED with a SELECT ... FOR UPDATE statement.

Warning: UPDATE/DELETE ... WHERE CURRENT OF <cursor> is not support by the Oracle database API. However, ROWIDs can be used for positioned updates/deletes.

Solution:

UPDATE/DELETE ... WHERE CURRENT OF instructions are managed by the ORACLE database interface. The ORACLE database interface replaces "WHERE CURRENT OF <cursor>" by "WHERE ROWID=:rid" and sets the value of the ROWID returned by the last FETCH done with the given cursor..

ODIORA033 - Querying system catalog tables

As in INFORMIX, ORACLE provides system catalog tables (actually, system views). But the table names and their structure are quite different.

Solution:

Warning: No automatic conversion of INFORMIX system tables is provided by the database interface.

ODIORA034 - Syntax of UPDATE statements

INFORMIX allows a specific syntax for UPDATE statements :

```
UPDATE table SET ( <col-list> ) = ( <val-list> )
```

or

```
UPDATE table SET table.* = myrecord.*
UPDATE table SET * = myrecord.*
```

Solution:

Static UPDATE statements using the above syntax are converted **by the compiler** to the standard form:

```
UPDATE table SET column=value [,...]
```

ODIORA036 - INTERVAL data type

INFORMIX's INTERVAL data type stores a value that represents a span of time. INTERVAL types are divided into two classes : ***year-month intervals*** and ***day-time intervals*****.**

ORACLE 8i does not provide a data type similar to Informix INTERVAL.

Starting from version 9i, ORACLE provides the INTERVAL data type similar to INFORMIX, with two classes (YEAR TO MONTH and DAY TO SECOND), but Oracle's INTERVAL cannot be defined with a precision different from these two classes (for example, you cannot define an INTERVAL HOUR TO MINUTE in Oracle). The class DAY TO SECOND(n) is equivalent to the INFORMIX INTERVAL class DAY TO FRACTION(n).

Solution:

When using Oracle 8i

It is not recommended that you use the INTERVAL data type because Oracle 8i has no equivalent native data type. This would cause problems when doing INTERVAL arithmetic on the database server side. However, INTERVAL values can be stored in CHAR columns.

When using Oracle 9i and higher

Informix INTERVAL YEAR(n) TO MONTH data is stored in Oracle INTERVAL YEAR(n) TO MONTH columns. These data types are equivalent.

Informix INTERVAL DAY(n) TO FRACTION(p) data is stored in Oracle INTERVAL DAY(n) TO SECOND(p) columns. These data types are equivalent.

Other Informix INTERVAL types must be stored in CHAR() columns as with Oracle 8i, because the high qualifier precision cannot be specified with Oracle INTERVALs. For example, Informix INTERVAL HOUR(5) TO MINUTE has no native equivalent in Oracle.

ODIORA039 - Data storage concepts

An attempt should be made to preserve as much of the storage specification as possible when converting from INFORMIX to ORACLE. Most important storage decisions made for INFORMIX database objects (like initial sizes and physical placement) can be reused for the ORACLE database.

Storage concepts are quite similar in INFORMIX and in ORACLE, but the names are different.

The following table compares INFORMIX storage concepts to ORACLE storage concepts :

INFORMIX	ORACLE
Physical units of storage	
The largest unit of physical disk space is a "**chunk**", which can be allocated either as a cooked file (I/O is controlled by the OS) or as raw device (=UNIX partition, I/O is controlled by the database engine). A "dbspace" uses at least one "chunk" for storage. You must add "chunks" to "dbspaces" in order to increase the size of the logical unit of storage.	One or more "**data files**" are created for each "tablespace" to physically store the data of all logical structures. Like INFORMIX "chunks", a "data file" can be an OS file or a raw device. You can add "data files" to a "tablespace" in order to increase the size of the logical unit of storage or you can use the AUTOEXTEND option when using OS files.
A "**page**" is the smallest physical unit of disk storage that the engine uses to read from and write to databases. A "chunk" contains a certain number of "pages". The size of a "page" must be equal to the operating system's block size.	At the finest level of granularity, ORACLE stores data in "**data blocks**" which size corresponds to a multiple of the operating system's block size. You set the "data block" size when creating the database.
An "**extent**" consists of a collection of contiguous "pages" that the engine uses to allocate both initial and subsequent storage space for database tables. When creating a table, you can specify	An "**extent**" is a specific number of contiguous "data blocks", obtained in a single allocation. When creating a table, you can specify the first extent size and the size of future extents with the STORAGE()

the first extent size and the size of future extents with the EXTENT SIZE and NEXT EXTENT options. For a single table, "extents" can be located in different "chunks" of the same "dbspace".	option. For a single table, "extents" can be located in different "data files" of the same "tablespace".
Logical units of storage	
A "**table**" is a logical unit of storage that contains rows of data values.	Same concept as INFORMIX.
A "**database**" is a logical unit of storage that contains table and index data. Each database also contains a system catalog that tracks information about database elements like tables, indexes, stored procedures, integrity constraints and user privileges.	Same concept as INFORMIX, but one ORACLE instance can manage only one database, in the meaning of INFORMIX.
Database tables are created in a specific "**dbspace**", which defines a logical place to store data. If no dbspace is given when creating the table, INFORMIX defaults to the current database dbspace.	Database tables are created in a specific "**tablespace**", which defines a logical place to store data. If no tablespace is given when creating the table, ORACLE defaults to the user's default tablespace.
The total disk space allocated for a table is the "**tblspace**", which includes "pages" allocated for data, indexes, blobs, tracking page usage within table extents. **Warning:** Do not confuse the INFORMIX "tblspace" concept and ORACLE "tablespaces".	A "**segment**" is a set of "extents" allocated for a certain logical structure. There are four different types of segments, including data segments, index segments, rollback segments and temporary segments.
Other concepts	
When initializing an INFORMIX engine, a "**root dbspace**" is created to store information about all databases, including storages information (chunks used, other dbspaces, etc.)	Each ORACLE database has a "**control file**" that records the physical structure of the database, like the database name, location and names of "data files" and "redo log" files, and time stamp of database creation.
The "**physical log**" is a set of continuous disk pages where the engine stores "before-images" of data that has been modified during processing.	A "**rollback segment**" records the actions of SQL transactions that could be rolled back, and it records the data as it existed before an operation in a transaction.
The "**logical log**" is a set of "**logical-log files**" used to record logical operations during on-line processing. All transaction	The "**redo log files**" hold all changes made to the database, in case the database experiences an instance

information is stored in the logical log files if a database has been created with transaction log. INFORMIX combines "physical log" and "logical log" information when doing fast recovery. Saved "logical logs" can be used to restore a database from tape.	failure. Each database has at least two "redo log files". Redo entries record data that can be used to reconstruct all changes made to the database, including the rollback segments stored in the database buffers of the SGA. Therefore, the online redo log also protects rollback data.

ODIORA040 - National characters data types

INFORMIX: NCHAR & NVARCHAR
ORACLE: NCHAR & NVARCHAR2

- Only OCI V8 supports NCHAR datatype.
- String constants must be preceded by the character 'N'.

Solution:

Warning: National character data types are not supported yet.

ODIORA046 - The LOAD and UNLOAD instructions

INFORMIX provides two SQL instructions to export / import data from / into a database table: The UNLOAD instruction copies rows from a database table into a text file and the LOAD instructions insert rows from a text file into a database table.

ORACLE does not provide LOAD and UNLOAD instructions, but provides external tools like SQL*Plus and SQL*Loader.

Solution:

In 4gl programs, the LOAD and UNLOAD instructions are supported with ORACLE, with some limitations:

Warning: There is a difference when using ORACLE DATE columns. DATE columns created in the ORACLE database are equivalent to INFORMIX DATETIME YEAR TO SECOND columns. In LOAD and UNLOAD, all ORACLE DATE columns are treated as INFORMIX DATETIME YEAR TO SECOND columns and thus will be unloaded with the "YYYY-MM-DD hh:mm:ss" format.
The same problem appears for INFORMIX INTEGER and SMALLINT values, which are

stored in an ORACLE database as NUMBER(?) columns. Those values will be unloaded as INFORMIX DECIMAL(10) and DECIMAL(5) values, that is, with a trailing dot-zero ".0".

Warning: When using an INFORMIX database, simple dates are unloaded using the DBDATE format (ex: "23/12/1998"). Therefore, unloading from an INFORMIX database for loading into an ORACLE database is not supported.

ODIORA047 - The USER constant

Both INFORMIX and ORACLE provide the **USER** constant, which identifies the current user connected to the database server.

Example:

INFORMIX: `SELECT **USER** FROM systables WHERE tabid=1`
ORACLE: `SELECT **USER** FROM DUAL`

However, there is a difference:

- INFORMIX returns the user identifier as defined in the operating system, where it can be case sensitive (UNIX) or not (NT).
- ORACLE returns the user identifier which is stored in the database. By default ORACLE converts the user name to uppercase letters, if you do not put the user name in double quotes when creating it.

This is important if your application stores user names in database records (for example, to audit data modifications). You can, for example, connect to ORACLE with the name 'scott', and perform the following SQL operations :

```
(1) INSERT INTO mytab ( creator, comment )
          VALUES ( USER, 'example' );
(2) SELECT * FROM mytab
           WHERE creator = 'scott';
```

The first command inserts 'SCOTT' (in uppercase letters) in the creator column. The second statement will not find the row.

Solution:

When creating a user in ORACLE, you can put double quotes around the user name in order to force ORACLE to store the given user identifier as is :

```
CREATE USER "scott" IDENTIFIED BY <pswd>
```

To verify the user names defined in the ORACLE database, connect as SYSTEM and list the records of the ALL_USERS table as follows :

```
SELECT * FROM ALL_USERS
```

```
USERNAME     USER_ID      CREATED
-------------------------------------------------------------
SYS                0      02-OCT-98
SYSTEM             5      02-OCT-98
DBSNMP            17      02-OCT-98
FBDL              20      03-OCT-98
Toto              21      03-OCT-98
```

ODIORA051 - Setup database statistics

INFORMIX provides a special instruction to compute database statistics in order to help the optimizer to find the right query execution plan :

```
UPDATE STATISTICS ...
```

Oracle has the following instruction to compute database statistics:

```
ANALYZE ...
```

See Oracle documentation for more details.

Solution:

Centralize the optimization instruction in a function.

ODIORA052 - The GROUP BY clause

INFORMIX allows you to use column numbers in the GROUP BY clause

```
SELECT ord_date, sum(ord_amount) FROM order GROUP BY 1
```

Oracle does not support column numbers in the GROUP BY clause.

Solution:

Use column names instead:

```
SELECT ord_date, sum(ord_amount) FROM order GROUP BY ord_date
```

ODIORA053 - The ALTER TABLE instruction

INFORMIX and ORACLE have different implementations of the ALTER TABLE instruction. For example, INFORMIX allows you to use multiple ADD clauses separated by commas. This is not supported by ORACLE :

INFORMIX:
```
     ALTER TABLE customer ADD(col1 INTEGER), ADD(col2 CHAR(20))
```

ORACLE:
```
     ALTER TABLE customer ADD(col1 INTEGER, col2 CHAR(20))
```

Solution:

Warning: No automatic conversion is done by the database interface. There is no real standard for this instruction (that is, no common syntax for all database servers). Read the SQL documentation and review the SQL scripts or the BDL programs in order to use the database server specific syntax for ALTER TABLE.

ODIORA054 - The star (asterisk) in SELECT statements

Informix allows you to use the star character in the select list along with other expressions :

```
   SELECT col1, * FROM tab1 ...
```

Oracle does not support this. You must use the table name as a prefix to the star:

```
SELECT col1, tab1.* FROM tab1 ...
```

Solution:

Always use the table name before the star.

ODIORA055 - NULLs in indexed columns

Oracle btree indexes do not store null values, while Informix btree indexes do. This means that if you index a single column and select all the rows where that column is null, Informix will do an indexed read to fetch just those rows, but Oracle will do a sequential scan of all rows to find them. Having an index unusable for "is null" criteria can also completely change the behavior and performance of more complicated selects without causing a sequential scan.

Solution:

Declare the indexed columns as NOT NULL with a default value and change the programmatic logic. If you do not want to change the programs, partitioning the table so that the nulls have a partition of their own will reduce the sequential scan to just the nulls (un-indexed) partition, which is relatively fast.

ODIORA056 - SQL Interruption

With Informix, it is possible to interrupt a long running query if the SQL INTERRUPT ON option is set by the Genero program. The database server returns SQLCODE -213, which can be trapped to detect a user interruption.

```
MAIN
  DEFINE n INTEGER
  DEFER INTERRUPT
  OPTIONS SQL INTERRUPT ON
  DATABASE test1
  WHENEVER ERROR CONTINUE
  -- Start long query (self join takes time)
  -- From now on, user can hit CTRL-C in TUI mode to stop the query
  SELECT COUNT(*) INTO n FROM customers a, customers b
       WHERE a.cust_id <> b.cust_id
  IF SQLCA.SQLCODE == -213 THEN
     DISPLAY "Statement was interrupted by user..."
     EXIT PROGRAM 1
  END IF
  WHENEVER ERROR STOP
  ...
END MAIN
```

Oracle supports SQL Interruption in a similar way as Informix. The db client must issue an OCIBreak() OCI call to interrupt a query.

Solution:

The ORACLE database driver supports SQL interruption and converts the native SQL error code -1013 to the Informix error code -213.

ODIORA100 - Data type conversion table

INFORMIX Data Types	ORACLE Data Types
CHAR(n)	CHAR(n) (Oracle limit = 2000c!)
VARCHAR(n)	VARCHAR2(n) (Oracle limit = 4000c!)
INTEGER	NUMBER(10)

SMALLINT	NUMBER(5)
FLOAT[(n)]	NUMBER
SMALLFLOAT	NUMBER
DECIMAL(p,s)	NUMBER(p,s)
DECIMAL(p)	FLOAT(p*3.32193)
MONEY(p,s)	NUMBER(p,s)
TEXT	CLOB (Oracle >=8i only)
BYTE	BLOB (Oracle >=8i only)
DATE	DATE
DATETIME YEAR TO YEAR	DATE
DATETIME YEAR TO MONTH	DATE
DATETIME YEAR TO DAY	DATE
DATETIME YEAR TO HOUR	DATE
DATETIME YEAR TO MINUTE	DATE
DATETIME YEAR TO SECOND	DATE
DATETIME YEAR TO FRACTION(n)	TIMESTAMP(n) (Oracle >=9i only)
DATETIME MONTH TO MONTH	DATE
DATETIME MONTH TO DAY	DATE
DATETIME MONTH TO HOUR	DATE
DATETIME MONTH TO MINUTE	DATE
DATETIME MONTH TO SECOND	DATE
DATETIME MONTH TO FRACTION(n)	(Oracle >=9i only)
DATETIME DAY TO DAY	DATE
DATETIME DAY TO HOUR	DATE
DATETIME DAY TO MINUTE	DATE
DATETIME DAY TO SECOND	DATE
DATETIME DAY TO FRACTION(n)	TIMESTAMP(n) (Oracle >=9i only)
DATETIME HOUR TO HOUR	DATE
DATETIME HOUR TO MINUTE	DATE
DATETIME HOUR TO SECOND	DATE
DATETIME HOUR TO FRACTION(n)	TIMESTAMP(n) (Oracle >=9i only)
DATETIME MINUTE TO MINUTE	DATE
DATETIME MINUTE TO SECOND	DATE
DATETIME MINUTE TO FRACTION(n)	TIMESTAMP(n) (Oracle >=9i only)
DATETIME SECOND TO SECOND	DATE
DATETIME SECOND TO	TIMESTAMP(n) (Oracle >=9i

FRACTION(n)	only)
DATETIME FRACTION TO FRACTION(n)	TIMESTAMP(n) (Oracle >=9i only)
INTERVAL YEAR[(n)] TO MONTH	Oracle >=9i: INTERVAL YEAR[(n)] TO MONTH Oracle <9i: CHAR(50)
INTERVAL MONTH[(n)] TO MONTH	CHAR(50)
INTERVAL DAY[(n)] TO FRACTION(p)	Oracle >=9i: INTERVAL DAY[(n)] TO SECOND(p) Oracle <9i: CHAR(50)
INTERVAL HOUR[(n)] TO HOUR	CHAR(50)
INTERVAL HOUR[(n)] TO MINUTE	CHAR(50)
INTERVAL HOUR[(n)] TO SECOND	CHAR(50)
INTERVAL HOUR[(n)] TO FRACTION(p)	CHAR(50)
INTERVAL MINUTE[(n)] TO MINUTE	CHAR(50)
INTERVAL MINUTE[(n)] TO SECOND	CHAR(50)
INTERVAL MINUTE[(n)] TO FRACTION(p)	CHAR(50)
INTERVAL SECOND[(n)] TO SECOND	CHAR(50)
INTERVAL SECOND[(n)] TO FRACTION(p)	CHAR(50)
INTERVAL FRACTION[(n)] TO FRACTION	CHAR(50)

ODI Adaptation Guide For SQL Server 2000, 2005, 2008

Runtime configuration

- Install SQL Server and create a database
- Prepare the runtime environment

Database concepts

- Database concepts
- Data storage concepts
- Data consistency and concurrency management
- Transactions handling
- Defining database users
- Setting privileges

Data dictionary

- CHARACTER data types
- NUMERIC data types
- DATE and DATETIME data types
- INTERVAL data type
- SERIAL data type
- ROWIDs
- Case sensitivity
- Very large data types
- National character data types
- The ALTER TABLE instruction
- Constraints
- Triggers
- Stored procedures
- Name resolution of SQL objects
- Setup database statistics
- Data type conversion table

Data manipulation

Reserved words
Outer joins
Transactions handling
Temporary tables
Substrings in SQL
Name resolution of SQL objects
String delimiters
Getting one row with SELECT
MATCHES and LIKE conditions
Querying system catalog tables
Syntax of UPDATE statements
The LENGTH() function

BDL programming

Executing SQL statements
SERIAL data type
INFORMIX specific SQL statements in BDL
INSERT cursors
Cursors WITH HOLD
SELECT FOR UPDATE
The LOAD and UNLOAD instructions
SQL Interruption

Runtime configuration

Install SQL Server and create a database

1. Install the Microsoft SQL Server on your computer.
2. Create a SQL Server **database** entity with the SQL Server Management Studio.

   In the database properties:
   - Choose the right **code page / collation** to get a case-sensitive database; this cannot be changed later.
   - Make sure the **"ANSI NULL Default"** option is **TRUE** if you want to have the same default NULL constraint as in INFORMIX (i.e. a column created without NULL

constraint will allow null values, users must specify NOT NULL to deny nulls).
    - Make sure the **"Quoted Identifiers Enabled"** option is **FALSE** to use database object names without quotes as in INFORMIX.
3. Create and declare a database user dedicated to your application: the **application administrator**.
4. If you plan to use SERIAL emulation based on triggers using a registration table, create the SERIALREG table and create the serial triggers for all tables using a SERIAL. See issue ODIMSV005 for more details.
5. Create the **application tables**. Do not forget to convert INFORMIX data types to SQL Server data types. See issue ODIMSV100 for more details.
   Warning: In order to make application tables visible to all users, make sure that the tables are created with the '**dbo**' owner.

Prepare the runtime environment

1. **Warning: Genero FGL 2.11 provides three kind of SQL Server drivers identified by the following codes: MSV, SNC and FTM.**
   All drivers are based on the ODBC API.
    - **The MSV driver works with the Microsoft Data Access Component ODBC driver (SQLSVR32.DLL), and can be used with SQL Server 2000.**
      The MSV driver is supported since first versions of Genero FGL, but is not available for SQL Server 2008.
    - **If you have SQL Server 2005 (or higher) it is recommended to use the SNC driver based on the new SQL Native Client ODBC driver (SQLNCLI.DLL). This is the new ODBC driver recommended by Microsoft for SQL Server 2005 and +.**
      Note that the SNC driver is not supported in a VC++ 6 environment.
      The SNC driver is supported starting from Genero FGL 2.10.
    - **If you need to connect from a UNIX platform to SQL Server, you can use the FTM driver. This driver is based on the FreeTDS client open source software (www.freetds.org). You need at least FreeTDS version 0.82.**

The FTM driver is supported starting from Genero FGL 2.11.

2. An ODBC data source must be configured to allow BDL program to establish connections to SQL Server. Make sure you select the correct ODBC driver (**MSV** = "SQL Server", **SNC** = "SQL Native Client", **FTM** = "FreeTDS").
   Warning: When using the **FTM** driver (FreeTDS), you have to define the ODBCINI and ODBCINST environment variable to point to the odbc.ini and odbcinst.ini files.
3. When using an **MSV** or **SNC** driver, you must have the **Microsoft SQL Server Native Client** installed on the computer running Genero applications.
4. When using the **FTM** driver, you must install **FreeTDS**. Note that in this case, there is no need to install a driver manager like unixODBC: The **FTM** driver is linked directly with the **libtdsodbc.so** shared library. However, you must create the odbc.ini and odbcinst.ini files to defined the data source. See FreeTDS documentation for more details about the data source configuration in ODBC files.
5. **Warning:** On Windows platforms, BDL programs are executed in a CONSOLE environment, not a GUI environment. CONSOLE and GUI environments may use different code pages on your system. Start the **SQL Server Configuration Manager** to setup your client environment and make sure no wrong character conversion occurs. See Microsoft SQL Server documentation for more details.
6. If needed, set up the **fglprofile** entries for database connections.
7. Check that the Genero distribution package has installed the database driver you need (i.e. a "**dbmmsv***", "**dbmsnc***" or "**dbmftm***" driver must be installed.

ODIMSV001 - DATE and DATETIME data types

INFORMIX provides two data types to store dates and time information:

- **DATE** = for year, month and day storage.
- **DATETIME** = for year to fraction(1-5) storage.

Microsoft SQL Server provides two data type to store dates:

- **DATETIME** = for year, month, day, hour, min, second, fraction(3) storage (from January 1, 1753 through December 31, 9999). Values are rounded to increments of .000, .003, or .007 seconds.
- **SMALLDATETIME** = for year, month, day, hour, minutes storage (from January 1, 1900, through June 6, 2079). Values with 29.998 seconds or lower are rounded down to the nearest minute; values with 29.999 seconds or higher are rounded up to the nearest minute.

Starting with Microsoft SQL Server 2008, following new date data types are available:

- **DATE** = for year, month, day storage as Informix DATEs.
- **TIME(n)** = for hour, minute, second and fraction(7) storage. Here **n** defines the precision of fractional seconds.
- **DATETIME2(n)** = for year, month, day, hour, minute, second and fraction(7) storage. Here **n** defines the precision of fractional seconds.
- **DATETIMEOFFSET(n)** = for year, month, day, hour, minute, second, fraction(7) and time zone information storage. Here **n** defines the precision of fractional seconds.

String representing date time information:

INFORMIX is able to convert quoted strings to DATE / DATETIME data if the string contents matches environment parameters (i.e. DBDATE, GL_DATETIME). As in INFORMIX, Microsoft SQL Server can convert quoted strings to DATETIME data. The CONVERT( ) SQL function allows you to convert strings to dates.

Date time arithmetic:

- INFORMIX supports date arithmetic on DATE and DATETIME values. The result of an arithmetic expression involving dates/times is a number of days when only DATEs are used and an INTERVAL value if a DATETIME is used in the expression.
- INFORMIX automatically converts an integer to a date when the integer is used to set a value of a date column. Microsoft SQL Server does not support this automatic conversion.
- Complex DATETIME expressions (involving INTERVAL values for example) are INFORMIX specific and have no equivalent in Microsoft SQL Server.
- Microsoft SQL Server does not allow direct arithmetic operations on datetimes; the date handling SQL functions must be used instead (DATEADD & DATEDIFF).
- The SQL Server provides equivalent functions for YEAR(), MONTH() and DAY(). Be careful with the DAY(*n*) function on SQL Server because it begins from January 1, 1900 while INFORMIX begins from December 31, 1899.

INFORMIX	Microsoft SQL Server

`select day(0), month(0), year(0) FROM systables WHERE tabid=1;`
`------ ------ ------`
	`select day(0), month(0), year(0)`
	`----------- ----------- -----`
	`------`

```
    31     12   1899                    1
1 Row(s) affected             1        1900
                              (1 row(s) affected)
```

- The SQL Server equivalent for WEEKDAY() is the DATEPART(dw,<date>) function. The weekday date part depends on the value set by SET DATEFIRST *n*, which sets the first day of the week (1=Monday...7=Sunday-default).
- SQL Server uses a different basis for the day of the week. In SQL Server, Sunday is day 7 and Monday is day 1 while INFORMIX defines Sunday as the day 0 (zero) and Monday as 1.

Solution:

The SQL Server drivers will automatically map Informix date/time types to native SQL Server type, according the the server version. Conversions are described in this table:

INFORMIX date/time type	Microsoft SQL Server date/time type	
	Before SQL Server 2008	Since SQL Server 2008
DATE	DATETIME	DATE
DATETIME HOUR TO SECOND	DATETIME (filled with 1900-01-01)	TIME(0)
DATETIME HOUR TO FRACTION(n)	DATETIME (filled with 1900-01-01)	TIME(n)
DATETIME YEAR TO SECOND	DATETIME	DATETIME2(0)
Any other sort of DATETIME type	DATETIME (filled with 1900-01-01)	DATETIME2(n)

With SQL Server 2005 and lower, INFORMIX DATETIME with any precision from YEAR to FRACTION(3) is stored in SQL Server DATETIME columns.

For heterogeneous DATETIME types like DATETIME HOUR TO MINUTE, the database interface fills missing date or time parts to 1900-01-01 00:00:00.0. For example, when using a DATETIME HOUR TO MINUTE with the value of "11:45", the SQL Server datetime value will be "1900-01-01 11:45:00.0".

Warning: SQL Server SMALLDATETIME can store dates from January 1, 1900, through June 6, 2079. Therefore, we do not recommend to use this data type.

Warning: With SQL Server 2005 and lower, the fractional second part of a SQL Server DATETIME has a precision of 3 digits while INFORMIX has a precision up to 5 digits. Do not try to insert a datetime value in a SQL Server DATETIME with a precision more than 3 digits or a conversion error could occur. You can use the MS SUBSTRING() function to truncate the fraction part of the INFORMIX datetimes or another BDL solution. The fraction part of a SQL Server DATETIME is an approximate value. For example, when you insert a datetime value with a fraction of 111, the database actually stores 110. This may cause problems because INFORMIX DATETIMEs with a fraction part are exact values with a precision up to 5 digits. Starting with SQL Server 2008, the DATETIME2

native type will be used. This new type can store fraction of seconds with a precision of 7 digits, so Informix DATETIME values can be stored without precision lost.

Warning: When migrating to SQL Server 2008, you must pay attention if the database has DATETIME columns used to store Informix DATETIME HOUR TO SECOND or DATETIME HOUR TO FRACTION(n) types: Before version 2008, those types were stored in SQL Server DATETIME columns (filling missing date part with 1900-01-01). The SNC driver for SQL Server 2008 maps now DATETIME HOUR TO SECOND / FRACTION(n) to a TIME data type, which is not compatible with an SQL Server DATETIME type. To solve this problem, SQL Server DATETIME columns used to store DATETIME HOUR TO SECOND/FRACTION(n) must be converted to TIME columns (ALTER TABLE).

Warning: When fetching a TIME or DATETIME2 with a precision that is greater as 5 (the 4gl DATETIME precision limit), the database interface will allocate a buffer of VARCHAR(16) for the TIME and VARCHAR(27) for the DATETIME2 column. As a result, you can fetch such data into a CHAR or VARCHAR variable.

Warning: Using integers as a number of days in an expression with dates is not supported by SQL Server. Check your code to detect where you are using integers with DATE columns.

Warning: Literal DATETIME and INTERVAL expressions (i.e. DATETIME (1999-10-12) YEAR TO DAY) are not converted.

Warning: It is strongly recommended to use BDL variables in dynamic SQL statements instead of quoted strings representing DATEs. For example :

```
   LET stmt = "SELECT ... FROM customer WHERE creat_date >'", adate,"'"
```

is not portable; use a question mark place holder instead and OPEN the cursor USING adate:

```
   LET stmt = "SELECT ... FROM customer WHERE creat_date > ?"
```

Warning: Review the program logic if you are using the INFORMIX WEEKDAY() function because SQL Server uses a different basis for the days numbers (Monday = 1).

Warning: SQL Statements using expressions with TODAY / CURRENT / EXTEND must be reviewed and adapted to the native syntax. Use the MS GETDATE() function to get the system current date.

ODIMSV003 - Reserved words

Microsoft Transact-SQL does not allow you to use reserved words as database object names (tables, columns, constraint, indexes, triggers, stored procedures, ...). An example of a common word which is part of SQL Server grammar is '**go**' (see the 'Reserved keywords' section in the SQL Server Documentation).

Solution:

Database objects having a name which is a Transact-SQL reserved word must be renamed.

All BDL application sources must be verified. To check if a given keyword is used in a source, you can use UNIX 'grep' or 'awk' tools. Most modifications can be automatically done with UNIX tools like 'sed' or 'awk'.

Warning: You can use SET QUOTED_IDENTIFIER ON with double-quotes to enforce the use of keywords in the database objects naming, but it is not recommended.

ODIMSV004 - ROWIDs

When creating a table, INFORMIX automatically adds a "ROWID" column of type integer (applies to non-fragmented tables only). The ROWID column is auto-filled with a unique number and can be used like a primary key to access a given row.

Microsoft SQL Server tables have no ROWIDs.

Solution:

If the BDL application uses ROWIDs, the program logic should be reviewed in order to use the real primary keys (usually, serials which can be supported).

However, if your existing INFORMIX application depends on using ROWID values, you can use the IDENTITY property of the DECIMAL, INT, NUMERIC, SMALLINT, BIGINT, or TINYINT data types, to simulate this functionality.

All references to SQLCA.SQLERRD[6] must be removed because this variable will not hold the ROWID of the last INSERTed or UPDATEd row when using the Microsoft SQL Server interface.

ODIMSV005 - SERIAL data type

INFORMIX SERIAL data type and automatic number production:

- The table column must be of type SERIAL.
- To generate a new serial, **no value** or a **zero** value is given to the INSERT statement:
```
INSERT INTO tab1 ( c ) VALUES ( 'aa' )
INSERT INTO tab1 ( k, c ) VALUES ( 0, 'aa' )
```
- After INSERT, the new SERIAL value is provided in **`SQLCA.SQLERRD[2]`**.

INFORMIX allows you to insert rows with a value different from zero for a serial column. Using an explicit value will automatically increment the internal serial counter, to avoid conflicts with future INSERTs that are using a zero value:

```
CREATE TABLE tab ( k SERIAL );  --> internal counter = 0
INSERT INTO tab VALUES ( 0 );   --> internal counter = 1
INSERT INTO tab VALUES ( 10 );  --> internal counter = 10
INSERT INTO tab VALUES ( 0 );   --> internal counter = 11
DELETE FROM tab;                --> internal counter = 11
INSERT INTO tab VALUES ( 0 );   --> internal counter = 12
```

Microsoft SQL Server **IDENTITY** columns:

- When creating a table, the IDENTITY keyword must be specified after the column data type:
  ```
  CREATE TABLE tab1 ( k integer identity, c char(10) )
  ```
- You can specify a start value and an increment with "identity(start,incr)".
  ```
  CREATE TABLE tab1 ( k integer identity(100,2), ...
  ```
- A new number is automatically created when inserting a new row:
  ```
  INSERT INTO tab1 ( c ) VALUES ( 'aaa' )
  ```
- To get the last generated number, Microsoft SQL Server provides a global variable:
  ```
  SELECT @@IDENTITY
  ```
- To put a specific value into a IDENTITY column, the SET command must be used:
  ```
  SET IDENTITY_INSERT tab1 ON
  INSERT INTO tab1 ( k, c ) VALUES ( 100, 'aaa' )
  SET IDENTITY_INSERT tab1 OFF
  ```

INFORMIX SERIALs and MS SQL Server IDENTITY columns are quite similar; the main difference is that MS SQL Server does not allow you to use the zero value for the identity column when inserting a new row.

This problem cannot be resolved with triggers because Microsoft SQL Server does not support row-level triggers (INSERT Triggers are fired only once per INSERT statement).

Solution:

To emulation INFORMIX serials, you can use **IDENTITY columns** (1) or **insert triggers based on the SERIALREG** table (2). The first solution is faster, but does not allow explicit serial value specification in insert statements; the second solution is slower but allows explicit serial value specification.

Warning: The second emulation based on triggers is provided to simplify the conversion to SQL Server. We strongly recommend you to use native IDENTITY columns instead.

With the following fglprofile entry, you define the technique to be used for SERIAL emulation :

```
   dbi.database.<dbname>.ifxemul.datatype.serial.emulation =
{"native"|"regtable"}
```

The '**native**' value defines the IDENTITY column technique and the '**regtable**' defines the trigger technique.

This entry must be used with:

```
   dbi.database.<dbname>.ifxemul.datatype.serial = {true|false}
```

If this entry is set to false, the emulation method specification entry is ignored.

Warning: When no entry is specified, the default is SERIAL emulation enabled with 'native' method (IDENTITY-based).

1. Using IDENTITY columns

In database creation scripts, all SERIAL[(n)] data types must be converted by hand to INTEGER IDENTITY[(n,1)] data types.

Tables created from the BDL programs can use the SERIAL data type : When a BDL program executes a CREATE [TEMP] TABLE with a SERIAL column, the database interface automatically converts the "SERIAL[(n)]" data type to "INTEGER IDENTITY[(n,1)]".

In BDL, the new generated SERIAL value is available from the SQLCA.SQLERRD[2] variable. This is supported by the database interface which performs a "SELECT @@IDENTITY".

Warning: By default (see SET IDENTITY_INSERT), MS SQL Server does not allow you to specify the IDENTITY column in INSERT statements; you must convert all INSERT statements to remove that column from the list.
For example, the following statement:

```
   INSERT INTO tab (col1,col2) VALUES (0, p_value)
```

must be converted to :

```
   INSERT INTO tab (col2) VALUES (p_value)
```

Since 2.10.06, SELECT * FROM table INTO TEMP with original table having an IDENTITY column are supported: The database driver converts the INFORMIX SELECT INTO TEMP to the following sequence of statements:

1. SELECT <*selection items*> INTO #table FROM ... WHERE 1=2
2. SET IDENTITY_ INSERT #table ON
3. INSERT INTO #table (*column-list*) SELECT <*original select clauses*>
4. SET IDENTITY_ INSERT #table OFF

See also temporary tables.

2. Using triggers with the SERIALREG table

First, you must prepare the database and create the SERIALREG table as follows:

```
CREATE TABLE serialreg (
     tablename VARCHAR(50) NOT NULL,
     lastserial INTEGER NOT NULL,
     PRIMARY KEY ( tablename )
)
```

Warning: Note that the SERIALREG table and columns have to be created with lower case names, since the SQL Server database is created with case sensitive names, because triggers are using this table in lower case.

In database creation scripts, all SERIAL[(n)] data types must be converted to INTEGER data types and you must create one trigger for each table. To know how to write those triggers, you can create a small Genero program that creates a table with a SERIAL column. Set the FGLSQLDEBUG environment variable and run the program. The debug output will show you the native trigger creation command.

Tables created from the BDL programs can use the SERIAL data type. When a BDL program executes a CREATE [TEMP] TABLE with a SERIAL column, the database interface automatically converts the "SERIAL[(n)]" data type to "INTEGER" and creates the insert triggers.

Warning: This serial emulation is only supported with **SQL Server 2000** and higher, because it is implemented with INSTEAD OF triggers.

Warning: SQL Server does not allow you to create triggers on temporary tables. Therefore, you cannot create temp tables with a SERIAL column when using this solution.

Warning: SELECT ... INTO TEMP statements using a table created with a SERIAL column do not automatically create the SERIAL triggers in the temporary table. The type of the column in the new table is INTEGER.

Warning: When a table is dropped, all associated triggers are also dropped.

Warning: INSERT statements using NULL for the SERIAL column will produce a new serial value, instead of using NULL:

```
INSERT INTO tab (col1,col2) VALUES (NULL,'data')
```

This behavior is mandatory in order to support INSERT statements which do not use the serial column:

```
INSERT INTO tab (col2) VALUES ('data')
```

Check if your application uses tables with a SERIAL column that can contain a NULL value.

Warning: The serial production is based on the SERIALREG table which registers the last generated number for each table. If you delete rows of this table, sequences will restart at 1 and you will get unexpected data.

ODIMSV006 - Outer joins

The syntax of OUTER joins is quite different in INFORMIX and Microsoft SQL Server :

In INFORMIX SQL, outer tables are defined in the FROM clause with the **OUTER** keyword:

```
SELECT ... FROM cust, OUTER(order)
 WHERE cust.key = order.custno
SELECT ... FROM cust, OUTER(order,OUTER(item))
 WHERE cust.key = order.custno
   AND order.key = item.ordno
   AND order.accepted = 1
```

Microsoft SQL Server supports the ANSI outer join syntax :

```
SELECT ... FROM cust LEFT OUTER JOIN order
                    ON cust.key = order.custno
SELECT ...
  FROM cust LEFT OUTER JOIN order
            ON cust.key = order.custno
            LEFT OUTER JOIN item
            ON order.key = item.ordno
 WHERE order.accepted = 1
```

Remark: The old way to define outers in SQL Server looks like the following :

```
SELECT ... FROM a, b WHERE a.key *= b.key
```

See the SQL Server reference manual for a complete description of the syntax.

Solution:

The Microsoft SQL Server interface can convert simple INFORMIX OUTER specifications to Microsoft SQL Server ANSI outer joins.

Prerequisites:

1. The outer join in the WHERE part must use the table name as prefix.
   Example : "WHERE tab1.col1 = tab2.col2 ".
2. Additional conditions on outer table columns cannot be detected and therefore are not supported :
   Example : "... FROM tab1, OUTER(tab2) WHERE tab1.col1 = tab2.col2 AND tab2.colx > 10".
3. Statements composed of 2 or more SELECT instructions using OUTERs are not supported.
   Example : "SELECT ... UNION SELECT" or "SELECT ... WHERE col IN (SELECT...)"

Remarks:

1. Table aliases are detected in OUTER expressions.
   OUTER example with table alias : "OUTER(tab1 alias1)".
2. In the outer join, <outer table>.<col> can be placed on both right or left sides of the equal sign.
   OUTER join example with table on the left : "WHERE outertab.col1 = maintab.col2 ".
3. Table names detection is not case-sensitive.
   Example : "SELECT ... FROM tab1, TAB2 WHERE tab1.col1 = tab2.col2".
4. Temporary tables are supported in OUTER specifications.

ODIMSV007a - Database concepts

As in INFORMIX, an SQL Server engine can manage multiple database entities. When creating a database object like a table, Microsoft SQL Server allows you to use the same object name in different databases.

ODIMSV008a - Data consistency and concurrency management

Data consistency involves readers which want to access data currently modified by writers and concurrency data access involves several writers accessing the same data for modification. Locking granularity defines the amount of data concerned when a lock is set (row, page, table, ...).

INFORMIX

INFORMIX uses a locking mechanism to manage data consistency and concurrency. When a process modifies data with UPDATE, INSERT or DELETE, an exclusive lock is set on the affected rows. The lock is held until the end of the transaction. Statements performed outside a transaction are treated as a transaction containing a single operation and therefore release the locks immediately after execution. SELECT statements can set shared locks according to the isolation level. In case of locking conflicts (for example, when two processes want to acquire an exclusive lock on the same row for modification or when a writer is trying to modify data protected by a shared lock), the behavior of a process can be changed by setting the lock wait mode.

Control:

- Isolation level : SET ISOLATION TO ...
- Lock wait mode : SET LOCK MODE TO ...
- Locking granularity : CREATE TABLE ... LOCK MODE {PAGE|ROW}
- Explicit locking : SELECT ... FOR UPDATE

Defaults:

- The default isolation level is READ COMMITTED.
- The default lock wait mode is NOT WAIT.
- The default locking granularity is per page.

SQL Server

As in INFORMIX, SQL Server uses locks to manage data consistency and concurrency. The database manager sets exclusive locks on the modified rows and shared locks when data is read, according to the isolation level. The locks are held until the end of the transaction. When multiple processes want to access the same data, the latest processes must wait until the first finishes its transaction or the lock timeout occurred. The locking strategy of SQL Server is row locking with possible promotion to page or table locking. SQL Server dynamically determines the appropriate level at which to place locks for each Transact-SQL statement.

Control:

- Lock wait mode : SET LOCK_TIMEOUT <milliseconds> (returns error 1222 on time out).
- Isolation level : SET TRANSACTION ISOLATION LEVEL ...
- Locking granularity : Row, Page or Table level (Automatic - See Dynamic Locking).
- Explicit locking : SELECT ... FROM ... WITH (UPDLOCK) (See Locking Hints)

Defaults:

- The default isolation level is READ COMMITTED (readers cannot see uncommitted data).
- The default LOCK_TIMEOUT is -1 (indicates no time-out period, wait forever).

Solution:

For portability, it is recommended that you work with INFORMIX in the read committed isolation level, to make processes wait for each other (lock mode wait) and to create tables with the "lock mode row" option.

See INFORMIX and SQL Server documentation for more details about data consistency, concurrency and locking mechanisms.

When using SET LOCK MODE and SET ISOLATION LEVEL instructions in BDL, the database interface sets automatically the native database session options.

ODIMSV008b - SELECT FOR UPDATE

A lot of BDL programs use pessimistic locking in order to avoid several users editing the same rows at the same time.

```
DECLARE cc CURSOR FOR
      SELECT ... FOR UPDATE
OPEN cc
FETCH cc <-- lock is acquired
CLOSE cc <-- lock is released
```

- A transaction must be started before opening cursors declared for update.
- The row must be fetched in order to set the lock.
- The lock is released when the transaction ends (if the cursor is not declared "WITH HOLD") or when the cursor is closed.

Microsoft SQL Server allows individual and exclusive row locking by using the (UPDLOCK) hint after the table names in the FROM clause :

```
SELECT ... FROM tab1 WITH (UPDLOCK) WHERE ...
```

The FOR UPDATE clause is not mandatory; the (UPDLOCK) hint is important.

- Individual locks are acquired when fetching the rows.
- When the cursor (WITH HOLD) is opened outside a transaction, locks are released when the cursor is closed.
- When the cursor is opened inside a transaction, locks are released when the transaction ends.

SQL Server's locking granularity is at the row level, page level or table level (the level is automatically selected by the engine for optimization).

To control the behavior of the program when locking rows, INFORMIX provides a specific instruction to set the wait mode :

```
SET LOCK MODE TO { WAIT | NOT WAIT | WAIT seconds }
```

The default mode is NOT WAIT. This as an INFORMIX specific SQL statement.

Solution:

The Genero Driver for MS SQL Server uses the SCROLL LOCKS concurrency options for cursors (SQL_ATTR_CONCURRENCY = SQL_CONCUR_LOCK).

This option implements pessimistic concurrency control, in which the application attempts to lock the underlying database rows at the time they are read into the cursor result set.
When using server cursors, an update lock is placed on the row when it is read into the cursor.
If the cursor is opened within a transaction, the transaction update lock is held until the transaction is either committed or rolled back; the cursor lock is dropped when the next row is fetched.
If the cursor has been opened outside a transaction, the lock is dropped when the next row is fetched.
Therefore, a cursor should be opened in a transaction whenever the user wants full

pessimistic concurrency control.
An update lock prevents any other task from acquiring an update or exclusive lock, which prevents any other task from updating the row.
An update lock, however, does not block a shared lock, so it does not prevent other tasks from reading the row unless the second task is also requesting a read with an update lock.

SELECT FOR UPDATE statements are well supported in BDL as long as they are used inside a transaction. Avoid cursors declared WITH HOLD.

Warning: SQL Server locks the rows when you open the cursor. You will have to test SQLCA.SQLCODE after doing an OPEN.

Warning: The database interface is based on an emulation of an INFORMIX engine using transaction logging. Therefore, opening a SELECT ... FOR UPDATE cursor declared outside a transaction will raise an SQL error -255 (not in transaction).

Warning: The SELECT FOR UPDATE statement cannot contain an ORDER BY clause if you want to perform positioned updates/deletes with WHERE CURRENT OF.

Warning: Cursors declared with SELECT ... FOR UPDATE using the "WITH HOLD" clause cannot be supported with SQL Server.

You must review the program logic if you use pessimistic locking because it is based on the NOT WAIT mode which is not supported by SQL Server.

ODIMSV009 - Transactions handling

INFORMIX and Microsoft SQL Server handle transactions in a similar manner.

INFORMIX native mode (non ANSI):

- Transactions are started with "BEGIN WORK".
- Transactions are validated with "COMMIT WORK".
- Transactions are canceled with "ROLLBACK WORK".
- Statements executed outside of a transaction are automatically committed.
- DDL statements can be executed (and canceled) in transactions.

Microsoft SQL Server:

- Transactions are started with "BEGIN TRANSACTION [name]".
- Transactions are validated with "COMMIT TRANSACTION [name]".
- Transactions are canceled with "ROLLBACK TRANSACTION [name]".
- Transactions save points can be placed with "SAVE TRANSACTION [name]".
- Microsoft SQL Server supports named and nested transactions.

- Statements executed outside of a transaction are automatically committed (autocommit mode).
  This behavior can be changed with "SET IMPLICIT_TRANSACTION ON".
- DDL statements are not supported in transactions blocks.

Transactions in stored procedures : avoid using transactions in stored procedure to allow the client applications to handle transactions, according to the transaction model.

Solution:

INFORMIX transaction handling commands are automatically converted to Microsoft SQL Server instructions to start, validate or cancel transactions.

Regarding the transaction control instructions, the BDL applications do not have to be modified in order to work with Microsoft SQL Server.

ODIMSV011 - CHARACTER data types

As in INFORMIX, Microsoft SQL Server provides the CHAR and VARCHAR data types to store character data.

INFORMIX CHAR type can store up to **32767** characters and the VARCHAR data type is limited to **255** characters.

Microsoft SQL Server CHAR and VARCHAR both have a limit of **8000** characters.

Microsoft SQL server provides the TEXT data type to store large character strings. Only the LIKE operator can be used for searches. TEXT columns cannot be used in classic comparison expressions (as col = 'value').

Solution:

The database interface supports character string variables in SQL statements for input (BDL USING) and output (BDL INTO) up to the limit defined by Microsoft SQL Server for CHAR and VARCHAR data types.

Warning: Check that your database schema does not use CHAR or VARCHAR types with a length exceeding the SQL Server limit.

Warning: TEXT values cannot be used as input or output parameters in SQL statements and therefore are not supported.

See also: National character data types

ODIMSV012 - Constraints

Constraint naming syntax:

Both INFORMIX and Microsoft SQL Server support primary key, unique, foreign key, default and check constraints. But the constraint naming syntax is different : SQL Server expects the "CONSTRAINT" keyword **before** the constraint specification and INFORMIX expects it **after**.

UNIQUE constraint example:

INFORMIX	Microsoft SQL Server

`CREATE TABLE scott.emp (`
`...`
`empcode CHAR(10) UNIQUE`
`  [CONSTRAINT pk_emp],`
`...`
	`CREATE TABLE scott.emp (`
	`...`
	`empcode CHAR(10)`
	`  [CONSTRAINT pk_emp]`
	`UNIQUE,`
	`...`

Warning: SQL Server does not produce an error when using the INFORMIX syntax of constraint naming

The NULL / NOT NULL constraint:

Warning: Microsoft SQL Server creates columns as **NOT NULL by default**, when no NULL constraint is specified (**`colname datatype {NULL | NOT NULL}`**). A special option is provided to invert this behavior: ANSI_NULL_DFLT_ON. This option can be enabled with the SET command, or in the database options of SQL Server Management Studio.

Solution:

Constraint naming syntax:

The database interface does not convert constraint naming expressions when creating tables from BDL programs. Review the database creation scripts to adapt the constraint naming clauses for Microsoft SQL Server.

The NULL / NOT NULL constraint:

Warning: Before using a database, you must check the "ANSI NULL Default" option in the database properties if you want to have the same default NULL constraint as in INFORMIX databases.

ODIMSV013 - Triggers

INFORMIX and Microsoft SQL Server provide triggers with similar features, but the programming languages are totally different.

Warning: Microsoft SQL Server does not support "BEFORE" triggers.

Warning: Microsoft SQL Server does not support row-level triggers.

Solution:

INFORMIX triggers must be converted to Microsoft SQL Server triggers "by hand".

ODIMSV014 - Stored procedures

Both INFORMIX and Microsoft SQL Server support stored procedures, but the programming languages are totally different :

- INFORMIX stored procedures must be written in **SPL**.
- Microsoft SQL Server stored procedures must be written in **Transact-SQL**.

Solution:

INFORMIX stored procedures must be converted to Microsoft SQL Server "by hand".

ODIMSV016a - Defining database users

INFORMIX users are defined at the operating system level, they must be members of the 'informix' group, and the database administrator must grant CONNECT, RESOURCE or DBA privileges to those users.

Before a user can access an SQL Server database, the system administrator (SA) must add the user's **login** to the SQL Server Login list and add a **user name** for that database. The user name is a name that is assigned to a login ID for the purpose of allowing that user to access a specified database. Database users are members of a **user group**; the default group is 'public'.

Microsoft SQL Server offers two authentication modes : The **SQL Server authentication mode**, which requires a login name and a password, and the **Windows NT authentication mode**, which uses the security mechanisms within Windows NT when validating login connections. With this mode, user do not have to enter a login ID and password - their login information is taken directly from the network connection.

Warning: SQL Server 2000 supports only Windows NT authentication by default. If you want to use SQL Server authentication, you must change a parameter in the server properties.

Solution:

Both SQL Server and Windows NT authentication methods can be used to allow BDL program users to connect to Microsoft SQL Server and access a specific database.

See SQL Server documentation for more details on database logins and users.

ODIMSV016b - Setting privileges

INFORMIX and Microsoft SQL Server user privileges management are quite similar.

Microsoft SQL Server provides **user groups** to grant or revoke permissions to more than one user at the same time.

ODIMSV017 - Temporary tables

INFORMIX temporary tables are created through the **`CREATE TEMP TABLE`** DDL instruction or through a **`SELECT ... INTO TEMP`** statement. Temporary tables are automatically dropped when the SQL session ends, but they can also be dropped with the DROP TABLE command. There is no name conflict when several users create temporary tables with the same name.

Remark : BDL reports create a temporary table when the rows are not sorted externally (by the source SQL statement).

INFORMIX allows you to create indexes on temporary tables. No name conflict occurs when several users create an index on a temporary table by using the same index identifier.

Microsoft SQL Server provides local (SQL session wide) or global (database wide) temporary tables by using the '#' or '##' characters as table name prefix. No 'TEMP' keyword is required in CREATE TABLE, and the INTO clause can be used within a SELECT statement to create and fill a temporary table in one step :

```
CREATE TABLE #temp1 ( kcol INTEGER, .... )
SELECT * INTO #temp2 FROM customers WHERE ...
```

Solution:

In BDL, INFORMIX temporary tables instructions are converted to generate native SQL Server temporary tables.

Warning: Microsoft SQL Server does not support scroll cursors based on a temporary table.

ODIMSV018 - Substrings in SQL

INFORMIX SQL statements can use subscripts on columns defined with the character data type:

```
    SELECT ... FROM tab1 WHERE  col1[2,3] = 'RO'
    SELECT ... FROM tab1 WHERE col1[10]  = 'R'   -- Same as
col1[10,10]
    UPDATE tab1 SET col1[2,3] = 'RO' WHERE ...
    SELECT ... FROM tab1 ORDER BY col1[1,3]
```

.. while Microsoft SQL Server provides the SUBSTR( ) function, to extract a substring from a string expression:

```
    SELECT .... FROM tab1 WHERE  SUBSTRING(col1,2,2) = 'RO'
    SELECT SUBSTRING('Some text',6,3) FROM tab1  -- Gives 'tex'
```

Solution:

You must replace all INFORMIX col[x,y] expressions by SUBSTRING(col,x,y-x+1).

Warning: In UPDATE instructions, setting column values through subscripts will produce an error with Microsoft SQL Server:

```
    UPDATE tab1 SET col1[2,3] = 'RO' WHERE ...
```

is converted to:

```
    UPDATE tab1 SET SUBSTRING(col1,2,3-2+1) = 'RO' WHERE ...
```

Warning: Column subscripts in ORDER BY expressions are also converted and produce an error with Microsoft SQL Server:

```
    SELECT ... FROM tab1 ORDER BY col1[1,3]
```

is converted to:

```
    SELECT ... FROM tab1 ORDER BY SUBSTRING(col1,1,3-1+1)
```

ODIMSV019 - Name resolution of SQL objects

INFORMIX uses the following form to identify an SQL object:

```
  [database[@dbservername]:][{owner|"owner"}.]identifier
```

With Microsoft SQL Server, an object name takes the following form:

```
  [[database.]owner.]identifier
```

Object names are limited to 128 characters in SQL Server and cannot start with one of the following characters : @ (local variable) # (temp object).

To support double quotes as string delimiters in SQL Server, switch **OFF** the database option "Use quoted identifiers" in the database properties panel. But quoted table and column names are not supported when this option is OFF.

Solution:

Switch **OFF** the database option "Use quoted identifiers" to support double quoted strings.

Check for single or double quoted table or column names in your source and remove them.

ODIMSV020 - String delimiters

The ANSI string delimiter character is the single quote ('string'). Double quotes are used to delimit database object names ("object-name").

Example: `WHERE "tabname"."colname" = 'a string value'`

INFORMIX allows double quotes as string delimiters, but SQL Server doesn't. This is important, since many BDL programs use that character to delimit the strings in SQL commands.

Note: This problem concerns only double quotes within SQL statements. Double quotes used in BDL string expressions are not subject of SQL compatibility problems.

National character strings:

With SQL Server, all UNICODE strings must be prefaced with an N character:

```
UPDATE cust SET cust_name = N'    ' WHERE cust_id=123
```

If you don't specify the N prefix, SQL Server will convert the characters from the current system locale to the database locale. If the string

Solution:

The SQL Server database interface can automatically replace all double quotes by single quotes.

Escaped string delimiters can be used inside strings like the following:

```
'This is a single quote : '''
'This is a single quote : \''
"This is a double quote : """
"This is a double quote : \""
```

Warning: Database object names cannot be delimited by double quotes because the database interface cannot determine the difference between a database object name and a quoted string !

For example, if the program executes the SQL statement:
`WHERE "tabname"."colname" = "a string value"`
replacing all double quotes by single quotes would produce :
`WHERE 'tabname'.'colname' = 'a string value'`
This would produce an error since `'tabname'.'colname'` is not allowed by ORACLE.

Although double quotes are replaced automatically in SQL statements, you should use only single quotes to enforce portability.

National character strings:

When using the **snc** driver, all string literals of an SQL statement are automatically changed to get the N prefix. Thus, you don't need to add the N prefix by hand in all of your programs. This solution makes by the way your Genero code portable to other databases.

With the **snc** driver, character string data is converted from the current FGL locale to Wide Char (UTF-16), before is it used in an ODBC call such as SQLPrepareW or SQLBindParameter(SQL_C_WCHAR). When fetching character data, the **snc** driver converts from Wide Char to the current FGL locale. The current FGL locale is defined by LANG, and if LANG is not defined, the default is the ANSI Code Page of the system.

ODIMSV021 - NUMERIC data types

Microsoft SQL Server offers numeric data types which are quite similar to INFORMIX numeric data types. The table below shows general conversion rules for numeric data types :

INFORMIX	Microsoft SQL Server
SMALLINT	**SMALLINT**
INTEGER (synonym: INT)	**INTEGER** (synonym: INT)
DECIMAL[(p[,s)] (synonyms: DEC, NUMERIC) DECIMAL(p,s) defines a fixed point decimal where **p** is the total number of significant digits and **s** the number	**DECIMAL[(p[,s)]** (synonyms: DEC, NUMERIC) DECIMAL[(p[,s])] defines a fixed point decimal where **p** is the total number of significant digits and **s** the number

of digits that fall on the right of the decimal point. DECIMAL(p) defines a floating point decimal where **p** is the total number of significant digits. The precision **p** can be from 1 to 32. DECIMAL is treated as DECIMAL(16).	of digits that fall on the right of the decimal point. The maximum precision is 38. Without any decimal storage specification, the precision defaults to 18 and the scale defaults to zero: - DECIMAL in SQL Server = DECIMAL(18,0) in INFORMIX - DECIMAL(p) in SQL Server = DECIMAL(p,0) in INFORMIX
MONEY[(p[,s])	SQL Server provides the MONEY and SMALLMONEY data types, but the currency symbol handling is quite different. Therefore, INFORMIX MONEY columns should be implemented as **DECIMAL** columns in SQL Server.
SMALLFLOAT (synonyms: REAL)	**REAL**
FLOAT[(n)] (synonyms: DOUBLE PRECISION) The precision (n) is ignored.	**FLOAT(n)** (synonyms: DOUBLE PRECISION) Where n must be from 1 to 15.

Solution:

In BDL programs :

When creating tables from BDL programs, the database interface automatically converts INFORMIX data types to corresponding Microsoft SQL Server data types.

Database creation scripts:

- SMALLINT and INTEGER columns do not have to use another data type in SQL Server.
- For DECIMALs, check the precision limit. Always use a precision and a scale.
- Convert MONEY columns to DECIMAL(p,s) columns. Always use a precision and a scale.
- Convert SMALLFLOAT columns to REAL columns.
- Since FLOAT precision is ignored in INFORMIX, convert this data type to FLOAT(15).

ODIMSV022 - Getting one row with SELECT

With INFORMIX, you must use the system table with a condition on the table id :

```
SELECT user FROM systables WHERE tabid=1
```

With SQL Server, you can omit the FROM clause to generate one row only:

```
SELECT user
```

Solution:

Check the BDL sources for "`FROM systables WHERE tabid=1`" and use dynamic SQL to resolve this problem.

ODIMSV024 - MATCHES and LIKE in SQL conditions

INFORMIX supports MATCHES and LIKE in SQL statements, while Microsoft SQL Server supports the LIKE statement only.

The MATCHES operator of INFORMIX uses the star (*), question mark (?) and square braces ([ ]) wildcard characters.
The LIKE operator of SQL Server offers the percent (%), underscore (_) and square braces ([ ]) wildcard characters.

The following substitutions must be made to convert a MATCHES condition to a LIKE condition :

- MATCHES keyword must be replaced by LIKE.
- All '`*`' characters must be replaced by '`%`'.
- All '`?`' characters must be replaced by '_'.

Solution:

Warning: SQL statements using MATCHES expressions must be reviewed in order to use LIKE expressions.

See also: MATCHES operator in SQL Programming.

ODIMSV025 - INFORMIX specific SQL statements in BDL

The BDL compiler supports several INFORMIX specific SQL statements that have no meaning when using Microsoft SQL Server.

Examples:

- CREATE DATABASE dbname **IN dbspace WITH BUFFERED LOG**
- **START DATABASE** (SE only)
- **ROLLFORWARD DATABASE**
- CREATE TABLE ... **IN dbspace WITH LOCK MODE ROW**

Solution:

Review your BDL source and remove all static SQL statements that are INFORMIX specific.

ODIMSV028 - INSERT cursors

INFORMIX supports insert cursors. An "insert cursor" is a special BDL cursor declared with an INSERT statement instead of a SELECT statement. When this kind of cursor is open, you can use the PUT instruction to add rows and the FLUSH instruction to insert the records into the database.

For INFORMIX database with transactions, OPEN, PUT and FLUSH instructions must be executed within a transaction.

Microsoft SQL Server does not support insert cursors.

Solution:

Insert cursors are emulated by the Microsoft SQL Server database interface.

ODIMSV030 - Very large data types

INFORMIX and Genero support the **TEXT** and **BYTE** types. TEXT is used to store large text data, while BYTE is used to store large binary data like images or sound.

Microsoft SQL Server provides **text**, **ntext** and **image** data types to store large data, but these data types are considered as obsolete in SQL Server 2005 and will be removed in a future version. When using SQL Server 2005, Microsoft recommends to user **varchar(max)**, **nvarchar(max)** and **varbinary(max)** data type instead. These "max" data types are not supported with the **msv** database driver, since it is based on MDAC ODBC. You must use the new **snc** driver based on the SQL Native Client ODBC driver shipped with SQL Server 2005.

Solution:

When using the **msv** database driver based on MDAC ODBC, the **TEXT** and **BYTE** data types of a static CREATE TABLE statement are converted to **text** and **image** SQL Server types.

When using the **snc** database driver based on SQL Native Client ODBC, the **TEXT** and **BYTE** data types of a static CREATE TABLE statement are converted to **varchar(max)** and **varbinary(max)** SQL Server types.

Both **msv** and **snc** drivers make the appropriate bindings to use TEXT and BYTE types as SQL parameters and fetch buffers.

ODIMSV031 - Cursors WITH HOLD

INFORMIX automatically closes opened cursors when a transaction ends unless the WITH HOLD option is used in the DECLARE instruction.

Microsoft SQL Server does not close cursors when a transaction ends. You can change this behavior using the SET CURSOR_CLOSE_ON_COMMIT ON.

Solution:

BDL cursors that are not declared "WITH HOLD" are automatically closed by the database interface when a COMMIT WORK or ROLLBACK WORK is performed by the BDL program.

ODIMSV033 - Querying system catalog tables

As in INFORMIX, Microsoft SQL Server provides system catalog tables (sysobjects,syscolumns,etc) in each database, but the table names and their structure are quite different.

Solution:

Warning: No automatic conversion of INFORMIX system tables is provided by the database interface.

ODIMSV034 - Syntax of UPDATE statements

INFORMIX allows a specific syntax for UPDATE statements:

```
UPDATE table SET ( <col-list> ) = ( <val-list> )
```

or

```
UPDATE table SET table.* = myrecord.*
UPDATE table SET * = myrecord.*
```

Solution:

Static UPDATE statements using the above syntax are converted **by the compiler** to the standard form :

```
UPDATE table SET column=value [,...]
```

ODIMSV035 - The LENGTH() function

INFORMIX provides the LENGTH() function:

```
SELECT LENGTH("aaa"), LENGTH(col1) FROM table
```

Microsoft SQL Server has a equivalent function called LEN().

Do not confuse LEN() with DATALEN(), which returns the data size used for storage(number of bytes).

Both INFORMIX and SQL Server ignore trailing blanks when computing the length of a string.

Solution:

You must adapt the SQL statements using LENGTH() and use the LEN() function.

Warning: If you create a user function in SQL Server as follows:

```
create function length(@s varchar(8000))
returns integer
as
begin
return len(@s)
end
```

You must qualify the function with the owner name:

```
SELECT dbo.length(col1) FROM table
```

ODIMSV036 - INTERVAL data type

INFORMIX's INTERVAL data type stores a value that represents a span of time. INTERVAL types are divided into two classes : *year-month intervals* and *day-time intervals.*

SQL Server does not provide a data type corresponding to the INFORMIX INTERVAL data type.

Solution:

Warning: The INTERVAL data type is not well supported because the database server has no equivalent native data type. However, you can store into and retrieve from CHAR columns BDL INTERVAL values.

ODIMSV039 - Data storage concepts

An attempt should be made to preserve as much of the storage information as possible when converting from INFORMIX to Microsoft SQL Server. Most important storage decisions made for INFORMIX database objects (like initial sizes and physical placement) can be reused in an SQL Server database.

Storage concepts are quite similar in INFORMIX and in Microsoft SQL Server, but the names are different.

The following table compares INFORMIX storage concepts to Microsoft SQL Server storage concepts :

INFORMIX	Microsoft SQL Server
Physical units of storage	
The largest unit of physical disk space is a "**chunk**", which can be allocated either as a cooked file (I/O is controlled by the OS) or as raw device (=UNIX partition, I/O is controlled by the database engine). A "dbspace" uses at least one "chunk" for storage. You must add "chunks" to "dbspaces" in order to increase the size of the logical unit of storage.	SQL Server uses "**filegroups**", based on Windows NT operating system files and therefore define the physical location of data.
A "**page**" is the smallest physical unit of disk storage that the engine uses to read from and write to databases. A "chunk" contains a certain number of "pages". The size of a "page" must be equal to the operating system's block size.	As in INFORMIX, SQL Server stores data in "**pages**" with a size fixed at 2Kb in V6.5 and 8Kb in V7 and later.
An "**extent**" consists of a collection of continuous "pages" that the engine uses to allocate both initial and subsequent storage space for database tables. When creating a table, you can specify the first extent size and the	An "**extent**" is a specific number of 8 contiguous pages, obtained in a single allocation. Extents are allocated in the filegroup used by the database.

size of future extents with the EXTENT SIZE and NEXT EXTENT options. For a single table, "extents" can be located in different "chunks" of the same "dbspace".	
Logical units of storage	
A "**table**" is a logical unit of storage that contains rows of data values.	Same concept as INFORMIX.
A "**database**" is a logical unit of storage that contains table and index data. Each database also contains a system catalog that tracks information about database elements like tables, indexes, stored procedures, integrity constraints and user privileges.	Same concept as INFORMIX. When creating a "**database**", you must specify which "database devices" (V6.5) or "filegroup" (V7) has to be used for physical storage.
Database tables are created in a specific "**dbspace**", which defines a logical place to store data. If no dbspace is given when creating the table, INFORMIX defaults to the current database dbspace.	Database tables are created in a database based on "database devices" (V6.5) or a "filegroup" (V7), which defines the physical storage.
The total disk space allocated for a table is the "**tblspace**", which includes "pages" allocated for data, indexes, blobs, tracking page usage within table extents..	No equivalent.
Other concepts	
When initializing an INFORMIX engine, a "**root dbspace**" is created to store information about all databases, including storage information (chunks used, other dbspaces, etc.).	SQL Server uses the "**master**" database to hold system stored procedures, system messages, SQL Server logins, current activity information, configuration parameters of other databases.
The "**physical log**" is a set of continuous disk pages where the engine stores "before-images" of data that has been modified during processing. The "**logical log**" is a set of "**logical-log files**" used to record logical operations during on-line processing. All transaction information is stored in the logical log files if a database has been created with transaction log. INFORMIX combines "physical log" and "logical log" information when	Each database has its own "**transaction log**" that records all changes to the database. The "transaction log" is based on a "database device" (V6.5) or "filegroup" (V7) which is specified when creating the database. SQL Server checks the "transaction logs" for automatic recovery.

doing fast recovery. Saved "logical logs" can be used to restore a database from tape.

ODIMSV040 - National characters data types

INFORMIX offers the NCHAR and NVARCHAR data types to store strings in a localized character set. The only difference between CHAR/VARCHAR and NCHAR/NVARCHAR in Informix is for sorting: N type use the collation order, while normal types use the byte order.

NCHAR/NVARCHAR in SQL Server :

Microsoft SQL Server translates the bit patterns in **char**, **varchar**, and **text** columns to characters using the definitions in the code page installed with SQL Server. Client computers use the code page installed with the operating system to interpret character bit patterns. There are many different code pages. Some characters appear on some code pages, but not on others. Some characters are defined with one bit pattern on some code pages, and with a different bit pattern on other code pages. When you build international systems that must handle different languages, it becomes difficult to pick code pages for all the computers that meet the language requirements of multiple countries. It is also difficult to ensure that every computer performs the correct translations when interfacing with a system using a different code page.

The Unicode specification addresses this problem by using 2 bytes to encode each character. There are enough different patterns (65,536) in 2 bytes for a single specification covering the most common business languages. Because all Unicode systems consistently use the same bit patterns to represent all characters, there is no problem with characters being converted incorrectly when moving from one system to another.

In Microsoft SQL Server, these data types support Unicode data:

- **nchar**
- **nvarchar**
- **ntext**

Use of **nchar**, **nvarchar**, and **ntext** is the same as **char**, **varchar**, and **text**, respectively, except:

- Unicode supports a wider range of characters.
- More space is needed to store Unicode characters (x2).
- The maximum size of **nchar** and **nvarchar** columns is 4,000 characters, not 8,000 characters like **char** and **varchar**.
- Unicode string literals are specified with a leading N. For example: N’ ’

Solution:

National character set data types are not supported properly with the **msv** driver. You must use the **snc** driver based on the SQL Native Client library.

With the **snc** driver, NCHAR / NVARCHAR and NTEXT SQL Server column data types can be used in tables. However, you must use CHAR / VARCHAR / TEXT Genero types for program variable to hold NCHAR, NVARCHAR and NTEXT data. Make sure the size of the program variables is large enough to hold all sort of UNICODE characters in the code page used by the program. For example, using byte length semantics and a UTF-8 code page, an NCHAR(10) value can be hold in a CHAR(40) program variable, because some UTF-8 characters can be encoded on 4 bytes. If you want to store 10 of such characters you will need 40 bytes.

When using the **snc** driver, all string literals of an SQL statement are automatically changed to get the N prefix. Thus, you don't need to add the N prefix by hand in all of your programs. This solution makes by the way your Genero code portable to other databases.

With the **snc** driver, character string data is converted from the current FGL locale to Wide Char (UTF-16), before is it used in an ODBC call such as SQLPrepareW or SQLBindParameter(SQL_C_WCHAR). When fetching character data, the **snc** driver converts from Wide Char to the current FGL locale. The current FGL locale is defined by LANG, and if LANG is not defined, the default is the ANSI Code Page of the system.

ODIMSV041 - Executing SQL statements

The database driver for Microsoft SQL Server is based on ODBC. The ODBC driver implementation provided with SQL Server uses system stored procedures to prepare and execute SQL statements (You can see this with the Profiler).

Some Transact-SQL statements like SET DATEFORMAT have a local execution context effect (for example, when executed in a stored procedure, it is reset to the previous values when procedure execution is finished).

To support such statements in BDL programs, the database driver uses the SQLExecDirect() ODBC API function when the SQL statement is not a SELECT, INSERT, UPDATE or DELETE. This way the SET statement is executed 'directly', without using the system stored procedures. The result is that the SET statement has the expected effect (i.e. a permanent effect).

However, if the SQL statement uses parameters, the ODBC driver forces the use of system stored procedures to execute the statement.

See the MSDN for more details about system stored procedures used by Microsoft APIs.

ODIMSV046 - The LOAD and UNLOAD instructions

INFORMIX provides two SQL instructions to export / import data from / into a database table: The UNLOAD instruction copies rows from a database table into an text file and the LOAD instruction inserts rows from an text file into a database table.

Warning: Microsoft SQL Server has LOAD and UNLOAD instructions, but those commands are related to database backup and recovery. Do not confuse with INFORMIX commands.

Solution:

LOAD and UNLOAD instructions are supported.

Warning: The LOAD instruction does not work with tables using emulated SERIAL columns because the generated INSERT statement holds the "SERIAL" column which is actually a IDENTITY column in SQL Server. See the limitations of INSERT statements when using SERIALs.

Warning: With Microsoft SQL Server versions prior to 2008, INFORMIX DATE data is stored in DATETIME columns, but DATETIME columns are similar to INFORMIX DATETIME YEAR TO FRACTION(3) columns. Therefore, when using LOAD and UNLOAD, those columns are converted to text data with the format "YYYY-MM-DD hh:mm:ss.fff". However, since SQL Server 2008, INFORMIX DATE data is stored in SQL Server DATE columns, so the result of a LOAD or UNLOAD statement is equivalent when using a DATE column with SQL Server 2008.

Warning: With Microsoft SQL Server versions prior to 2008, INFORMIX DATETIME data is stored in DATETIME columns, but DATETIME columns are similar to INFORMIX DATETIME YEAR TO FRACTION(3) columns. Therefore, when using LOAD and UNLOAD, those columns are converted to text data with the format "YYYY-MM-DD hh:mm:ss.fff". With SQL Server 2008, INFORMIX DATETIME data is stored in SQL Server DATETIME2(n<=5) or TIME(n<=5) columns. Concerning DATETIME2(n<=5) columns, the result of LOAD and UNLOAD is equivalent to INFORMIX DATETIME columns, as long as the original INFORMIX type starts with the YEAR qualifier. The text data will be "YYYY-MM-DD hh:mm:ss.<*fraction-digits*>", where *fraction-digits* depends on the precision (n) of the DATETIME2(n) column. Concerning TIME(n) columns, the type is converted to an INFORMIX DATETIME HOUR TO SECOND or FRACTION(n). The text data will be be "hh:mm:ss.<*fraction-digits*>", where *fraction-digits* depends on the precision (n) of the TIME(n) column.

Warning: When using an INFORMIX database, simple dates are unloaded with the DBDATE format (ex: "23/12/1998"). Therefore, unloading from an INFORMIX database for loading into a Microsoft SQL Server database is not supported.

ODIMSV047 - Case sensitivity

In INFORMIX, database object names like table and column names are not case sensitive :

```
CREATE TABLE Customer ( Custno INTEGER, ... )
SELECT CustNo FROM cuSTomer ...
```

In SQL Server, database object names **and character data** are case-**in**sensitive by default:

```
CREATE TABLE Customer ( Custno INTEGER, CustName
CHAR(20) )
INSERT INTO CUSTOMER VALUES ( 1, 'TECHNOSOFT' )
SELECT CustNo FROM cuSTomer WHERE custname =
'techNOSoft'
```

The installation program of SQL Server allows you to customize the **sort order**. The sort order specifies the rules used by SQL Server to collate, compare, and present character data. **It also specifies whether SQL Server is case-sensitive**.

Solution:

Select the case-sensitive sort order when installing SQL Server.

ODIMSV051 - Setup database statistics

INFORMIX provides a special instruction to compute database statistics in order to help the optimizer find the right query execution plan :

```
UPDATE STATISTICS ...
```

Microsoft SQL Server offers a similar instruction, but it uses different clauses :

```
UPDATE STATISTICS ...
```

See SQL Server documentation for more details.

Solution:

Centralize the optimization instruction in a function.

ODIMSV052 - String concatenation operator

INFORMIX concatenation operator is the double pipe (||) :

```
SELECT firstname || ' ' || lastname FROM employee
```

Microsoft SQL Server concatenation operator is the plus sign :

```
SELECT firstname + ' ' + lastname FROM employee
```

Solution:

The database interface detects double-pipe operators in SQL statements and converts them to a plus sign automatically.

ODIMSV053 - The ALTER TABLE instruction

INFORMIX and MS SQL Server use different implementations of the ALTER TABLE instruction. For example, INFORMIX allows you to use multiple ADD clauses separated by comma. This is not supported by SQL Server :

INFORMIX:

```
ALTER TABLE customer  ADD(col1 INTEGER), ADD(col2 CHAR(20))
```

SQL Server:

```
ALTER TABLE customer  ADD col1 INTEGER, col2 CHAR(20)
```

Solution:

Warning: No automatic conversion is done by the database interface. There is even no real standard for this instruction (that is, no common syntax for all database servers). Read the SQL documentation and review the SQL scripts or the BDL programs in order to use the database server specific syntax for ALTER TABLE.

ODIMSV054 - SQL Interruption

With Informix, it is possible to interrupt a long running query if the SQL INTERRUPT ON option is set by the Genero program. The database server returns SQLCODE -213, which can be trapped to detect a user interruption.

```
MAIN
  DEFINE n INTEGER
  DEFER INTERRUPT
  OPTIONS SQL INTERRUPT ON
```

```
    DATABASE test1
    WHENEVER ERROR CONTINUE
    -- Start long query (self join takes time)
    -- From now on, user can hit CTRL-C in TUI mode to stop the query
    SELECT COUNT(*) INTO n FROM customers a, customers b
         WHERE a.cust_id <> b.cust_id
    IF SQLCA.SQLCODE == -213 THEN
       DISPLAY "Statement was interrupted by user..."
       EXIT PROGRAM 1
    END IF
    WHENEVER ERROR STOP
    ...
END MAIN
```

SQL Server 2005 supports SQL Interruption in a similar way as Informix. The db client must issue an SQLCancel() ODBC call to interrupt a query.

Solution:

The **SNC** database driver supports SQL interruption and converts the SQLSTATE HY008 to the Informix error code -213.

Warning: Make sure you have SQL Server 2005 or higher installed and that you use the **SNC** driver.

ODIMSV100 - Data type conversion table

INFORMIX Data Types	SQL Server Data Types (<2008)	SQL Server Data Types (>=2008)
CHAR(n)	CHAR(n) (limit = 8000c!)	CHAR(n) (limit = 8000c!)
VARCHAR(n)	VARCHAR(n) (limit = 8000c!)	VARCHAR(n) (limit = 8000c!)
INTEGER	INTEGER	INTEGER
SMALLINT	SMALLINT	SMALLINT
FLOAT[(n)]	FLOAT(n)	FLOAT(n)
SMALLFLOAT	REAL	REAL
DECIMAL(p,s)	DECIMAL(p,s)	DECIMAL(p,s)
MONEY(p,s)	DECIMAL(p,s)	DECIMAL(p,s)
DATE	DATETIME	DATE
DATETIME HOUR TO MINUTE	DATETIME	TIME(0)
DATETIME HOUR TO FRACTION(n)	DATETIME	TIME(n)
DATETIME YEAR	DATETIME	DATETIME2(0)

TO SECOND		
Other sort of DATETIME type	DATETIME	DATETIME2(n)
INTERVAL q1 TO q2	CHAR(n)	CHAR(n)
TEXT	VARCHAR(MAX)	VARCHAR(MAX)
BYTE	VARBINARY(MAX)	VARBINARY(MAX)

ODI Adaptation Guide For PostgreSQL 8.0.2, 8.1.x, 8.2.x, 8.3.x

Runtime configuration

- Install PostgreSQL and create a database
- Prepare the runtime environment

Database concepts

- Database concepts
- Data storage concepts
- Data consistency and concurrency management
- Transactions handling
- Defining database users
- Setting privileges

Data dictionary

- CHARACTER data types
- NUMERIC data types
- DATE and DATETIME data types
- INTERVAL data type
- SERIAL data type
- ROWIDs
- Very large data types
- National character data types
- Constraints
- Triggers
- Stored procedures
- Name resolution of SQL objects
- Data type conversion table

Data manipulation

- Reserved words
- Outer joins
- Transactions handling

Temporary tables
Substrings in SQL
The LENGTH( ) function
Name resolution of SQL objects
String delimiters
Using column aliases in SELECT
MATCHES and LIKE conditions
Querying system catalog tables
Syntax of UPDATE statements
The LENGTH() function

BDL programming

SERIAL data type
Handling SQL errors when preparing statements
INFORMIX specific SQL statements in BDL
INSERT cursors
Cursors WITH HOLD
SELECT FOR UPDATE
UPDATE/DELETE WHERE CURRENT OF <cursor>
The LOAD and UNLOAD instructions
SQL Interruption

Runtime configuration

Install PostgreSQL and create a database

1. Compile and install the PostgreSQL Server on your computer. PostgreSQL is a free database, you can download the sources from www.postgresql.org.
2. Set configuration parameters in **postgresql.conf**:

Warning for PGS 8.1 and 8.2: UPDATE / DELETE WHERE CURRENT OF needs **oid** column support. Starting with PostgreSQL version 8.1, user tables do not get the **oid** column by default. You must set the **default_with_oid** configuration parameter to "**on**" in order to get oid columns created. You do no more need to set this parameter with PostgreSQL 8.3 when using the **dbmpgs83x** driver.

3. Start a **postmaster** process to listen to database client connections.

   Warning: If you want to connect through TCP (for example from a Windows PostgreSQL client), you must start postmaster with the **-i** option and setup the **pg_hba.conf** file for security (trusted hosts and users).
4. Create a PostgreSQL database with the **createdb** utility:

   ```
   $ createdb -h hostname dbname
   ```
5. If you plan to use SERIAL emulation, you must create the **plpgsql** procedure language, because the database interface uses this language to create serial triggers:

   ```
   $ createlang -h hostname plpgsql
   dbname
   ```
6. Connect to the database as the administrator user and create a database user dedicated to your application, the **application administrator**:

   ```
   dbname=# CREATE USER appadmin
   PASSWORD 'password';
   CREATE USER
   dbname=# GRANT ALL PRIVILEGES ON
   DATABASE dbname TO appadmin;
   GRANT
   dbname=# \q
   ```
7. Create the **application tables**. Do not forget to convert Informix data types to PostgreSQL data types. See issue ODIPGS100 for more details.
8. If you plan to use the SERIAL emulation, you must prepare the database. See issue ODIPGS005 for more details.

Prepare the runtime environment

1. The **PostgreSQL client software** is required to connect to a database server. Check if the PostgreSQL client library (**libpq.***) is installed on the machine where the 4gl programs run.
2. Set up the **fglprofile** entries for database connections
3. In order to connect to PostgreSQL, you must have a database driver "**dbmpgs***" installed.

   Warning: On HP/UX LP64, the database driver

must be linked with the **libxnet** library if you want to use networking.

ODIPGS001 - DATE and DATETIME data types

INFORMIX provides two data types to store dates and time information:

- **DATE** = for year, month and day storage.
- **DATETIME** = for year to fraction(1-5) storage.

PostgreSQL provides the following data type to store dates:

- **DATE** = for year, month, day storage.
- **TIME [(p)] [{with|without} time zone]** = for hour, minute, second and fraction storage.
- **TIMESTAMP [(p)] [{with|without} time zone]** = for year, month, day, hour, minute, second, fraction storage.

String representing date time information:

INFORMIX is able to convert quoted strings to DATE / DATETIME data if the string contents matches environment parameters (i.e. DBDATE, GL_DATETIME). As in INFORMIX, PostgreSQL can convert quoted strings to date time data according to the DateStyle session parameter. PostgreSQL always accepts ISO date time strings.

Date arithmetic:

- INFORMIX supports date arithmetic on DATE and DATETIME values. The result of an arithmetic expression involving dates/times is a number of days when only DATEs are used and an INTERVAL value if a DATETIME is used in the expression.
- In PostgreSQL, the result of an arithmetic expression involving DATE values is an INTEGER representing a number of days.
- INFORMIX automatically converts an integer to a date when the integer is used to set a value of a date column. PostgreSQL does not support this automatic conversion.
- Complex DATETIME expressions (involving INTERVAL values for example) are INFORMIX specific and have no equivalent in PostgreSQL.

Solution:

PostgreSQL has the same **DATE** data type as INFORMIX (year, month, day). So you can use PostgreSQL DATE data type for Informix DATE columns.

PostgreSQL **TIME(0) WITHOUT TIME ZONE** data type can be used to store INFORMIX DATETIME HOUR TO SECOND values. The database interface makes the conversion automatically.

INFORMIX DATETIME values with any precision from YEAR to FRACTION(5) can be stored in PostgreSQL **TIMESTAMP(5) WITHOUT TIME ZONE** columns. The database interface makes the conversion automatically. Missing date or time parts default to 1900-01-01 00:00:00.0. For example, when using a DATETIME HOUR TO MINUTE with the value of "11:45", the PostgreSQL TIMESTAMP value will be "1900-01-01 11:45:00.0".

Warning: SQL Statements using expressions with TODAY / CURRENT / EXTEND must be reviewed and adapted to the native syntax.

Warning: Literal DATETIME and INTERVAL expressions (i.e. DATETIME (1999-10-12) YEAR TO DAY) are not converted.

ODIPGS003 - Reserved words

SQL object names like table and column names cannot be SQL reserved words in PostgreSQL.

Solution:

Table or column names which are PostgreSQL reserved words must be renamed.

ODIPGS004 - ROWIDs

PostgreSQL tables are automatically created with a OID column (Object Identifier) of type INTEGER. The behavior is equivalent to INFORMIX ROWID columns.

Solution:

The database automatically converts ROWID keywords to OID for PostgreSQL. So you can execute "SELECT ROWID FROM" and "UPDATE .. WHERE ROWID = ?" statements as with INFORMIX.

Warning: SQLCA.SQLERRD[6] is not supported. All references to SQLCA.SQLERRD[6] must be removed because this variable will not hold the ROWID of the last INSERTed or UPDATEd row when using the PostgreSQL interface.

ODIPGS005 - SERIAL data type

INFORMIX SERIAL data type and automatic number production:

- The table column must be of type SERIAL.

- To generate a new serial, **no value** or **zero** value is given to the INSERT statement:
```
INSERT INTO tab1 ( c ) VALUES ( 'aa' )
INSERT INTO tab1 ( k, c ) VALUES ( 0, 'aa' )
```
- After INSERT, the new SERIAL value is provided in **SQLCA.SQLERRD[2]**.
- INFORMIX allows you to insert rows with a value different from zero for a serial column. Using an explicit value will automatically increment the internal serial counter, to avoid conflicts with future INSERT statements that are using a zero value :
```
CREATE TABLE tab ( k SERIAL ); --> internal counter = 0
INSERT INTO tab VALUES ( 0 );   --> internal counter =
1
INSERT INTO tab VALUES ( 10 );  --> internal counter =
10
INSERT INTO tab VALUES ( 0 );   --> internal counter =
11
DELETE FROM tab;                --> internal counter =
11
INSERT INTO tab VALUES ( 0 );   --> internal counter =
12
```

PostgreSQL SERIAL data type:

- This data type has the same name but is NOT the same as INFORMIX SERIALs.
- You cannot define a start value (SERIAL(100)).
- You must omit the column name when inserting rows because the technique is based on default values.
- You can insert a specific value but the next serial will be generated from the sequence.
- When you drop the table you must drop the sequence too.

PostgreSQL sequences:

- Sequences are totally detached from tables.
- The purpose of sequences is to provide unique integer numbers.
- Sequences are identified by a sequence name.
- To create a sequence, you must use the CREATE SEQUENCE statement. Once a sequence is created, it is permanent (like a table).
- To get a new sequence value, you must use the **nextval** function:
```
INSERT INTO tab1 VALUES ( nextval('tab1_seq'), ...
)
```
- To get the last generated number, PostgreSQL provides the **currval** function :
```
SELECT currval('tab1_seq') FROM DUAL
```

Solution:

The INFORMIX SERIAL data type can be emulated with three methods, according to FGLPROFILE settings:

1. When dbi.database.<dbname>.ifxemul.datatype.serial.emulation = "native" (the default), the native PostgreSQL SERIAL data type is used. The **sqlca.sqlerrd[2]** register is **NOT** filled as expected.
2. When dbi.database.<dbname>.ifxemul.datatype.serial.emulation = "regtable", the SERIAL data type is emulated with a PostgreSQL **INTEGER** data type and **INSERT triggers** using the table **SERIALREG** which is dedicated to sequence production. After an insert, **sqlca.sqlerrd[2]** register holds the last generated serial value.
3. When dbi.database.<dbname>.ifxemul.datatype.serial.emulation = "trigseq", the SERIAL data type is emulated with a PostgreSQL **INTEGER** data type and **INSERT triggers** using a sequence **<tablename>_seq**. After an insert, **sqlca.sqlerrd[2]** register holds the last generated serial value.

Warning: When using solution 1 (which is the default), the **sqlca.sqlerrd[2]** register is not set after an INSERT. You must configure FGLPROFILE to use solution 2 or 3 when you need this feature.

When using solution 2 or 3, the triggers can be created **manually** during the application database installation procedure, or **automatically** from a BDL program: When a BDL program executes a CREATE [TEMP] TABLE with a SERIAL column, the database interface automatically converts the SERIAL data type to INTEGER and dynamically creates the triggers. However, when using solution 2, the table SERIALREG must be created before the triggers. The serial production is based on the SERIALREG table which registers the last generated number for each table. If you delete rows of this table, sequences will restart at 1 and you will get unexpected data.

If you plan to use the second method, you must create the SERIALREG table as follows:

```
CREATE TABLE SERIALREG (
     TABLENAME VARCHAR(50) NOT NULL,
     LASTSERIAL INTEGER NOT NULL,
     PRIMARY KEY ( TABLENAME )
)
```

Warning: This table must exist in the database before creating the serial triggers.

In database creation scripts, all SERIAL[(n)] data types must be converted to INTEGER data types and you must create one trigger for each table. To know how to write those triggers, you can create a small Genero program that creates a table with a SERIAL column. Set the FGLSQLDEBUG environment variable and run the program. The debug output will show you the native trigger creation command.

Warning: With PostgreSQL, INSERT statements using NULL for the SERIAL column will produce a new serial value, not a NULL like INFORMIX does:

```
INSERT INTO tab (col1,col2) VALUES (NULL,'data')
```

This behavior is mandatory in order to support INSERT statements which do not use the serial column:

```
INSERT INTO tab (col2) VALUES ('data')
```

Check if your application uses tables with a SERIAL column that can contain a NULL value.

ODIPGS006 - Outer joins

In INFORMIX SQL, outer tables are defined in the FROM clause with the **OUTER** keyword:

```
SELECT ... FROM a, OUTER(b)
 WHERE a.key = b.akey
SELECT ... FROM a, OUTER(b,OUTER(c))
 WHERE a.key = b.akey
   AND b.key1 = c.bkey1
   AND b.key2 = c.bkey2
```

PostgreSQL supports the ANSI outer join syntax:

```
SELECT ... FROM cust LEFT OUTER JOIN order
                    ON cust.key = order.custno
SELECT ...
  FROM cust LEFT OUTER JOIN order
                 LEFT OUTER JOIN item
                 ON order.key = item.ordno
            ON cust.key = order.custno
 WHERE order.cdate > current date
```

See the PostgreSQL reference for a complete description of the syntax.

Solution:

The PostgreSQL interface can convert most INFORMIX OUTER specifications to ANSI outer joins.

Prerequisites:

1. In the FROM clause, the main table must be the first item and the outer tables must figure from left to right in the order of outer levels.
   Example which does not work : "`FROM OUTER(tab2), tab1`".
2. The outer join in the WHERE part must use the table name as prefix.
   Example : "`WHERE tab1.col1 = tab2.col2`".

Restrictions:

1. Additional conditions on outer table columns cannot be detected and therefore are not supported:
   Example : "... `FROM tab1, OUTER(tab2) WHERE tab1.col1 = tab2.col2 AND tab2.colx > 10`".
2. Statements composed of 2 or more SELECT instructions using OUTERs are not supported.
   Example : "`SELECT ... UNION SELECT`" or "`SELECT ... WHERE col IN (SELECT...)`"

Remarks:

1. Table aliases are detected in OUTER expressions.
   OUTER example with table alias : "`OUTER( tab1 alias1)`".
2. In the outer join, <outer table>.<col> can be placed on both right or left sides of the equal sign.
   OUTER join example with table on the left : "`WHERE outertab.col1 = maintab.col2` ".
3. Table names detection is not case-sensitive.
   Example : "`SELECT ... FROM tab1, TAB2 WHERE tab1.col1 = tab2.col2`".
4. Temporary tables are supported in OUTER specifications.

ODIPGS007a - Database concepts

Most BDL applications use only one database entity (in the meaning of INFORMIX). But the same BDL application can connect to different occurrences of the same database schema, allowing several users to connect to those different databases.

Like INFORMIX servers, PostgreSQL can handle multiple database entities. Tables created by a user can be accessed without the owner prefix by other users as long as they have access privileges to these tables.

Solution:

Create a PostgreSQL database for each INFORMIX database.

ODIPGS008a - Data consistency and concurrency management

Data consistency involves readers which want to access data currently modified by writers and **concurrency data access** involves several writers accessing the same data for modification. **Locking granularity** defines the amount of data concerned when a lock is set (row, page, table, ...).

INFORMIX

INFORMIX uses a locking mechanism to handle data consistency and concurrency. When a process changes database information with UPDATE, INSERT or DELETE, an **exclusive lock** is set on the touched rows. The lock remains active until the end of the transaction. Statements performed outside a transaction are treated as a transaction containing a single operation and therefore release the locks immediately after execution. SELECT statements can set **shared locks** according to the **isolation level**. In case of locking conflicts (for example, when two processes want to acquire an

exclusive lock on the same row for modification or when a writer is trying to modify data protected by a shared lock), the behavior of a process can be changed by setting the **lock wait mode**.

Control:

- Lock wait mode : SET LOCK MODE TO ...
- Isolation level : SET ISOLATION TO ...
- Locking granularity : CREATE TABLE ... LOCK MODE {PAGE|ROW}
- Explicit exclusive lock : SELECT ... FOR UPDATE

Defaults:

- The default isolation level is read committed.
- The default lock wait mode is "not wait".
- The default locking granularity is per page.

PostgreSQL

When data is modified, **exclusive locks** are set and held until the end of the transaction. For data consistency, PostgreSQL uses a **multi-version consistency model**: A copy of the original row is kept for readers before performing writer modifications. Readers do not have to wait for writers as in INFORMIX. The simplest way to think of PostgreSQL implementation of read consistency is to imagine each user operating a private copy of the database, hence the multi-version consistency model. The **lock wait** mode cannot be changed as in INFORMIX. Locks are set at the **row level** in PostgreSQL and this cannot be changed.

Control:

- No lock wait mode control is provided.
- Isolation level : SET TRANSACTION ISOLATION LEVEL ...
- Explicit exclusive lock : SELECT ... FOR UPDATE

Defaults:

- The default isolation level is Read Committed.

The main difference between INFORMIX and PostgreSQL is that readers do not have to wait for writers in PostgreSQL.

Solution:

For portability, it is recommended that you work with INFORMIX in the read committed isolation level, make processes wait for each other (lock mode wait), and create tables with the "lock mode row" option.

See INFORMIX and PostgreSQL documentation for more details about data consistency, concurrency and locking mechanisms.

ODIPGS008b - SELECT FOR UPDATE

A lot of BDL programs use pessimistic locking in order to avoid several users editing the same rows at the same time.

```
DECLARE cc CURSOR FOR
      SELECT ... FROM tab WHERE ... FOR UPDATE
OPEN cc
FETCH cc <-- lock is acquired
...
CLOSE cc <-- lock is released
```

In both INFORMIX and PostgreSQL, locks are released when closing the cursor or when the transaction ends.

PostgreSQL locking granularity is at the row level.

To control the behavior of the program when locking rows, INFORMIX provides a specific instruction to set the wait mode:

```
SET LOCK MODE TO { WAIT | NOT WAIT | WAIT seconds }
```

The default mode is NOT WAIT. This as an INFORMIX specific SQL statement.

Warning: PostgreSQL has no equivalent for "SET LOCK MODE TO NOT WAIT".

Solution:

Warning: The database interface is based on an emulation of an Informix engine using transaction logging. Therefore, opening a SELECT ... FOR UPDATE cursor declared outside a transaction will raise an SQL error -255 (not in transaction).

You must review the program logic if you use pessimistic locking because it is based on the NOT WAIT mode which is not supported by PostgreSQL.

ODIPGS009 - Transactions handling

INFORMIX and PostgreSQL handle transactions in a similar manner.

INFORMIX native mode (non ANSI):

- Transactions are started with "BEGIN WORK".
- Transactions are validated with "COMMIT WORK".
- Transactions are canceled with "ROLLBACK WORK".

- Statements executed outside of a transaction are automatically committed.
- DDL statements can be executed (and canceled) in transactions.

PostgreSQL:

- Transactions are started with "BEGIN WORK".
- Transactions are validated with "COMMIT WORK".
- Transactions are canceled with "ROLLBACK WORK".
- Statements executed outside of a transaction are automatically committed.
- DDL statements can be executed (and canceled) in transactions.
- If an SQL error occurs in a transaction, the whole transaction is aborted.

Transactions in stored procedures: avoid using transactions in stored procedures to allow the client applications to handle transactions, according to the transaction model.

Warning: The main difference between Informix and PostgreSQL resides in the fact that PostgreSQL cancels the whole transaction if an SQL error occurs in one of the statements executed inside the transaction. The following code example illustrates this difference:

```
   CREATE TABLE tab1 ( k INT PRIMARY KEY, c CHAR(10) )
   WHENEVER ERROR CONTINUE
   BEGIN WORK
   INSERT INTO tab1 ( 1, 'abc' )
   INSERT INTO tab1 ( 1, 'abc' ) -- PK constraint violation = SQL
Error
   COMMIT WORK
```

With Informix, this code will leave the table with one row inside, since the first INSERT statement succeeded. With PostgreSQL, the table will remain empty after executing this piece of code, because the server will rollback the whole transaction.

Solution:

INFORMIX transaction handling commands are automatically converted to PostgreSQL instructions to start, validate or cancel transactions.

Regarding the transaction control instructions, the BDL applications do not have to be modified in order to work with PostgreSQL.

Warning: You must review the SQL statements inside BEGIN WORK / COMMIT WORK instruction and check if these can raise and SQL error. With PostgreSQL, the whole transaction will be aborted. The code example shown above could for example be converted to this:

```
   CREATE TABLE tab1 ( k INT PRIMARY KEY, c CHAR(10) )
   WHENEVER ERROR CONTINUE
   BEGIN WORK
   INSERT INTO tab1 ( 1, 'abc' )
   IF SQLCA.SQLCODE < 0 THEN ROLLBACK WORK GOTO _END_ END IF
```

```
   INSERT INTO tab1 ( 1, 'abc' ) -- PK constraint violation = SQL
Error
   IF SQLCA.SQLCODE < 0 THEN ROLLBACK WORK GOTO _END_ END IF
   COMMIT WORK
LABEL _END_:
   ...
```

ODIPGS010 - Handling SQL errors when preparing statements

The PostgreSQL connector is implemented with the PostgreSQL **libpq** API. This library does not provide a way to send SQL statements to the database server during the BDL PREPARE instruction, like the INFORMIX interface. The statement is sent to the server only when opening the cursors or when executing the statement.

Therefore, when preparing an SQL statement with the BDL PREPARE instruction, no SQL errors can be returned if the statement has syntax errors or if a column or a table name does not exist in the database.

However, an SQL error will occur after the OPEN or EXECUTE instructions.

Solution:

Check that your BDL programs do not test STATUS or SQLCA.SQLCODE variable just after PREPARE instructions.

Change the program logic in order to handle the SQL errors when opening the cursors (OPEN) or when executing SQL statements (EXECUTE).

ODIPGS011a - CHARACTER data types

As in INFORMIX, PostgreSQL provides the CHAR and VARCHAR data types to store character data.

INFORMIX CHAR type can store up to **32767** characters and the VARCHAR data type is limited to **255** characters.

Since PostgreSQL CHAR and VARCHAR have a size limit of **1GB**.

Solution:

The database interface supports character string variables in SQL statements for input (BDL USING) and output (BDL INTO).

ODIPGS011b - The LENGTH( ) function

Warning: PostgreSQL raises an error if the LENGTH() parameter is NULL. INFORMIX returns zero instead.

Solution:

The PostgreSQL database interface cannot simulate the behavior of the INFORMIX LENGTH() SQL function.

Review the program logic and make sure you do not pass NULL values to the LENGTH() SQL function.

ODIPGS012 - Constraints

Constraint naming syntax:

Both INFORMIX and PostgreSQL support primary key, unique, foreign key, default and check constraints, but the constraint naming syntax is different. PostgreSQL expects the "CONSTRAINT" keyword **before** the constraint specification and INFORMIX expects it **after**.

UNIQUE constraint example:

INFORMIX	PostgreSQL

`CREATE TABLE scott.emp (`
`...`
`empcode CHAR(10) UNIQUE`
`  [CONSTRAINT pk_emp],`
`...`
	`CREATE TABLE scott.emp (`
	`...`
	`empcode CHAR(10)`
	`  [CONSTRAINT pk_emp]`
	`UNIQUE,`
	`...`

Unique constraints:

Warning: When using a unique constraint, INFORMIX allows only one row with a NULL value, while PostgreSQL allows several rows with NULL!

Solution:

The database interface does not convert constraint naming expressions when creating tables from BDL programs. Review the database creation scripts to adapt the constraint naming clauses for PostgreSQL.

ODIPGS013 - Triggers

INFORMIX and PostgreSQL provide triggers with similar features, but the trigger creation syntax and the programming languages are totally different.

Solution:

INFORMIX triggers must be converted to PostgreSQL triggers "by hand".

ODIPGS014 - Stored procedures

Both INFORMIX and PostgreSQL support stored procedures, but the programming languages are totally different. With PostgreSQL you must create the stored procedure language before writing triggers or stored procedures.

Solution:

INFORMIX stored procedures must be converted to PostgreSQL manually.

ODIPGS016a - Defining database users

INFORMIX users are defined at the operating system level, they must be members of the 'informix' group, and the database administrator must grant CONNECT, RESOURCE or DBA privileges to those users.

PostgreSQL users must be registered in the database. They are created by the **createuser** utility:

```
$ createuser --username=<username> --password
```

Solution:

According to the application logic (is it a multi-user application ?), you have to create one or several PostgreSQL users.

ODIPGS016b - Setting privileges

INFORMIX and PostgreSQL user privileges management are quite similar.

PostgreSQL provides **user groups** to grant or revoke permissions to more than one user at the same time.

ODIPGS017 - Temporary tables

INFORMIX temporary tables are created through the **`CREATE TEMP TABLE`** DDL instruction or through a **`SELECT ... INTO TEMP`** statement. Temporary tables are automatically dropped when the SQL session ends, but they can be dropped with the DROP TABLE command. There is no name conflict when several users create temporary tables with the same name.

INFORMIX allows you to create indexes on temporary tables. No name conflict occurs when several users create an index on a temporary table by using the same index identifier.

PostgreSQL support temporary tables as INFORMIX does, with a little syntax difference in the SELECT INTO TEMP instruction.

Solution:

Temporary tables are well supported with native PostgreSQL temp tables.

ODIPGS018 - Substrings in SQL

INFORMIX SQL statements can use subscripts on columns defined with the character data type:

```
    SELECT ... FROM tab1 WHERE col1[2,3] = 'RO'
    SELECT ... FROM tab1 WHERE col1[10]  = 'R'   -- Same as
col1[10,10]
    UPDATE tab1 SET col1[2,3]= 'RO' WHERE ...
    SELECT ... FROM tab1 ORDER BY col1[1,3]
```

.. while PostgreSQL provides the SUBSTR( ) function, to extract a substring from a string expression:

```
    SELECT .... FROM tab1 WHERE SUBSTRING(col1 from 2 for 2) =
'RO'
    SELECT SUBSTRING('Some text' from 6 for 3) ...   -- Gives
'tex'
```

Solution:

You must replace all Informix col[x,y] expressions by SUBSTRING(col from x for (y-x+1)).

Warning: In UPDATE instructions, setting column values through subscripts will produce an error with PostgreSQL :

```
    UPDATE tab1 SET col1[2,3] = 'RO' WHERE ...
```

is converted to:

```
    UPDATE tab1 SET SUBSTRING(col1 from 2 for (3-2+1)) = 'RO'
WHERE ...
```

Warning: Column subscripts in ORDER BY expressions are also converted and produce an error with PostgreSQL:

```
    SELECT ... FROM tab1 ORDER BY col1[1,3]
```

is converted to:

```
    SELECT ... FROM tab1 ORDER BY SUBSTRING(col1 from 1 for(3-
1+1))
```

ODIPGS019 - Name resolution of SQL objects

INFORMIX uses the following form to identify an SQL object :

```
  [database[@dbservername]:][{owner|"owner"}.]identifier
```

With PostgreSQL, an object name takes the following form:

```
  [owner.]identifier
```

Solution:

Check for single or double quoted table or column names in your source and remove them.

ODIPGS020 - String delimiters

The ANSI string delimiter character is the single quote ('string'). Double quotes are used to delimit database object names ("object-name").

Example: `WHERE "tabname"."colname" = 'a string value'`

INFORMIX allows double quotes as string delimiters, but PostgreSQL doesn't. This is important since many BDL programs use that character to delimit the strings in SQL commands.

Note: This problem concerns only double quotes within SQL statements. Double quotes used in pure BDL string expressions are not subject to SQL compatibility problems.

Solution:

The PostgreSQL database interface can automatically replace all double quotes by single quotes.

Escaped string delimiters can be used inside strings like following:

```
'This is a single quote: '''
'This is a single quote : \''
"This is a double quote : """
"This is a double quote : \""
```

Warning: Database object names cannot be delimited by double quotes because the database interface cannot determine the difference between a database object name and a quoted string!

For example, if the program executes the SQL statement:
`WHERE "tabname"."colname" = "a string value"`
replacing all double quotes by single quotes would produce:
`WHERE 'tabname'.'colname' = 'a string value'`
This would produce an error since `'tabname'.'colname'` is not allowed by PostgreSQL.

Although double quotes are replaced automatically in SQL statements, you should use only single quotes to enforce portability.

ODIPGS021 - NUMERIC data types

INFORMIX supports several data types to store numbers:

INFORMIX Data Type	Description
SMALLINT	16 bit integer (-2^15 to 2^15)
INT/INTEGER	32 bit integer (-2^31 to 2^31)
DEC/DECIMAL(p)	Floating-point decimal number
DEC/DECIMAL(p,s)	Fixed-point decimal number
MONEY	Equivalent to DECIMAL(16,2)
MONEY(p)	Equivalent to DECIMAL(p,2)
MONEY(p,s)	Equivalent to DECIMAL(p,s)
REAL/SMALLFLOAT	approx floating point (C float)
DOUBLE PREC./FLOAT	approx floating point (C double)

Solution:

PostgreSQL supports the following data types to store numbers:

PostgreSQL data type	Description
NUMERIC(p,s)	Decimals (no limit)
DECIMAL(p,s)	Decimals (8000 digits)
FLOAT4	4 bytes variable precision
FLOAT8	8 bytes variable precision
INT2	2 bytes integer
INT4	4 bytes integer
INT8	8 bytes integer

ANSI types like SMALLINT, INTEGER, FLOAT are supported by PostgreSQL as aliases to INT2, INT4 and FLOAT8 native types.

ODIPGS022 - Using column aliases in SELECT

PostgreSQL expects the ANSI notation for column aliases :

```
SELECT col1 AS col1_alias FROM ...
```

INFORMIX supports the ANSI notation.

Solution:

Warning: The database interface cannot convert INFORMIX alias specification to the ANSI notation.

Review your programs and replace the INFORMIX notation with the ANSI form.

ODIPGS024 - MATCHES and LIKE in SQL conditions

INFORMIX supports MATCHES and LIKE in SQL statements. PostgreSQL supports the LIKE statement as in INFORMIX, plus the ~ operators that are similar but different from the INFORMIX MATCHES operator.

MATCHES allows brackets to specify a set of matching characters at a given position :

```
( col MATCHES '[Pp]aris' ).
( col MATCHES '[0-9][a-z]*' ).
```

In this case, the LIKE statement has not equivalent feature.

The following substitutions must be made to convert a MATCHES condition to a LIKE condition :

- MATCHES keyword must be replaced by LIKE.
- All '*' characters must be replaced by '%'.
- All '?' characters must be replaced by '_'.
- Remove all brackets expressions.

PostgreSQL ~ operator expects regular expressions as follows:

```
( col ~ 'a.*' )
```

Solution:

Warning: SQL statements using MATCHES expressions must be reviewed in order to use LIKE expressions.

See also: MATCHES operator in SQL Programming.

ODIPGS025 - INFORMIX specific SQL statements in BDL

The BDL compiler supports several INFORMIX specific SQL statements that have no meaning when using PostgreSQL.

- CREATE DATABASE
- DROP DATABASE
- START DATABASE (SE only)
- ROLLFORWARD DATABASE
- SET [BUFFERED] LOG
- CREATE TABLE with special options (storage, lock mode, etc.)

Solution:

Review your BDL source and remove all static SQL statements which are INFORMIX specific.

ODIPGS028 - INSERT cursors

INFORMIX supports insert cursors. An "insert cursor" is a special BDL cursor declared with an INSERT statement instead of a SELECT statement. When this kind of cursor is open, you can use the PUT instruction to add rows and the FLUSH instruction to insert the records into the database.

For INFORMIX database with transactions, OPEN, PUT and FLUSH instructions must be executed within a transaction.

PostgreSQL does not support insert cursors.

Solution:

Insert cursors are emulated by the PostgreSQL database interface.

ODIPGS030 - Very large data types

Both INFORMIX and PostgreSQL Server provide special data types to store very large texts or images, but the names are different:

INFORMIX Data Type	PostgreSQL Data Type
TEXT	TEXT
BYTE	BYTEA

Solution:

Very large data types are not supported yet by the PostgreSQL database interface.

ODIPGS031 - Cursors WITH HOLD

INFORMIX closes opened cursors automatically when a transaction ends unless the WITH HOLD option is used in the DECLARE instruction. In PostgreSQL, opened cursors using SELECT statements without a FOR UPDATE clause are not closed when a transaction ends. Actually, all PostgreSQL cursors are 'WITH HOLD' cursors unless the FOR UPDATE clause is used in the SELECT statement.

Warning: Cursors declared FOR UPDATE and using the WITH HOLD option cannot be supported with PostgreSQL because FOR UPDATE cursors are automatically closed by PostgreSQL when the transaction ends.

Solution:

BDL cursors that are not declared "WITH HOLD" are automatically closed by the database interface when a COMMIT WORK or ROLLBACK WORK is performed.

Warning: Since PostgreSQL automatically closes FOR UPDATE cursors when the transaction ends, opening cursors declared FOR UPDATE and WITH HOLD option results in an SQL error that does not normally appear with INFORMIX, in the same conditions. Review the program logic in order to find another way to set locks.

ODIPGS032 - UPDATE/DELETE WHERE CURRENT OF <cursor>

INFORMIX allows positioned UPDATEs and DELETEs with the "WHERE CURRENT OF <cursor>" clause, if the cursor has been DECLARED with a SELECT ... FOR UPDATE statement.

PGS 8.1 and 8.2: UPDATE/DELETE ... WHERE CURRENT OF <cursor> is not supported by PostgreSQL. However, you can use the OID column to do positioned updates/deletes.

Since PGS 8.3: UPDATE/DELETE ... WHERE CURRENT OF <cursor> is supported by PostgreSQL with server-side cursors created with a DECLARE statement.

Solution:

With PostgreSQL 8.1 (dbmpgs81x) and 8.2 (dbmpgs82x):

UPDATE/DELETE ... WHERE CURRENT OF instructions are managed by the PostgreSQL database interface. The PostgreSQL database interface replaces "WHERE CURRENT OF <cursor>" by "WHERE OID = ?" and sets the value of the Object Identifier returned by the last FETCH done with the given cursor..

Warning PGS 8.1 and 8.2: Starting with PostgreSQL version 8.1, user tables do not get the **oid** column by default. You must set the **default_with_oid** configuration parameter in the **postgresql.conf** file.

With PostgreSQL 8.3 (dbmpgs83x) and higher:

UPDATE/DELETE ... WHERE CURRENT OF instructions are just executed as is. Since SELECT FOR UPDATE statements are now executed with a server cursor by using a DECLARE PostgreSQL statement, native positioned update/delete takes place.

ODIPGS033 - Querying system catalog tables

As in INFORMIX, PostgreSQL provides system catalog tables (actually, system views). But the table names and their structure are quite different.

Solution:

Warning: No automatic conversion of INFORMIX system tables is provided by the database interface.

ODIPGS034 - Syntax of UPDATE statements

INFORMIX allows a specific syntax for UPDATE statements:

```
UPDATE table SET ( <col-list> ) = ( <val-list> )
```

or

```
UPDATE table SET table.* = myrecord.*
UPDATE table SET * = myrecord.*
```

Solution:

Static UPDATE statements using the above syntax are converted **by the compiler** to the standard form:

```
UPDATE table SET column=value [,...]
```

ODIPGS035 - The LENGTH() function

INFORMIX provides the LENGTH() function:

```
SELECT LENGTH("aaa"), LENGTH(col1) FROM table
```

PostgreSQL has a equivalent function with the same name, but there is some difference:

Informix does not count the trailing blanks neither for CHAR not for VARCHAR expressions, while PostgreSQL counts the trailing blanks.

With the PostgreSQL LENGTH function, when using a CHAR column, values are always blank padded, so the function returns the size of the CHAR column. When using a VARCHAR column, trailing blanks are significant, and the function returns the number of characters, including trailing blanks.

Solution:

You must check if the trailing blanks are significant when using the LENGTH() function.

If you want to count the number of character by ignoring the trailing blanks, you must use the RTRIM() function:

ODIPGS036 - INTERVAL data type

INFORMIX INTERVAL data type stores a value that represents a span of time. INTERVAL types are divided into two classes : ***year-month intervals*** and ***day-time intervals***.

PostgreSQL provides an INTERVAL data type, but it is totally different from the INFORMIX INTERVAL type. For example, you specify a INTERVAL literal as follows :

```
25 years 2 months 23 days
```

Solution:

Warning: The INTERVAL data type is not well supported because the database server has no equivalent native data type. However, you can store into and retrieve from CHAR columns BDL INTERVAL values.

ODIPGS039 - Data storage concepts

An attempt should be made to preserve as much of the storage information as possible when converting from INFORMIX to PostgreSQL. Most important storage decisions made for INFORMIX database objects (like initial sizes and physical placement) can be reused for the PostgreSQL database.

Storage concepts are quite similar in INFORMIX and in PostgreSQL, but the names are different.

ODIPGS040 - National characters data types

INFORMIX: NCHAR & NVARCHAR
PostgreSQL:

Solution:

Warning: National character data types are not supported yet.

ODIPGS046 - The LOAD and UNLOAD instructions

INFORMIX provides two SQL instructions to export / import data from / into a database table: The UNLOAD instruction copies rows from a database table into a text file and the LOAD instructions insert rows from a text file into a database table.

PostgreSQL does not provide LOAD and UNLOAD instructions, but provides external tools like SQL*Plus and SQL*Loader.

Solution:

LOAD and UNLOAD instructions are supported.

Warning: There is a difference when using PostgreSQL DATE columns: DATE columns created in the PostgreSQL database are similar to INFORMIX DATETIME YEAR TO SECOND columns. In LOAD and UNLOAD, all PostgreSQL DATE columns are treated as INFORMIX DATETIME YEAR TO SECOND columns and thus will be unloaded with the "YYYY-MM-DD hh:mm:ss" format.
The same problem appears for INFORMIX INTEGER and SMALLINT values which are stored in a PostgreSQL database as NUMBER(?) columns. Those values will be unloaded as INFORMIX DECIMAL(10) and DECIMAL(5) values, that is, with a trailing dot-zero ".0".

Warning: When using an INFORMIX database, simple dates are unloaded with the DBDATE format (ex: "23/12/1998"). Therefore, unloading from an INFORMIX database for loading into an PostgreSQL database is not supported.

Warning: UNLOAD instructions based on SELECT statements using date expressions as `"WHERE ?-datecol>?"` are not supported because of database server limitations. A syntax error would be raised in this case.

ODIPGS047 - SQL Interruption

With Informix, it is possible to interrupt a long running query if the SQL INTERRUPT ON option is set by the Genero program. The database server returns SQLCODE -213, which can be trapped to detect a user interruption.

```
MAIN
  DEFINE n INTEGER
  DEFER INTERRUPT
  OPTIONS SQL INTERRUPT ON
  DATABASE test1
  WHENEVER ERROR CONTINUE
  -- Start long query (self join takes time)
  -- From now on, user can hit CTRL-C in TUI mode to stop the query
  SELECT COUNT(*) INTO n FROM customers a, customers b
       WHERE a.cust_id <> b.cust_id
  IF SQLCA.SQLCODE == -213 THEN
     DISPLAY "Statement was interrupted by user..."
     EXIT PROGRAM 1
  END IF
  WHENEVER ERROR STOP
  ...
END MAIN
```

PostgreSQL supports SQL Interruption in a similar way as Informix. The db client must issue an PQcancel() libPQ call to interrupt a query.

Solution:

The PostgreSQL database driver supports SQL interruption and converts the SQLSTATE code 57014 to the Informix error code -213.

ODIPGS100 - Data type conversion table

INFORMIX Data Types	PostgreSQL Data Types
CHAR(n)	CHAR(n)
VARCHAR(n)	VARCHAR(n)
INTEGER	INT4
SMALLINT	INT2
FLOAT[(n)]	FLOAT4
SMALLFLOAT	FLOAT8
DECIMAL(p,s)	NUMERIC(p,s)
MONEY(p,s)	NUMERIC(p,s)
DATE	DATE
DATETIME HOUR TO SECOND	TIME(0) WITHOUT TIME ZONE
DATETIME YEAR TO FRACTION(p)	TIMESTAMP(p) WITHOUT TIME ZONE
INTERVAL q1 TO q2	N/A

ODI Adaptation Guide For MySQL 4.1.x, 5.0.x, 5.1.x

Runtime configuration

- Install MySQL and create a database
- Prepare the runtime environment

Database concepts

- Database concepts
- Data storage concepts
- Data consistency and concurrency management
- Transactions handling
- Defining database users

Data dictionary

- CHARACTER data types
- NUMERIC data types
- DATE and DATETIME data types
- INTERVAL data type
- SERIAL data type
- ROWIDs
- Constraints
- Name resolution of SQL objects
- Data type conversion table

Data manipulation

- Reserved words
- Outer joins
- Transactions handling
- Temporary tables
- Substrings in SQL
- Name resolution of SQL objects
- Database object name delimiters
- MATCHES and LIKE conditions

Syntax of UPDATE statements

BDL programming

SERIAL data type
Handling SQL errors when preparing statements
INFORMIX specific SQL statements in BDL
INSERT cursors
Cursors WITH HOLD
SELECT FOR UPDATE
UPDATE/DELETE WHERE CURRENT OF <cursor>
The LOAD and UNLOAD instructions
SQL Interruption

Runtime configuration

Install MySQL and create a database

1. **Warning:** Supported MySQL versions are **4.1.2** and higher.
2. Compile and/or install the MySQL Server on your computer. MySQL is a free database; you can download the sources or binary packages from www.mysql.com. For more details about MySQL installation and configuration, read the documentation.
3. Configure the database server.
   Warning: In order to have transaction support by default, you must define **INNODB** as default storage engine:
   In the **my.cnf** or **my.ini** file, you must have these lines:
   ```
   [mysqld]
   default-storage-engine = INNODB
   ```
   You can also set the default table type option in the command line when starting the engine:
   ```
   mysqld_safe --
   default_table_type=InnoDB
   ```
4. The **mysqld** process must be started to listen to database client connections. See MySQL documentation for more details about starting the database server process.

5. Create a database user dedicated to your application, the **application administrator.** Connect as the MySQL root user and GRANT all privileges to this user:
```
mysql -u root
...
mysql> grant all privileges on *.*
              to
'mysuser'@'localhost'
              identified by
'password';
```
6. Connect as the application administrator and create a MySQL database with the **CREATE DATABASE** statement:
```
mysql -u mysuser
...
mysql> create database mydatabase;
```
7. Create the **application tables**. Do not forget to convert Informix data types to MySQL data types. See issue ODIMYS100 for more details. If you have a transactional-safe table handler activated by default, you do not need to specify the TYPE option.

Prepare the runtime environment

1. The **MySQL client software** is required to connect to a database server. Check if the MySQL client library (**libmysqlclient.***) is installed on the system.

   Warning: MySQL ships the client library as a simple .a archive, there is no shared library provided by default. You must create a .so or .DLL library in order to use one of the MySQL ODI drivers of Genero.

   For example, to create a **libmysqlclient.so** shared library on Linux, execute the following commands:

```
$ cd $MYSQLDIR/lib
$ mkdir tmp
$ cd tmp
$ ar -x ../libmysqlclient.a
$ gcc --shared -o ../libmysqlclient.so
*.o -lz
$ cd ..
% rm -rf tmp
```
2. Set up the **fglprofile** entries for database connections.
3. In order to connect to MySQL, you must have a database driver "**dbmmys***" installed.

ODIMYS001 - DATE and DATETIME data types

INFORMIX provides two data types to store dates and time information:

- **DATE** = for year, month and day storage.
- **DATETIME** = for year to fraction(1-5) storage.

MySQL provides the following data type to store dates:

- **DATE** = for year, month, day storage.
- **TIME** = for hour, minute, second storage.
- **DATETIME** = for year, month, day, hour, minute, second storage.
- **TIMESTAMP** = automatically updated when row is touched.

String representing date time information:

INFORMIX is able to convert quoted strings to DATE / DATETIME data if the string contents matches environment parameters (i.e. DBDATE, GL_DATETIME). As in INFORMIX, MySQL can convert quoted strings to datetime data according the ISO datetime format (YYYY-MM-DD hh:mm:ss').

Date arithmetic:

- INFORMIX supports date arithmetic on DATE and DATETIME values. The result of an arithmetic expression involving dates/times is a number of days when only DATEs are used and an INTERVAL value if a DATETIME is used in the expression.
- In MySQL, the result of an arithmetic expression involving DATE values is an INTEGER representing a number of days.
- INFORMIX automatically converts an integer to a date when the integer is used to set a value of a date column. MySQL does not support this automatic conversion.
- Complex DATETIME expressions (involving INTERVAL values for example) are INFORMIX specific and have no equivalent in MySQL.

Solution:

MySQL has the same **DATE** data type as INFORMIX (year, month, day). So you can use MySQL DATE data type for Informix DATE columns.

MySQL **TIME** data type can be used to store INFORMIX DATETIME HOUR TO SECOND values. The database interface makes the conversion automatically.

INFORMIX DATETIME values with any precision from YEAR to SECOND can be stored in MySQL **DATETIME** columns. The database interface makes the conversion automatically. Missing date or time parts default to 1900-01-01 00:00:00. For example,

when using a DATETIME HOUR TO MINUTE with the value of "11:45", the MySQL DATETIME value will be "1900-01-01 11:45:00".

Warning: SQL Statements using expressions with TODAY / CURRENT / EXTEND must be reviewed and adapted to the native syntax.

Warning: Literal DATETIME and INTERVAL expressions (i.e. DATETIME (1999-10-12) YEAR TO DAY) are not converted.

ODIMYS003 - Reserved words

SQL object names like table and column names cannot be SQL reserved words in MySQL.

Solution:

Table or column names which are MySQL reserved words must be renamed.

ODIMYS004 - ROWIDs

MySQL does not have an equivalent for the Informix ROWID pseudo-column.

Solution:

Warning: ROWIDs are not supported. You must review the code using ROWIDs and use primary key columns instead.

ODIMYS005 - SERIAL data type

INFORMIX SERIAL data type and automatic number production:

- The table column must be of type SERIAL.
- To generate a new serial, **no value** or **zero** value is given to the INSERT statement :
```
INSERT INTO tab1 ( c ) VALUES ( 'aa' )
INSERT INTO tab1 ( k, c ) VALUES ( 0, 'aa' )
```
- After INSERT, the new SERIAL value is provided in `SQLCA.SQLERRD[2]`.
- INFORMIX allows you to insert rows with a value different from zero for a serial column. Using an explicit value will automatically increment the internal serial counter, to avoid conflicts with future INSERT statements that are using a zero value:
```
CREATE TABLE tab ( k SERIAL );  --> internal counter =
```

```
0
    INSERT INTO tab VALUES ( 0 );   --> internal counter =
1
    INSERT INTO tab VALUES ( 10 );  --> internal counter =
10
    INSERT INTO tab VALUES ( 0 );   --> internal counter =
11
    DELETE FROM tab;                --> internal counter =
11
    INSERT INTO tab VALUES ( 0 );   --> internal counter =
12
```

MySQL AUTO_INCREMENT column definition option:

- In CREATE TABLE, you specify a auto-incremented column with the AUTO_INCREMENT attribute
- Auto-incremented columns have the same behavior as INFORMIX SERIAL columns
- You define a start value with ALTER TABLE tabname AUTO_INCREMENT = value
- The column must be the primary key.
- When using the InnoDB engine, AUTO_INCREMENTED columns might re-use unused sequences after a server restart.
  Actually, when the server restarts, it issues a SELECT MAX(auto_increment_column) on each table with such as column to identify the next sequence to be generated. If you insert rows that generate the numbers 101, 102 and 103, then you delete rows 102 and 103; When the server is restarted next generated number will be 101 + 1 = 102.

Solution:

The INFORMIX SERIAL data type is emulated with MySQL **AUTO_INCREMENT** option. After an insert, **sqlca.sqlerrd[2]** holds the last generated serial value.

Warning: AUTO_INCREMENT columns must be primary keys. This is handled automatically when you create a table in a BDL program.

ODIMYS006 - Outer joins

In INFORMIX SQL, outer tables are defined in the FROM clause with the **OUTER** keyword:

```
SELECT ... FROM a, OUTER(b)
 WHERE a.key = b.akey
SELECT ... FROM a, OUTER(b,OUTER(c))
 WHERE a.key = b.akey
   AND b.key1 = c.bkey1
   AND b.key2 = c.bkey2
```

MySQL 3.23 supports the ANSI outer join syntax:

```
SELECT ... FROM cust LEFT OUTER JOIN order
                    ON cust.key = order.custno
SELECT ...
  FROM cust LEFT OUTER JOIN order
                 LEFT OUTER JOIN item
                 ON order.key = item.ordno
               ON cust.key = order.custno
 WHERE order.cdate > current date
```

See the MySQL reference for a complete description of the syntax.

Solution:

The MySQL interface can convert most INFORMIX OUTER specifications to ANSI outer joins.

Prerequisites:

1. In the FROM clause, the main table must be the first item and the outer tables must figure from left to right in the order of outer levels.
   Example which does not work : "FROM OUTER(tab2), tab1".
2. The outer join in the WHERE part must use the table name as prefix.
   Example : "WHERE tab1.col1 = tab2.col2".

Restrictions:

1. Additional conditions on outer table columns cannot be detected and therefore are not supported :
   Example : "... FROM tab1, OUTER(tab2) WHERE tab1.col1 = tab2.col2 AND tab2.colx > 10".
2. Statements composed by 2 or more SELECT instructions using OUTERs are not supported.
   Example : "SELECT ... UNION SELECT" or "SELECT ... WHERE col IN (SELECT...)"

Notes:

1. Table aliases are detected in OUTER expressions.
   OUTER example with table alias : "OUTER(tab1 alias1)".
2. In the outer join, <outer table>.<col> can be placed on both right or left side of the equal sign.
   OUTER join example with table on the left : "WHERE outertab.col1 = maintab.col2 ".
3. Table names detection is not case-sensitive.
   Example : "SELECT ... FROM tab1, TAB2 WHERE tab1.col1 = tab2.col2".
4. Temporary tables are supported in OUTER specifications.

ODIMYS007a - Database concepts

Most BDL applications use only one database entity (in the meaning of INFORMIX). But the same BDL application can connect to different occurrences of the same database schema, allowing several users to connect to those different databases.

Like INFORMIX servers, MySQL can handle multiple database entities. Tables created by a user can be accessed without the owner prefix by other users as long as they have access privileges to these tables.

Solution:

Create a MySQL database for each INFORMIX database.

ODIMYS008a - Data consistency and concurrency management

Data consistency involves readers which want to access data currently modified by writers and **concurrency data access** involves several writers accessing the same data for modification. **Locking granularity** defines the amount of data concerned when a lock is set (row, page, table, ...).

INFORMIX

INFORMIX uses a locking mechanism to handle data consistency and concurrency. When a process changes database information with UPDATE, INSERT or DELETE, an **exclusive lock** is set on the touched rows. The lock remains active until the end of the transaction. Statements performed outside a transaction are treated as a transaction containing a single operation and therefore release the locks immediately after execution. SELECT statements can set **shared locks** according to the **isolation level**. In case of locking conflicts (for example, when two processes want to acquire an exclusive lock on the same row for modification or when a writer is trying to modify data protected by a shared lock), the behavior of a process can be changed by setting the **lock wait mode**.

Control:

- Lock wait mode : SET LOCK MODE TO ...
- Isolation level : SET ISOLATION TO ...
- Locking granularity : CREATE TABLE ... LOCK MODE {PAGE|ROW}
- Explicit exclusive lock : SELECT ... FOR UPDATE

Defaults:

- The default isolation level is read committed.
- The default lock wait mode is "not wait".

- The default locking granularity is per page.

MySQL

When data is modified, **exclusive locks** are set and held until the end of the transaction. For data consistency, MySQL uses a **locking mechanism**. Readers must wait for writers as in INFORMIX.

Control:

- No lock wait mode control is provided.
- Isolation level : SET TRANSACTION ISOLATION LEVEL ...
- Explicit exclusive lock : SELECT ... FOR UPDATE

Defaults:

- The default isolation level is Read Committed.
- The default locking granularity is per table (pre page when using BDB tables).

Solution:

For portability, it is recommended that you work with INFORMIX in the read committed isolation level, make processes wait for each other (lock mode wait), and create tables with the "lock mode row" option.

See INFORMIX and MySQL documentation for more details about data consistency, concurrency and locking mechanisms.

ODIMYS008b - SELECT FOR UPDATE

A lot of BDL programs use pessimistic locking in order to avoid several users editing the same rows at the same time.

```
DECLARE cc CURSOR FOR
      SELECT ... FROM tab WHERE ... FOR UPDATE
OPEN cc
FETCH cc <-- lock is acquired
...
CLOSE cc <-- lock is released
```

Warning: MySQL locking mechanism depends upon the transaction manager. The default locking granularity is per table when you use the default non-transactional configuration.

Solution:

Review the program logic.

ODIMYS009 - Transactions handling

INFORMIX and MySQL handle transactions in a similar manner.

INFORMIX native mode (non ANSI):

- Transactions are started with "BEGIN WORK".
- Transactions are validated with "COMMIT WORK".
- Transactions are canceled with "ROLLBACK WORK".
- Statements executed outside of a transaction are automatically committed.
- DDL statements can be executed (and canceled) in transactions.

MySQL :

- Transactions are started with "START TRANSACTION".
- Transactions are validated with "COMMIT".
- Transactions are canceled with "ROLLBACK".
- Statements executed outside of a transaction are automatically committed.
- DDL statements can be executed (and canceled) in transactions.

Solution:

INFORMIX transaction handling commands are automatically converted to MySQL instructions to start, validate or cancel transactions.

Warning: MySQL does not support transactions by default. You must set the server system parameter **table_type=InnoDB**.

Regarding the transaction control instructions, the BDL applications do not have to be modified in order to work with MySQL, as long as you have a transaction manager installed with MySQL.

ODIMYS010 - Handling SQL errors when preparing statements

The MySQL connector is implemented with the MySQL **libmysqlclient** API. This library does not provide a way to send SQL statements to the database server during the BDL PREPARE instruction, like the INFORMIX interface. The statement is sent to the server only when opening the cursors or when executing the statement.

Therefore, when preparing a SQL statement with the BDL PREPARE instruction, no SQL errors can be returned if the statement has syntax errors or if a column or a table name does not exist in the database.

However, a SQL error will occur after the OPEN or EXECUTE instructions.

Solution:

Check that your BDL programs do not test STATUS or SQLCA.SQLCODE variable just after PREPARE instructions.

Change the program logic in order to handle the SQL errors when opening the cursors (OPEN) or when executing SQL statements (EXECUTE).

ODIMYS011 - CHARACTER data types

As INFORMIX, MySQL provides the CHAR and VARCHAR data types to store character data.

INFORMIX CHAR type can store up to **32767** characters and the VARCHAR data type is limited to **255** characters.

MySQL CHAR and VARCHAR both have a limit of **255** characters. You can define CHAR and VARCHAR columns with a length greater than 255, but the native data type will automatically be changed to TEXT.

Warning: MySQL automatically creates a TEXT data type when using a size that exceeds 255 characters.

Solution:

The database interface supports character string variables in SQL statements for input (BDL USING) and output (BDL INTO).

ODIMYS012 - Constraints

Constraint naming syntax:

Both INFORMIX and MySQL support primary key, unique, foreign key and default, but the constraint naming syntax is different : MySQL expects the "CONSTRAINT" keyword **before** the constraint specification and INFORMIX expects it **after**.

UNIQUE constraint example:

INFORMIX	MySQL
`CREATE TABLE scott.emp (` `...`	`CREATE TABLE scott.emp (` `...`

```
empcode CHAR(10) UNIQUE
   [CONSTRAINT pk_emp],
...
```

```
empcode CHAR(10)
   [CONSTRAINT pk_emp]
UNIQUE,
...
```

Primary keys:

Like INFORMIX, MySQL creates an index to enforce PRIMARY KEY constraints (some RDBMS do not create indexes for constraints). Using "CREATE UNIQUE INDEX" to define unique constraints is obsolete (use primary keys or a secondary key instead).

Warning: In MySQL, the name of a PRIMARY KEY is PRIMARY.

Unique constraints:

Like INFORMIX, MySQL creates an index to enforce UNIQUE constraints (some RDBMS do not create indexes for constraints).

Warning: When using a unique constraint, INFORMIX allows only one row with a NULL value, while MySQL allows several rows with NULL! Using CREATE UNIQUE INDEX is obsolete.

Foreign keys:

Both INFORMIX and MySQL support the ON DELETE CASCADE option. In MySQL, foreign key constraints are checked immediately, so NO ACTION and RESTRICT are the same.

Check constraints:

Warning: Check constraints are not yet supported in MySQL.

Solution:

Constraint naming syntax:

The database interface does not convert constraint naming expressions when creating tables from BDL programs. Review the database creation scripts to adapt the constraint naming clauses for MySQL.

ODIMYS016 - Defining database users

INFORMIX users are defined at the operating system level, they must be members of the 'informix' group, and the database administrator must grant CONNECT, RESOURCE or DBA privileges to those users.

MySQL users must be registered in the database. They are created with the **GRANT** SQL instruction:

```
$ mysql -u root -pmanager --host orion test

mysql> GRANT ALL PRIVILEGES ON * TO mike IDENTIFIED BY 'pswd';
```

Solution:

According to the application logic (is it a multi-user application ?), you have to create one or several MySQL users.

ODIMYS017 - Temporary tables

INFORMIX temporary tables are created through the **CREATE TEMP TABLE** DDL instruction or through a **SELECT ... INTO TEMP** statement. Temporary tables are automatically dropped when the SQL session ends, but they can be dropped with the DROP TABLE command. There is no name conflict when several users create temporary tables with the same name.

INFORMIX allows you to create indexes on temporary tables. No name conflict occurs when several users create an index on a temporary table by using the same index identifier.

MySQL support temporary tables with the following syntax:

```
    CREATE TEMPORARY TABLE tablename ( coldefs )
    CREATE TEMPORARY TABLE tablename LIKE other-table
```

Solution:

In BDL, INFORMIX temporary tables instructions are converted to generate native SQL Server temporary tables.

ODIMYS018 - Substrings in SQL

INFORMIX SQL statements can use subscripts on columns defined with the character data type:

```
    SELECT ... FROM tab1 WHERE col1[2,3] = 'RO'
    SELECT ... FROM tab1 WHERE col1[10]  = 'R'   -- Same as
col1[10,10]
    UPDATE tab1 SET col1[2,3] = 'RO' WHERE ...
    SELECT ... FROM tab1 ORDER BY col1[1,3]
```

.. while MySQL provides the SUBSTR( ) function, to extract a substring from a string expression :

```
SELECT .... FROM tab1 WHERE SUBSTRING(col1,2,3) = 'RO'
SELECT SUBSTRING('Some text',6,3) ...   -- Gives 'tex'
```

Solution:

You must replace all Informix col[x,y] expressions by SUBSTRING(col,x,y-x+1).

Warning: In UPDATE instructions, setting column values through subscripts will produce an error with MySQL :

```
UPDATE tab1 SET col1[2,3] = 'RO' WHERE ...
```

is converted to:

```
UPDATE tab1 SET SUBSTRING(col1,2,(3-2+1)) = 'RO' WHERE ...
```

Warning: Column subscripts in ORDER BY expressions are also converted and produce an error with MySQL :

```
SELECT ... FROM tab1 ORDER BY col1[1,3]
```

is converted to:

```
SELECT ... FROM tab1 ORDER BY SUBSTRING(col1,1,(3-1+1))
```

ODIMYS019 - Name resolution of SQL objects

INFORMIX uses the following form to identify a SQL object:

```
[database[@dbservername]:][{owner|"owner"}.]identifier
```

With MySQL, an object name takes the following form:

```
[database.]identifier
```

Solution:

Check for single or double quoted table or column names in your source and remove them.

ODIMYS020 - Database object name delimiters

INFORMIX identifies database object names with double quotes, while MySQL does not use the double quotes as database object identifiers.

Solution:

Check your programs for database object names having double quotes:

```
WHERE "tabname"."colname" = "a string value"
```

should be written as follows:

```
WHERE tabname.colname = 'a string value'
```

ODIMYS021 - NUMERIC data types

INFORMIX supports several data types to store numbers:

INFORMIX Data Type	Description
SMALLINT	16 bit integer (-2^15 to 2^15)
INT/INTEGER	32 bit integer (-2^31 to 2^31)
DEC/DECIMAL(p)	Floating-point decimal number
DEC/DECIMAL(p,s)	Fixed-point decimal number
MONEY	Equivalent to DECIMAL(16,2)
MONEY(p)	Equivalent to DECIMAL(p,2)
MONEY(p,s)	Equivalent to DECIMAL(p,s)
REAL/SMALLFLOAT	approx floating point (C float)
DOUBLE PREC./FLOAT	approx floating point (C double)

Solution:

MySQL supports the following data types to store numbers:

MySQL data type	Description
DECIMAL(p,s)	Decimals (with a maximum range like DOUBLE)
SMALLFLOAT	4 bytes variable precision
FLOAT	8 bytes variable precision
SMALLINT	4 bytes integer
INTEGER	8 bytes integer

Warning: MySQL DECIMALs have a maximum range like DOUBLEs.

ODIMYS024 - MATCHES and LIKE in SQL conditions

INFORMIX supports MATCHES and LIKE in SQL statements. MySQL supports the LIKE statement as in INFORMIX, plus the ~ operators that are similar but different from the INFORMIX MATCHES operator.

MATCHES allows brackets to specify a set of matching characters at a given position:

```
( col MATCHES '[Pp]aris' )
( col MATCHES '[0-9][a-z]*' )
```

In this case, the LIKE statement has no equivalent feature.

The following substitutions must be done to convert a MATCHES condition to a LIKE condition:

- MATCHES keyword must be replaced by LIKE.
- All '*' characters must be replaced by '%'.
- All '?' characters must be replaced by '_'.
- Remove all brackets expressions.

Solution:

Warning: SQL statements using MATCHES expressions must be reviewed in order to use LIKE expressions.

See also: MATCHES operator in SQL Programming.

ODIMYS025 - INFORMIX specific SQL statements in BDL

The BDL compiler supports several INFORMIX specific SQL statements that have no meaning when using MySQL:

(removed a sentence as unnecessary)

- CREATE DATABASE
- DROP DATABASE
- START DATABASE (SE only)
- ROLLFORWARD DATABASE
- SET [BUFFERED] LOG
- CREATE TABLE with special options (storage, lock mode, etc.)

Solution:

Review your BDL source and remove all static SQL statements which are INFORMIX specific.

ODIMYS028 - INSERT cursors

INFORMIX supports insert cursors. An "insert cursor" is a special BDL cursor declared with an INSERT statement instead of a SELECT statement. When this kind of cursor is open, you can use the PUT instruction to add rows and the FLUSH instruction to insert the records into the database.

For INFORMIX database with transactions, OPEN, PUT and FLUSH instructions must be executed within a transaction.

MySQL does not support insert cursors.

Solution:

Insert cursors are emulated by the MySQL database interface.

ODIMYS031 - Cursors WITH HOLD

INFORMIX closes opened cursors automatically when a transaction ends unless the WITH HOLD option is used in the DECLARE instruction. In MySQL, opened cursors using SELECT statements without a FOR UPDATE clause are not closed when a transaction ends. Actually, all MySQL cursors are 'WITH HOLD' cursors unless the FOR UPDATE clause is used in the SELECT statement.

Warning: Cursors declared FOR UPDATE and using the WITH HOLD option cannot be supported with MySQL because FOR UPDATE cursors are automatically closed by MySQL when the transaction ends.

Solution:

BDL cursors that are not declared "WITH HOLD" are automatically closed by the database interface when a COMMIT WORK or ROLLBACK WORK is performed.

Warning: Since MySQL automatically closes FOR UPDATE cursors when the transaction ends, opening cursors declared FOR UPDATE and WITH HOLD option results in an SQL error that does not normally appear with INFORMIX, in the same conditions. Review the program logic in order to find another way to set locks.

ODIMYS032 - UPDATE/DELETE WHERE CURRENT OF <cursor>

INFORMIX allows positioned UPDATEs and DELETEs with the "WHERE CURRENT OF <cursor>" clause, if the cursor has been DECLARED with a SELECT ... FOR UPDATE statement.

Solution:

Warning: WHERE CURRENT OF is not supported by MySQL.

ODIMYS034 - Syntax of UPDATE statements

INFORMIX allows a specific syntax for UPDATE statements:

```
UPDATE table SET ( <col-list> ) = ( <val-list> )
```

or

```
UPDATE table SET table.* = myrecord.*
UPDATE table SET * = myrecord.*
```

Solution:

Static UPDATE statements using the above syntax are converted **by the compiler** to the standard form:

```
UPDATE table SET column=value [,...]
```

ODIMYS036 - INTERVAL data type

INFORMIX INTERVAL data type stores a value that represents a span of time. INTERVAL types are divided into two classes : *year-month intervals* and *day-time intervals*.

MySQL provides an INTERVAL data type, but it is totally different from the INFORMIX INTERVAL type. For example, you specify an INTERVAL literal as follows :

```
25 years 2 months 23 days
```

Solution:

Warning: The INTERVAL data type is not well supported because the database server has no equivalent native data type. However, you can store into and retrieve from CHAR columns BDL INTERVAL values.

ODIMYS039 - Data storage concepts

An attempt should be made to preserve as much of the storage information as possible when converting from INFORMIX to MySQL. Most important storage decisions made for INFORMIX database objects (like initial sizes and physical placement) can be reused for the MySQL database.

Storage concepts are quite similar in INFORMIX and in MySQL, but the names are different.

ODIMYS046 - The LOAD and UNLOAD instructions

INFORMIX provides two SQL instructions to export / import data from / into a database table: The UNLOAD instruction copies rows from a database table into a text file and the LOAD instructions insert rows from a text file into a database table.

MySQL does not provide LOAD and UNLOAD instructions, but provides external tools like SQL*Plus and SQL*Loader.

Solution:

LOAD and UNLOAD instructions are supported.

Warning: There is a difference when using MySQL DATE columns; DATE columns created in the MySQL database are similar to INFORMIX DATETIME YEAR TO SECOND columns. In LOAD and UNLOAD, all MySQL DATE columns are treated as INFORMIX DATETIME YEAR TO SECOND columns and thus will be unloaded with the "YYYY-MM-DD hh:mm:ss" format.
The same problem appears for INFORMIX INTEGER and SMALLINT values which are stored in an MySQL database as NUMBER(?) columns. Those values will be unloaded as INFORMIX DECIMAL(10) and DECIMAL(5) values, that is, with a trailing dot-zero ".0".

Warning: When using an INFORMIX database, simple dates are unloaded with the DBDATE format (ex: "23/12/1998"). Therefore, unloading from an INFORMIX database for loading into a MySQL database is not supported.

Warning: UNLOAD instructions based on SELECT statements using date expressions such as `"WHERE ?-datecol>?"` are not supported because of database server limitations. A syntax error would be raised in this case.

ODIMYS047 - SQL Interruption

With Informix, it is possible to interrupt a long running query if the SQL INTERRUPT ON option is set by the Genero program. The database server returns SQLCODE -213, which can be trapped to detect a user interruption.

```
MAIN
  DEFINE n INTEGER
  DEFER INTERRUPT
  OPTIONS SQL INTERRUPT ON
  DATABASE test1
  WHENEVER ERROR CONTINUE
  -- Start long query (self join takes time)
  -- From now on, user can hit CTRL-C in TUI mode to stop the query
  SELECT COUNT(*) INTO n FROM customers a, customers b
       WHERE a.cust_id <> b.cust_id
  IF SQLCA.SQLCODE == -213 THEN
     DISPLAY "Statement was interrupted by user..."
     EXIT PROGRAM 1
  END IF
  WHENEVER ERROR STOP
  ...
END MAIN
```

Solution:

Warning: SQL Interruption is not supported with MySQL.

ODIMYS100 - Data type conversion table

INFORMIX Data Types	MySQL Data Types
CHAR(n)	CHAR(n) (n>255c=> TEXT(n))
VARCHAR(n)	VARCHAR(n) (n>255c=> TEXT(n))
INTEGER	INTEGER
SMALLINT	SHORT
FLOAT[(n)]	FLOAT
SMALLFLOAT	SMALLFLOAT
DECIMAL(p,s)	DECIMAL(p,s) (max range like double!)
MONEY(p,s)	DECIMAL(p,s) (max range like double!)

DATE	DATE
DATETIME HOUR TO SECOND	TIME
DATETIME q1 TO q2	DATETIME (YYY-MM-DD hh:mm:ss)
INTERVAL q1 TO q2	N/A

ODI Adaptation Guide For Sybase ASA 8.x

Runtime configuration

Install Sybase and create a database
Prepare the runtime environment

Database concepts

Database concepts
Data storage concepts
Data consistency and concurrency management
Transactions handling
Defining database users
Setting privileges

Data dictionary

CHARACTER data types
NUMERIC data types
DATE and DATETIME data types
INTERVAL data type
SERIAL data type
ROWIDs
Case sensitivity
Very large data types
National character data types
The ALTER TABLE instruction
Constraints
Triggers
Stored procedures
Name resolution of SQL objects
Setup database statistics
Data type conversion table

Data manipulation

Reserved words
Outer joins

Transactions handling
Temporary tables
Substrings in SQL
Name resolution of SQL objects
String delimiters
Getting one row with SELECT
MATCHES and LIKE conditions
Querying system catalog tables
Syntax of UPDATE statements

BDL programming

SERIAL data type
INFORMIX specific SQL statements in BDL
INSERT cursors
Cursors WITH HOLD
SELECT FOR UPDATE
The LOAD and UNLOAD instructions
SQL Interruption

Runtime configuration

Install Sybase ASA and create a database

1. Install Sybase ASA software on your computer.
2. Create a Sybase **database** entity with **dbinit** tool. Go to a directory where the database files must be created and run the dbinit tool.
   Warning: Create the database with case-sensitivity and blank padding for string comparisons:

   ```
   $ cd datadirectory
   $ dbinit -c -b databasename
   ```
3. Make sure that the database option ALLOW_NULLS_BY_DEFAULT option is set to ON.
   Warning: If this option is set to OFF, columns created without NULL or NOT NULL are NOT NULL by default.
4. Try to connect to the new created database with the **dbisql** tool. The default database user is **DBA/SQL**.

Warning: User logins and passwords are case sensitive!

5. Declare a database user dedicated to your application: the **application administrator**.

```
   grant connect to appadmin
identified by password
   grant resource to appadmin
```

   See documentation for more details about database users and privileges. You must create groups to make tables visible to all users.
6. If you plan to use SERIAL emulation based on triggers using a registration table, create the SERIALREG table. Create the triggers for each table using a SERIAL. See issue ODIASA005 for more details.
7. Create the **application tables**. Do not forget to convert Informix data types to Sybase ASA data types. See issue ODIASA100 for more details.
   Warning: In order to make application tables visible to all users, make sure that all users are members of the group of the owner of the application tables. For more details, see ASA documentation ("Database object names and prefixes").

Prepare the runtime environment

1. **Warning: No ODBC client environment is required. The Sybase ASA driver is designed to be linked with ESQL/C libs (libdblib8+libdbtools8).**
2. If you want to connect to a remote database server, you must have the **Sybase ASA Client Software** installed on the computer running 4gl applications.
3. Test the Sybase ASA Client Software: Make sure Sybase ASA is started on the database server and try to connect to a database by using the **Interactive SQL** tool.
4. Set up the **fglprofile** entries for database connections.
5. Define the connection timeout with the following fglprofile entry:

```
dbi.database.dbname.asa.logintime =
integer
```

   This entry defines the number of seconds to wait

for a connection.
Default is 5 seconds.

6. In order to connect to Sybase ASA, you must have a database driver "**dbmasa***" installed.

ODIASA001 - DATE and DATETIME data types

INFORMIX provides two data types to store dates and time information:

- **DATE** = for year, month and day storage.
- **DATETIME** = for year to fraction(1-5) storage.

Sybase ASA provides two data type to store dates :

- **DATE** = for year, month, day storage.
- **TIME** = for hour, minutes, seconds, fraction(3) storage.
- **TIMESTAMP** = for year, month, day, hour, minutes, seconds, fraction(3) storage.

String representing date time information :

INFORMIX is able to convert quoted strings to DATE / DATETIME data if the string contents matches environment parameters (i.e. DBDATE, GL_DATETIME). As in INFORMIX, Sybase ASA can convert quoted strings representing datetime data in the ANSI format. The CONVERT( ) SQL function allows you to convert strings to dates.

Date time arithmetic:

- INFORMIX supports date arithmetic on DATE and DATETIME values. The result of an arithmetic expression involving dates/times is a number of days when only DATEs are used and an INTERVAL value if a DATETIME is used in the expression.
- INFORMIX automatically converts an integer to a date when the integer is used to set a value of a date column. Sybase ASA does not support this automatic conversion.
- Complex DATETIME expressions (involving INTERVAL values for example) are INFORMIX specific and have no equivalent in Sybase ASA.
- Sybase ASA allows the following arithmetic on dates:
  - Date + integer => add n days to the date.
  - Date - integer => subtract n days from the date.
  - Timestamp + integer => add n days to the timestamp.
  - Timestamp - integer => subtract n days from the timestamp.
  - Date - Date => compute number of days between 2 dates.
  - Timestamp - Timestamp => compute number of days between 2 timestamps.
  - Date + Time => Create a Timestamp combining given date & time.
- INFORMIX converts automatically an integer to a date when the integer is used to set a value of a date column. Sybase ASA does not support this automatic conversion.

Solution:

Sybase ASA has the same **DATE** data type as INFORMIX (year, month, day). So you can use Sybase ASA DATE data type for Informix DATE columns.

Sybase ASA **TIME** data type can be used to store INFORMIX DATETIME HOUR TO FRAC(3) values. The database interface makes the conversion automatically.

INFORMIX DATETIME values with any precision from YEAR to FRACTION(5) can be stored in Sybase ASA **TIMESTAMP** columns. The database interface makes the conversion automatically. Missing date or time parts default to 1900-01-01 00:00:00.0. For example, when using a DATETIME HOUR TO MINUTE with the value of "11:45", the ASA TIMESTAMP value will be "1900-01-01 11:45:00.0".

Warning: Literal DATETIME and INTERVAL expressions (i.e. DATETIME (1999-10-12) YEAR TO DAY) are not converted.

Warning: Using integers as a number of days in an expression with dates is not supported by Sybase ASA. Check your code to detect where you are using integers with DATE columns.

Warning: It is strongly recommended to use BDL variables in dynamic SQL statements instead of quoted strings representing DATEs. For example :

```
   LET stmt = "SELECT ... FROM customer WHERE creat_date >'", adate,"'"
```

is not portable; use a question mark place holder instead and OPEN the cursor USING adate :

```
   LET stmt = "SELECT ... FROM customer WHERE creat_date > ?"
```

ODIASA003 - Reserved words

Even if ASA allows SQL reserved keywords as SQL object names if enclosed in double quotes (`"create table "table" ( col1 int )"`), you should take care of your existing database schema and check that you do not use Sybase ASA SQL words.

Solution:

Database objects having a name which is a Sybase ASA SQL reserved word must be renamed.

All BDL application sources must be verified. To check if a given keyword is used in a source, you can use UNIX 'grep' or 'awk' tools. Most modifications can be automatically done with UNIX tools like 'sed' or 'awk'.

ODIASA004 - ROWIDs

When creating a table, INFORMIX automatically adds a "ROWID" column of type integer (applies to non-fragmented tables only). The ROWID column is auto-filled with a unique number and can be used like a primary key to access a given row.

Sybase ASA tables have no ROWIDs.

Solution:

If the BDL application uses ROWIDs, the program logic should be reviewed in order to use the real primary keys (usually, serials which can be supported).

All references to SQLCA.SQLERRD[6] must be removed because this variable will not hold the ROWID of the last INSERTed or UPDATEd row when using the Sybase ASA interface.

ODIASA005 - SERIAL data type

INFORMIX SERIAL data type and automatic number production :

- The table column must be of type SERIAL.
- To generate a new serial, **no value** or a **zero** value is given to the INSERT statement :
```
INSERT INTO tab1 ( c ) VALUES ( 'aa' )
INSERT INTO tab1 ( k, c ) VALUES ( 0, 'aa' )
```
- After INSERT, the new SERIAL value is provided in `SQLCA.SQLERRD[2]`.

INFORMIX allows you to insert rows with a value different from zero for a serial column. Using an explicit value will automatically increment the internal serial counter, to avoid conflicts with future INSERTs that are using a zero value :

```
CREATE TABLE tab ( k SERIAL );  --> internal counter = 0
INSERT INTO tab VALUES ( 0 );   --> internal counter = 1
INSERT INTO tab VALUES ( 10 );  --> internal counter = 10
INSERT INTO tab VALUES ( 0 );   --> internal counter = 11
DELETE FROM tab;                --> internal counter = 11
INSERT INTO tab VALUES ( 0 );   --> internal counter = 12
```

Sybase ASA **IDENTITY** columns :

- When creating a table, the IDENTITY keyword must be specified after the column data type:
```
CREATE TABLE tab1 ( k integer identity, c char(10) )
```
- You cannot specify a start value
- A new number is automatically created when inserting a new row :
```
INSERT INTO tab1 ( c ) VALUES ( 'aaa' )
```

- To get the last generated number, Sybase ASA provides a global variable :
  ```
  SELECT @@IDENTITY
  ```
- When IDENTITY_INSERT is ON, you can set a specific value into a IDENTITY column, but zero does not generate a new serial:
  ```
  INSERT INTO tab1 ( k, c ) VALUES ( 100, 'aaa' )
  ```

INFORMIX SERIALs and MS Sybase ASA IDENTITY columns are quite similar; the main difference is that MS Sybase ASA does not generate a new serial when you specify a zero value for the identity column.

Solution **:**

You are free to use **IDENTITY columns** (1) or **insert triggers based on the SERIALREG** table (2). The first solution is faster, but does not allow explicit serial value specification in insert statements; the second solution is slower but allows explicit serial value specification. You can initially use the second solution to have unmodified 4gl programs working on Sybase ASA, but you should update your code to use native IDENTITY columns for performance.

With the following fglprofile entry, you define the technique to be used for SERIAL emulation :

```
   dbi.database.<dbname>.ifxemul.datatype.serial.emulation =
{"native"|"regtable"}
```

The '**native**' value defines the IDENTITY column technique and the '**regtable**' defines the trigger technique.

This entry must be used with :

```
   dbi.database.<dbname>.ifxemul.datatype.serial = {true|false}
```

If this entry is set to false, the emulation method specification entry is ignored.

Warning : When no entry is specified, the default is SERIAL emulation enabled with 'native' method (IDENTITY-based).

1. Using IDENTITY columns

In database creation scripts, all SERIAL data types must be converted by hand to INTEGER IDENTITY data types.

Warning : Start values SERIAL(n) cannot be converted, there is no INTEGER IDENTITY(n) in Sybase ASA.

Tables created from the BDL programs can use the SERIAL data type : When a BDL program executes a CREATE [TEMP] TABLE with a SERIAL column, the database interface automatically converts the "SERIAL[(n)]" data type to "INTEGER IDENTITY[(n,1)]".

In BDL, the new generated SERIAL value is available from the SQLCA.SQLERRD[2] variable. This is supported by the database interface which performs a "SELECT @@IDENTITY".

Warning : When you insert a row with zero as serial value, the serial column gets the value zero. You must review all INSERT statements using zero for the serial column. For example, the following statement:

```
INSERT INTO tab (col1,col2) VALUES (0, p_value)
```

must be converted to :

```
INSERT INTO tab (col2) VALUES (p_value)
```

Warning : SELECT * FROM table INTO TEMP with original table having an IDENTITY column is not supported: The database driver must convert the Informix SELECT INTO TEMP statement into a SELECT INTO #tab + INSERT (see temporary tables) because ODBC does not allow SQL parameters in DDL statements. As MS Sybase ASA does not allow you to insert a row by giving the identity column, the INSERT statement fails.

2. Using triggers with the SERIALREG table

First, you must prepare the database and create the SERIALREG table as follows:

```
CREATE TABLE SERIALREG (
     TABLENAME VARCHAR(50) NOT NULL,
     LASTSERIAL INTEGER NOT NULL,
     PRIMARY KEY ( TABLENAME )
)
```

In database creation scripts, all SERIAL[(n)] data types must be converted to INTEGER data types and you must create one trigger for each table. To know how to write those triggers, you can create a small Genero program that creates a table with a SERIAL column. Set the FGLSQLDEBUG environment variable and run the program. The debug output will show you the native trigger creation command.

Tables created from the BDL programs can use the SERIAL data type. When a BDL program executes a CREATE [TEMP] TABLE with a SERIAL column, the database interface automatically converts the "SERIAL[(n)]" data type to "INTEGER" and creates the insert triggers.

Warning : Sybase ASA does not allow you to create triggers on temporary tables. Therefore, you cannot create temp tables with a SERIAL column when using this solution.

Warning : SELECT ... INTO TEMP statements using a table created with a SERIAL column do not automatically create the SERIAL triggers in the temporary table. The type of the column in the new table is INTEGER.

Warning : Sybase ASA triggers are not automatically dropped when the corresponding table is dropped. Database administrators must be aware of this behavior when managing schemas.

Warning : INSERT statements using NULL for the SERIAL column will produce a new serial value, instead of a NULL like INFORMIX does:

```
INSERT INTO tab (col1,col2) VALUES (NULL,'data')
```

This behavior is mandatory in order to support INSERT statements which do not use the serial column :

```
INSERT INTO tab (col2) VALUES ('data')
```

Check if your application uses tables with a SERIAL column that can contain a NULL value.

Warning : The serial production is based on the SERIALREG table which registers the last generated number for each table. If you delete rows of this table, sequences will restart at 1 and you will get unexpected data.

ODIASA006 - Outer joins

The syntax of OUTER joins is quite different in INFORMIX and Sybase ASA :

In INFORMIX SQL, outer tables are defined in the FROM clause with the **OUTER** keyword :

```
SELECT ... FROM cust, OUTER(order)
 WHERE cust.key = order.custno
SELECT ... FROM cust, OUTER(order,OUTER(item))
 WHERE cust.key = order.custno
   AND order.key = item.ordno
   AND order.accepted = 1
```

Sybase ASA Version 7 supports the ANSI outer join syntax :

```
SELECT ... FROM cust LEFT OUTER JOIN order
                     ON cust.key = order.custno
SELECT ...
  FROM cust LEFT OUTER JOIN order
                 LEFT OUTER JOIN item
                 ON order.key = item.ordno
            ON cust.key = order.custno
 WHERE order.accepted = 1
```

The old way to define outer joins in Sybase ASA looks like the following :

```
SELECT ... FROM a, b WHERE a.key *= b.key
```

See the Sybase ASA reference manual for a complete description of the syntax.

Solution:

The Sybase ASA interface can convert simple INFORMIX OUTER specifications to Sybase ASA ANSI outer joins.

Prerequisites :

1. The outer join in the WHERE part must use the table name as prefix.
   Example : "`WHERE tab1.col1 = tab2.col2` ".
2. Additional conditions on outer table columns cannot be detected and therefore are not supported :
   Example : "... `FROM tab1, OUTER(tab2) WHERE tab1.col1 = tab2.col2 AND tab2.colx > 10`".
3. Statements composed of 2 or more SELECT instructions using OUTERs are not supported.
   Example : "`SELECT ... UNION SELECT" or "SELECT ... WHERE col IN (SELECT...)`"

Notes :

1. Table aliases are detected in OUTER expressions.
   OUTER example with table alias : "`OUTER( tab1 alias1)`".
2. In the outer join, <outer table>.<col> can be placed on both right or left sides of the equal sign.
   OUTER join example with table on the left : "`WHERE outertab.col1 = maintab.col2` ".
3. Table names detection is not case-sensitive.
   Example : "`SELECT ... FROM tab1, TAB2 WHERE tab1.col1 = tab2.col2`".
4. Temporary tables are supported in OUTER specifications.

ODIASA007a - Database concepts

As in INFORMIX, an Sybase ASA engine can manage multiple database entities. When creating a database object such as a table, Sybase ASA allows you to use the same object name in different databases.

ODIASA008a - Data consistency and concurrency management

Data consistency involves readers which want to access data currently modified by writers and **concurrency data access** involves several writers accessing the same data for modification. **Locking granularity** defines the amount of data concerned when a lock is set (row, page, table, ...).

INFORMIX

INFORMIX uses a locking mechanism to manage data consistency and concurrency. When a process modifies data with UPDATE, INSERT or DELETE, an exclusive lock is

set on the affected rows. The lock is held until the end of the transaction. Statements performed outside a transaction are treated as a transaction containing a single operation and therefore release the locks immediately after execution. SELECT statements can set **shared locks** according to the **isolation level**. In case of locking conflicts (for example, when two processes want to acquire an exclusive lock on the same row for modification or when a writer is trying to modify data protected by a shared lock), the behavior of a process can be changed by setting the **lock wait mode**.

Control :

- Isolation level : SET ISOLATION TO ...
- Lock wait mode : SET LOCK MODE TO ...
- Locking granularity : CREATE TABLE ... LOCK MODE {PAGE|ROW}
- Explicit locking : SELECT ... FOR UPDATE

Defaults :

- The default isolation level is read committed.
- The default lock wait mode is "not wait".
- The default locking granularity is per page.

Sybase ASA

As in INFORMIX, Sybase ASA uses locks to manage data consistency and concurrency. The database manager sets **exclusive locks** on the modified rows and **shared locks** when data is read, according to the **isolation level**. The locks are held until the end of the transaction. When multiple processes want to access the same data, the latest processes must wait until the first finishes its transaction or the lock timeout occurred. The **lock granularity** is at the row or table level. For more details, see Sybase ASA's Documentation, "Accessing and Changing Data", "Locking".

Control :

- Lock wait mode : Can only be set to on or off, and a timeout can be specified, with:
  SET TEMPORARY OPTION BLOCKING = { ON | OFF }
  SET TEMPORARY OPTION BLOCKING_TIMEOUT = n
- Isolation level : Can be set with:
  SET TEMPORARY OPTION ISOLATION LEVEL = {1|2|3|4}
- Locking granularity : Row level.
- Explicit locking : SELECT ... FOR UPDATE

Defaults :

- The default isolation level is Read Committed (readers cannot see uncommitted data; no shared lock is set when reading data).

Solution:

For portability, it is recommended that you work with INFORMIX in the read committed isolation level, to make processes wait for each other (lock mode wait) and to create tables with the "lock mode row" option.

See INFORMIX and Sybase ASA documentation for more details about data consistency, concurrency and locking mechanisms.

When using SET LOCK MODE and SET ISOLATION LEVEL instructions, the database interface sets automatically the native database session options.

ODIASA008b - SELECT FOR UPDATE

A lot of BDL programs use pessimistic locking in order to avoid several users editing the same rows at the same time.

```
DECLARE cc CURSOR FOR
      SELECT ... FOR UPDATE
OPEN cc
FETCH cc <-- lock is acquired
CLOSE cc <-- lock is released
```

- A transaction must be started before opening cursors declared for update.
- The row must be fetched in order to set the lock.
- The lock is released when the transaction ends (if the cursor is not declared "WITH HOLD") or when the cursor is closed.

Sybase ASA allows individual and exclusive row locking by using the FOR UPDATE clause, as Informix.

- Individual locks are acquired when fetching the rows.
- When the cursor (WITH HOLD) is opened outside a transaction, locks are released when the cursor is closed.
- When the cursor is opened inside a transaction, locks are released when the transaction ends.

Sybase ASA's locking granularity is at the row level, page level or table level (the level is automatically selected by the engine for optimization).

To control the behavior of the program when locking rows, INFORMIX provides a specific instruction to set the wait mode:

```
SET LOCK MODE TO { WAIT | NOT WAIT | WAIT seconds }
```

The default mode is WAIT. This as an INFORMIX specific SQL statement.

Solution:

SELECT FOR UPDATE statements are well supported.

Warning : Sybase ASA locks the rows when you open the cursor. You will have to test SQLCA.SQLCODE after doing an OPEN.

Warning : The database interface is based on an emulation of an Informix engine using transaction logging. Therefore, opening a SELECT ... FOR UPDATE cursor declared outside a transaction will raise an SQL error -255 (not in transaction).

Warning : The SELECT FOR UPDATE statement cannot contain an ORDER BY clause if you want to perform positioned updates/deletes with WHERE CURRENT OF.

You must review the program logic if you use pessimistic locking; it is based on the NOT WAIT mode, which is not supported by Sybase ASA.

ODIASA009 - Transactions handling

INFORMIX and Sybase ASA handle transactions in a similar manner.

INFORMIX native mode (non ANSI) :

- Transactions are started with "BEGIN WORK".
- Transactions are validated with "COMMIT WORK".
- Transactions are canceled with "ROLLBACK WORK".
- Statements executed outside of a transaction are automatically committed.
- DDL statements can be executed (and canceled) in transactions.

Sybase ASA :

- Transactions are started with "BEGIN TRANSACTION [name]".
- Transactions are validated with "COMMIT TRANSACTION [name]".
- Transactions are canceled with "ROLLBACK TRANSACTION [name]".
- Transactions save points can be placed with "SAVEPOINT [name]".
- Sybase ASA supports named and nested transactions.
- By default transactions are started implicitly as in the ANSI specification.
  This behavior can be changed with:
  SET TEMPORARY OPTION CHAINED = OFF
- DDL statements are not supported in transactions blocks.

Transactions in stored procedures : avoid using transactions in stored procedures to allow the client applications to handle transactions, according to the transaction model.

Solution:

INFORMIX transaction handling commands are automatically converted to Sybase ASA instructions to start, validate or cancel transactions.

The database driver sets the "CHAINED" option to OFF when connecting to the server.

Regarding the transaction control instructions, the BDL applications do not have to be modified in order to work with Sybase ASA.

ODIASA011 - CHARACTER data types

As in INFORMIX, Sybase ASA provides the CHAR and VARCHAR data types to store character data.

INFORMIX CHAR type can store up to **32767** characters and the VARCHAR data type is limited to **255** characters.

Sybase ASA CHAR and VARCHAR both have a limit of **32767** characters.

Sybase ASA provides the LONG VARCHAR data type to store large character strings. Only the LIKE operator can be used for searches. LONG VARCHAR columns cannot be used in classic comparison expressions (as col = 'value').

Solution:

The database interface supports character string variables in SQL statements for input (BDL USING) and output (BDL INTO) for CHAR and VARCHAR data types.

Warning : Check that your database schema does not use CHAR or VARCHAR types with a length exceeding the Sybase ASA limit.

Warning: TEXT values cannot be used as input or output parameters in SQL statements and therefore are not supported.

ODIASA012 - Constraints

Constraint naming syntax :

Both INFORMIX and Sybase ASA support primary key, unique, foreign key, default and check constraints. But Sybase ASA does not support constraint naming syntax:

UNIQUE constraint example :

INFORMIX	Sybase ASA
`CREATE TABLE scott.emp (` `...`	`CREATE TABLE scott.emp (` `...`

```
empcode CHAR(10) UNIQUE      empcode CHAR(10)UNIQUE,
   [CONSTRAINT pk_emp],      ...
...
```

Solution:

Constraint naming syntax :

The database interface does not convert constraint naming expressions when creating tables from BDL programs. Review the database creation scripts to adapt the constraint naming clauses for Sybase ASA.

ODIASA013 - Triggers

INFORMIX and Sybase ASA provide triggers with similar features, but the programming languages are totally different.

Warning : Sybase ASA does not support triggers on temporary tables.

Solution:

INFORMIX triggers must be converted to Sybase ASA triggers "by hand".

ODIASA014 - Stored procedures

Both INFORMIX and Sybase ASA support stored procedures, but the programming languages are totally different :

- INFORMIX stored procedures must be written in **SPL**.
- Sybase ASA stored procedures must be written in **Sybase ASA SQL**.

Solution:

INFORMIX stored procedures must be converted to Sybase ASA "by hand".

ODIASA016a - Defining database users

INFORMIX users are defined at the operating system level, they must be members of the 'informix' group, and the database administrator must grant CONNECT, RESOURCE or DBA privileges to those users.

Before a user can access an Sybase ASA database, the system administrator (DBA) must declare the application users in the database with the GRANT statement. You may also need to define groups in order to make tables visible to other users.

Solution:

See Sybase ASA documentation for more details on database logins and users.

ODIASA016b - Setting privileges

INFORMIX and Sybase ASA user privileges management are quite similar.

Sybase ASA provides **user groups** to grant or revoke permissions to more than one user at the same time.

ODIASA017 - Temporary tables

INFORMIX temporary tables are created through the **`CREATE TEMP TABLE`** DDL instruction or through a **`SELECT ... INTO TEMP`** statement. Temporary tables are automatically dropped when the SQL session ends, but they can also be dropped with the DROP TABLE command. There is no name conflict when several users create temporary tables with the same name.

Warning : The CREATE TEMP TABLE and SELECT INTO TEMP statements are not supported in Sybase ASA.

Sybase ASA supports temporary tables by using the DECLARE LOCAL TEMPORARY TABLE statement.

Solution:

The CREATE TEMP TABLE statements are converted by the database interface to DECLARE LOCAL TEMPORARY TABLE statements.

Warning : SELECT INTO TEMP statements cannot be converted, because Sybase ASA does not provide a way to create a temporary table from a result set, such as CREATE TABLE xx AS (SELECT ...).

ODIASA018 - Substrings in SQL

INFORMIX SQL statements can use subscripts on columns defined with the character data type :

```
    SELECT ... FROM tab1 WHERE col1[2,3] = 'RO'
    SELECT ... FROM tab1 WHERE col1[10]  = 'R'   -- Same as
col1[10,10]
    UPDATE tab1 SET col1[2,3] = 'RO' WHERE ...
    SELECT ... FROM tab1 ORDER BY col1[1,3]
```

.. while Sybase ASA provides the SUBSTR( ) function, to extract a substring from a string expression :

```
    SELECT .... FROM tab1 WHERE SUBSTRING(col1,2,2) = 'RO'
    SELECT SUBSTRING('Some text',6,3) FROM DUAL       -- Gives
'tex'
```

Solution:

You must replace all Informix col[x,y] expressions by SUBSTRING(col,x,y-x+1).

Warning: In UPDATE instructions, setting column values through subscripts will produce an error with Sybase ASA :

```
    UPDATE tab1 SET col1[2,3] = 'RO' WHERE ...
```

is converted to :

```
    UPDATE tab1 SET SUBSTRING(col1,2,3-2+1) = 'RO' WHERE ...
```

Warning: Column subscripts in ORDER BY expressions are also converted and produce an error with Sybase ASA :

```
    SELECT ... FROM tab1 ORDER BY col1[1,3]
```

is converted to :

```
    SELECT ... FROM tab1 ORDER BY SUBSTRING(col1,1,3-1+1)
```

ODIASA019 - Name resolution of SQL objects

INFORMIX uses the following form to identify an SQL object :

```
  [database[@dbservername]:][{owner|"owner"}.]identifier
```

With Sybase ASA, an object name takes the following form :

```
  [{owner|"owner"}.]{identifier|"identifier"}
```

Identifiers have a maximum length of 128 bytes and are composed of alphabetic characters (_, @, #, $ are considered as alphabetic characters) or digits. The first character must be alphabetic.

INFORMIX database object names are **not case sensitive** in non-ANSI databases. Sybase ASA database objects names are **case sensitive** by default, but this is related

to the -c option of the **dbinit** command. Databases must be created as case-sensitive, otherwise a string comparison such as "abc"="ABC" would evaluate to TRUE.

ODIASA020 - String delimiters

The ANSI string delimiter character is the single quote ('string'). Double quotes are used to delimit database object names ("object-name").

Example : `WHERE "tabname"."colname" = 'a string value'`

As INFORMIX, Sql Server Anywhere allows to use double quotes as string delimiters, if the QUOTED_IDENTIFIER session option is OFF, the default is ON:

```
SET TEMPORARY OPTION QUOTED_IDENTIFIER = OFF
```

Remark : This problem concerns only double quotes within SQL statements. Double quotes used in BDL string expressions are not subject of SQL compatibility problems.

Solution:

When the **ifxemul.dblquotes** option is set, the Sybase ASA database interface converts all double quotes to single quotes in SQL statements. The database driver does not set the `QUOTED_IDENTIFIER` option implicitly.

ODIASA021 - NUMERIC data types

Sybase ASA offers numeric data types which are quite similar to INFORMIX numeric data types. The table below shows general conversion rules for numeric data types :

INFORMIX	Sybase ASA
SMALLINT	**SMALLINT**
INTEGER (synonym: INT)	**INTEGER** (synonym: INT)
DECIMAL[(p[,s)] (synonyms: DEC, NUMERIC) DECIMAL(p,s) defines a fixed point decimal where **p** is the total number of significant digits and **s** the number of digits that fall on the right of the decimal point. DECIMAL(p) defines a floating point decimal where **p** is the total number of significant digits.	**DECIMAL[(p[,s)]** (synonyms: DEC, NUMERIC) DECIMAL[(p[,s])] defines a fixed point decimal where **p** is the total number of significant digits and **s** the number of digits that fall on the right of the decimal point. Without any decimal storage specification, the precision defaults to 30 and the scale defaults to 6 :

The precision **p** can be from 1 to 32. DECIMAL is treated as DECIMAL(16).	- DECIMAL in Sybase ASA = DECIMAL(30,0) in INFORMIX - DECIMAL(p) in Sybase ASA = DECIMAL(p,6) in INFORMIX
MONEY[(p[,s])	Sybase ASA provides the MONEY and SMALLMONEY data types, but the currency symbol handling is quite different. Therefore, Informix MONEY columns should be implemented as **DECIMAL** columns in Sybase ASA.
SMALLFLOAT (synonyms: REAL)	**REAL**
FLOAT[(n)] (synonyms: DOUBLE PRECISION) The precision (n) is ignored.	**FLOAT(n)** (synonyms: DOUBLE PRECISION) Where n must be from 1 to 15.

Solution:

In BDL programs :

When creating tables from BDL programs, the database interface automatically converts INFORMIX data types to corresponding Sybase ASA data types.

Database creation scripts :

- SMALLINT and INTEGER columns do not have to use another data type in Sybase ASA.
- For DECIMALs, check the precision limit. Always use a precision and a scale.
- Convert MONEY columns to DECIMAL(p,s) columns. Always use a precision and a scale.
- Convert SMALLFLOAT columns to REAL columns.
- Since FLOAT precision is ignored in INFORMIX, convert this data type to FLOAT(15).

ODIASA022 - Getting one row with SELECT

With INFORMIX, you must use the system table with a condition on the table id :

```
SELECT user FROM systables WHERE tabid=1
```

With Sybase ASA, you can omit the FROM clause to generate one row only:

```
SELECT user
```

Solution:

Check the BDL sources for "FROM systables WHERE tabid=1" and use dynamic SQL to resolve this problem.

ODIASA024 - MATCHES and LIKE in SQL conditions

INFORMIX supports MATCHES and LIKE in SQL statements, while Sybase ASA supports the LIKE statement only.

The MATCHES operator of INFORMIX uses the star, question mark and square braces wildcard characters.
The LIKE operator of Sybase ASA offers the percent, underscore and square braces wildcard characters.

The following substitutions must be made to convert a MATCHES condition to a LIKE condition :

- MATCHES keyword must be replaced by LIKE.
- All '*' characters must be replaced by '%'.
- All '?' characters must be replaced by '_'.

Solution:

Warning: SQL statements using MATCHES expressions must be reviewed in order to use LIKE expressions.

See also: MATCHES operator in SQL Programming.

ODIASA025 - INFORMIX specific SQL statements in BDL

The BDL compiler supports several INFORMIX specific SQL statements that have no meaning when using Sybase ASA.

Examples :

- CREATE DATABASE dbname IN dbspace WITH BUFFERED LOG
- START DATABASE (SE only)
- ROLLFORWARD DATABASE
- CREATE TABLE ... IN dbspace WITH LOCK MODE ROW

Solution:

Review your BDL source and remove all static SQL statements which are INFORMIX specific.

ODIASA028 - INSERT cursors

INFORMIX supports insert cursors. An "insert cursor" is a special BDL cursor declared with an INSERT statement instead of a SELECT statement. When this kind of cursor is open, you can use the PUT instruction to add rows and the FLUSH instruction to insert the records into the database.

For INFORMIX database with transactions, OPEN, PUT and FLUSH instructions must be executed within a transaction.

Sybase ASA does not support insert cursors.

Solution:

Insert cursors are emulated by the Sybase ASA database interface.

ODIASA030 - Very large data types

Both INFORMIX and Sybase ASA provide special data types to store very large texts or images.

Sybase ASA recommends the following conversion rules :

INFORMIX Data Type	Sybase ASA Data Type
TEXT	TEXT / LONG VARCHAR
BYTE	BINARY / LONG BINARY / IMAGE / VARBINARY

Solution:

Very large character data types are not supported yet by the Sybase ASA database interface.

ODIASA031 - Cursors WITH HOLD

INFORMIX automatically closes opened cursors when a transaction ends unless the WITH HOLD option is used in the DECLARE instruction.

Sybase ASA does not close cursors when a transaction ends, as long as the global parameter close_on_endtrans is off.

Solution:

BDL cursors that are not declared "WITH HOLD" are automatically closed by the database interface when a COMMIT WORK or ROLLBACK WORK is performed by the BDL program.

ODIASA033 - Querying system catalog tables

As in INFORMIX, Sybase ASA provides system catalog tables (sysobjects,syscolumns,etc) in each database, but the table names and their structure are quite different.

Solution:

Warning: No automatic conversion of INFORMIX system tables is provided by the database interface.

ODIASA034 - Syntax of UPDATE statements

INFORMIX allows a specific syntax for UPDATE statements :

```
UPDATE table SET ( <col-list> ) = ( <val-list> )
```

or

```
UPDATE table SET table.* = myrecord.*
UPDATE table SET * = myrecord.*
```

Solution:

Static UPDATE statements using the above syntax are converted by the compiler to the standard form :

```
UPDATE table SET column=value [,...]
```

ODIASA036 - INTERVAL data type

INFORMIX's INTERVAL data type stores a value that represents a span of time. INTERVAL types are divided into two classes : **year-month intervals** and **day-time intervals.**

Sybase ASA does not provide a data type corresponding to the INFORMIX INTERVAL data type.

Solution:

Warning: The INTERVAL data type is not well supported because the database server has no equivalent native data type. However, you can store into and retrieve from CHAR columns BDL INTERVAL values.

ODIASA039 - Data storage concepts

An attempt should be made to preserve as much of the storage information as possible when converting from INFORMIX to Sybase ASA. Most important storage decisions made for INFORMIX database objects (like initial sizes and physical placement) can be reused in an Sybase ASA database.

Storage concepts are quite similar in INFORMIX and in Sybase ASA, but the names are different.

The following table compares INFORMIX storage concepts to Sybase ASA storage concepts :

INFORMIX	Sybase ASA
Physical units of storage	
The largest unit of physical disk space is a "**chunk**", which can be allocated either as a cooked file (I/O is controlled by the OS) or as raw device (=UNIX partition, I/O is controlled by the database engine). A "dbspace" uses at least one "chunk" for storage. You must add "chunks" to "dbspaces" in order to increase the size of the logical unit of storage.	A database is composed of **tablespace**. Each tablespace is composed of a '.db' file. In a database, there is one tablespace at the creation, but can hold more than one tablespace. The size of a tablespace is increased automatically.
A "**page**" is the smallest physical unit of disk storage that the engine uses to read from and write to databases. A "chunk" contains a certain number of "pages". The size of a "page" must be equal to the operating system's block size.	At the finest level of granularity, Sql Server Anywhere stores data in "**page**" which size can be defined at the creation time.
An "**extent**" consists of a collection of continuous "pages" that the engine uses to allocate both initial and subsequent storage space for database	Database files are extended by 32 pages at a time when the space is needed.

tables. When creating a table, you can specify the first extent size and the size of future extents with the EXTENT SIZE and NEXT EXTENT options. For a single table, "extents" can be located in different "chunks" of the same "dbspace".	
Logical units of storage	
A "**table**" is a logical unit of storage that contains rows of data values.	Same concept as INFORMIX.
A "**database**" is a logical unit of storage that contains table and index data. Each database also contains a system catalog that tracks information about database elements like tables, indexes, stored procedures, integrity constraints and user privileges.	Same concept as INFORMIX.
Database tables are created in a specific "**dbspace**", which defines a logical place to store data. If no dbspace is given when creating the table, INFORMIX defaults to the current database dbspace.	?
The total disk space allocated for a table is the "**tblspace**", which includes "pages" allocated for data, indexes, blobs, tracking page usage within table extents.	?
Other concepts	
When initializing an INFORMIX engine, a "**root dbspace**" is created to store information about all databases, including storage information (chunks used, other dbspaces, etc.).	?
The "**physical log**" is a set of continuous disk pages where the engine stores "before-images" of data that has been modified during processing. The "**logical log**" is a set of "**logical-log files**" used to record logical operations during on-line processing. All transaction information is stored in the logical log files if a database has been created with	Sybase ASA uses "**database log files**" to record SQL transactions.

transaction log.

INFORMIX combines "physical log" and "logical log" information when doing fast recovery. Saved "logical logs" can be used to restore a database from tape.

ODIASA040 - National characters data types

INFORMIX offers the NCHAR and NVARCHAR data types to store strings in a localized character set.

Solution:

Warning : National character data types are not supported yet.

ODIASA046 - The LOAD and UNLOAD instructions

INFORMIX provides two SQL instructions to export / import data from / into a database table: The UNLOAD instruction copies rows from a database table into an text file and the LOAD instruction inserts rows from an text file into a database table.

Warning : Sybase ASA has LOAD and UNLOAD instructions, but those commands are related to database backup and recovery. Do not confuse with INFORMIX commands.

Solution:

LOAD and UNLOAD instructions are supported.

Warning : The LOAD instruction does not work with tables using emulated SERIAL columns because the generated INSERT statement holds the "SERIAL" column which is actually a IDENTITY column in Sybase ASA. See the limitations of INSERT statements when using SERIALs.

Warning : In Sybase ASA, INFORMIX DATE data is stored in DATETIME columns, but DATETIME columns are similar to INFORMIX DATETIME YEAR TO FRACTION(3) columns. Therefore, when using LOAD and UNLOAD, those columns are converted to text data with the format "YYYY-MM-DD hh:mm:ss.fff".

Warning : In Sybase ASA, INFORMIX DATETIME data is stored in DATETIME columns, but DATETIME columns are similar to INFORMIX DATETIME YEAR TO FRACTION(3) columns. Therefore, when using LOAD and UNLOAD, those columns are converted to text data with the format "YYYY-MM-DD hh:mm:ss.fff".

Warning : When using an INFORMIX database, simple dates are unloaded with the DBDATE format (ex: "23/12/1998"). Therefore, unloading from an INFORMIX database for loading into a Sybase ASA database is not supported.

ODIASA047 - Case sensitivity

In INFORMIX, database object names like table and column names are not case sensitive :

```
CREATE TABLE Customer ( Custno INTEGER, ... )
SELECT CustNo FROM cuSTomer ...
```

In Sybase ASA, database object names **and character data** are case-insensitive by default :

```
CREATE TABLE Customer ( Custno INTEGER, CustName
CHAR(20) )
INSERT INTO CUSTOMER VALUES ( 1, 'TECHNOSOFT' )
SELECT CustNo FROM cuSTomer WHERE custname =
'techNOSoft'
```

Solution:

When you create a Sybase ASA database with **dbinit**, you can use the -c option to make the database case-sensitive.

ODIASA051 - Setup database statistics

INFORMIX provides a special instruction to compute database statistics in order to help the optimizer find the right query execution plan :

```
UPDATE STATISTICS ...
```

Sybase ASA offers a similar instruction, but it uses different clauses :

```
UPDATE STATISTICS ...
```

See Sybase ASA documentation for more details.

Solution:

Centralize the optimization instruction in a function.

ODIASA053 - The ALTER TABLE instruction

INFORMIX and MS Sybase ASA use different implementations of the ALTER TABLE instruction. For example, INFORMIX allows you to use multiple ADD clauses separated by comma. This is not supported by Sybase ASA :

INFORMIX :

```
ALTER TABLE customer ADD(col1 INTEGER), ADD(col2 CHAR(20))
```

Sybase ASA :

```
ALTER TABLE customer ADD col1 INTEGER, col2 CHAR(20)
```

Solution:

Warning: No automatic conversion is done by the database interface. There is even no real standard for this instruction (that is, no common syntax for all database servers). Read the SQL documentation and review the SQL scripts or the BDL programs in order to use the database server specific syntax for ALTER TABLE.

ODIASA054 - SQL Interruption

With Informix, it is possible to interrupt a long running query if the SQL INTERRUPT ON option is set by the Genero program. The database server returns SQLCODE -213, which can be trapped to detect a user interruption.

```
MAIN
  DEFINE n INTEGER
  DEFER INTERRUPT
  OPTIONS SQL INTERRUPT ON
  DATABASE test1
  WHENEVER ERROR CONTINUE
  -- Start long query (self join takes time)
  -- From now on, user can hit CTRL-C in TUI mode to stop the query
  SELECT COUNT(*) INTO n FROM customers a, customers b
       WHERE a.cust_id <> b.cust_id
  IF SQLCA.SQLCODE == -213 THEN
     DISPLAY "Statement was interrupted by user..."
     EXIT PROGRAM 1
  END IF
  WHENEVER ERROR STOP
  ...
END MAIN
```

Solution:

Warning: SQL Interruption is not supported with Sybase ASA.

ODIASA100 - Data type conversion table

INFORMIX Data Types	Sybase ASA Data Types
CHAR(n)	CHAR(n) (limit = 32767c!)
VARCHAR(n)	VARCHAR(n) (limit = 32767c!)
INTEGER	INTEGER
SMALLINT	SMALLINT
FLOAT[(n)]	FLOAT(n)
SMALLFLOAT	REAL
DECIMAL(p,s)	DECIMAL(p,s)! upper limit = 128 digits
MONEY(p,s)	DECIMAL(p,s)! upper limit = 128 digits
DATE	DATE (yyyy-mm-dd)
DATETIME HOUR TO FRACTION	TIME (hh:mm:ss.fff)
DATETIME q1 TO q2 (q2>FRACTION)	TIMESTAMP (yyyy-mm-dd hh:mm:ss.fff)
INTERVAL q1 TO q2	CHAR(n)

File Management Class

Summary:

- Basics
- Syntax
- Methods
- Examples
    - Example 1: Extracting the parts of a file name
    - Example 2: Browsing directories

See also: Built-in Functions

Basics

The Path class provides functions to manipulate files and directories on the machine where the BDL program executes.

This API is provided as a Dynamic C Extension library; it is part of the standard package.

To use this extension, you must import the **os** package in your program:

```
IMPORT os
```

Warning: In order to manipulate files, this API give you access to low-level system functions. Pay attention to operating system specific conventions like path separators. Some functions are OS specific, like rwx() which works only on UNIX systems.

Syntax

The **Path** class provides an interface to manipulate files and directories.

Syntax:

```
os.Path
```

Notes:

1. This class does not have to be instantiated; it provides class methods for the current program.

Methods:

Class Methods	
Name	**Description**
separator	Returns the character used to separate path segments.
pathseparator	Returns the character used in environment variables to separate path elements.
basename	Returns the last element of a path.
dirname	Returns all components of a path excluding the last one.
rootname	Returns the file path without the file extension of the last element of the file path.
join	Joins two path segments adding the platform-dependent separator.
pathtype	Checks whether a path is a relative path or an absolute path.
exists	Checks if a file exists.
extension	Returns the file extension.
readable	Checks if a file is readable.
writable	Checks if a file is writable.
executable	Checks if a file is executable.
isfile	Checks if a file is a regular file.
isdirectory	Checks if a file is a directory.
ishidden	Checks if a file is hidden.
islink	Checks if a file is a UNIX symbolic link.
isroot	Checks if a file is a root path.
type	Returns the file type as a string.
size	Returns the file size.
atime	Returns the time of the last file access.
chown	Changes the UNIX owner and group of a file.
uid	Returns the UNIX user id of the file.
gid	Returns the UNIX group id of the file.
rwx	Returns the UNIX permissions on the file.
chrwx	Changes the UNIX permissions of a file.
mtime	Returns the time of the last file modification.
homedir	Returns the path to the HOME directory of the current user.
rootdir	Returns the root directory of the current path.
dirfmask	Defines the filter mask for a diropen call
dirsort	Defines the sort criteria and sort order for a diropen call
diropen	Opens a directory and returns an integer handle to this directory.

dirclose	Closes the directory referenced by the directory handle.
dirnext	Reads the next entry of the directory referenced by the directory handle.
pwd	Returns the current working directory.
chdir	Changes the current working directory
volumes	Returns the list of available volumes.
chvolume	Changes the current working volume.
mkdir	Creates a new directory.
delete	Deletes a file or a directory.
rename	Renames a file or a directory.
copy	Copies a regular file.

---

os.Path.separator

Purpose:

Returns the character used to separate path segments.

Syntax:

```
CALL os.Path.separator() RETURNING separator STRING
```

Notes:

1. *separator* contains the separator.
2. On Unix, the separator is '/'
3. On Windows, the separator is '\'

---

os.Path.pathseparator

Purpose:

Returns the character used in environment variables to separate path elements.

Syntax:

```
CALL os.Path.pathseparator() RETURNING separator STRING
```

Notes:

1. *separator* contains the path separator.
2. On Unix, the separator is ':'

3. On Windows, the separator is ';'

Usage:

You typically use this method to build a path from two components.

---

os.Path.basename

Purpose:

This method returns the last element of a path.

Syntax:

```
CALL os.Path.basename(filename STRING) RETURNING basename STRING
```

Notes:

1. *filename* is the name of the file.
2. *basename* is the last element of the path.

Usage:

This method extracts the last component of a path. For example, if you pass `"/root/dir1/file.ext"` as the parameter, it will return `"file.ext"`.

See Example 1 for more examples.

---

os.Path.dirname

Purpose:

Returns all components of a path excluding the last one.

Syntax:

```
CALL os.Path.dirname(filename STRING) RETURNING dirname STRING
```

Notes:

1. *filename* is the name of the file.
2. *dirname* contains all the elements of the path excluding the last one.

Usage:

This method removes the last component of a path. For example, if you pass `"/root/dir1/file.ext"` as the parameter, it will return `"/root/dir1"`.

See Example 1 for more examples.

---

os.Path.rootname

Purpose:

Returns the file path without the file extension of the last element of the file path.

Syntax:

```
CALL os.Path.rootname(filename STRING) RETURNING rootname STRING
```

Notes:

1. *filename* is the file path.
2. *rootname* contains the file path without the file extension of the last element.

Usage:

This method removes the file extension from the path. For example, if you pass `"/root/dir1/file.ext"` as the parameter it will return `"/root/dir1/file"`.

See Example 1 for more examples.

---

os.Path.join

Purpose:

Joins two path segments adding the platform-dependent separator.

Syntax:

```
CALL os.Path.join(begin STRING, end STRING) RETURNING newpath STRING
```

Notes:

1. *begin* is the beginning path segment.
2. *end* is the ending path segment.
3. *newpath* contains the joined path segments.

Usage:

You typically use this method to construct a path with no system-specific code to use the correct path separator:

```
01   LET path = os.Path.join(os.Path.homedir(), name)
```

This method returns the ending path segment if it is an absolute path.

---

os.Path.pathtype

Purpose:

Checks if a path is a relative path or an absolute path.

Syntax:

```
CALL os.Path.pathtype(path STRING) RETURNING pathtype STRING
```

Notes:

1. *path* is the path to check.
2. *pathtype* can be "absolute" if the path is an absolute path, or "relative" if the path is a relative path.

---

os.Path.exists

Purpose:

Checks if a file exists.

Syntax:

```
CALL os.Path.exists(fname STRING) RETURNING result INTEGER
```

Notes:

1. *fname* is the file name.
2. *result* is TRUE if the file exists, FALSE otherwise.

---

os.Path.extension

Purpose:

Returns the file extension.

Syntax:

```
CALL os.Path.extension(fname STRING) RETURNING extension STRING
```

Notes:

1. *fname* is the file name.
2. *extension* is the string following the last dot found in *fname*.
3. If fname does not have an extension, the function returns an empty string.

---

os.Path.readable

Purpose:

Checks if a file is readable.

Syntax:

```
CALL os.Path.readable(fname STRING) RETURNING result INTEGER
```

Notes:

1. *fname* is the file name.
2. *result* is TRUE if the file is readable, FALSE otherwise.

---

os.Path.writable

Purpose:

Checks if a file is writable.

Syntax:

```
CALL os.Path.writable(fname STRING) RETURNING result INTEGER
```

Notes:

1. *fname* is the file name.

2. *result* is TRUE if the file is writable, FALSE otherwise.

---

os.Path.executable

Purpose:

Checks if a file is executable.

Syntax:

```
CALL os.Path.executable(fname STRING) RETURNING result INTEGER
```

Notes:

1. *fname* is the file name.
2. *result* is TRUE if the file is executable, FALSE otherwise.

---

os.Path.isfile

Purpose:

Checks if a file is a regular file.

Syntax:

```
CALL os.Path.isfile(fname STRING) RETURNING result INTEGER
```

Notes:

1. *fname* is the file name.
2. *result* is TRUE if the file is a regular file, FALSE otherwise.

---

os.Path.isdirectory

Purpose:

Checks if a file is a directory.

Syntax:

```
CALL os.Path.isdirectory(fname STRING) RETURNING result INTEGER
```

Notes:

1. *fname* is the file name.
2. *result* is TRUE if the file is a directory, FALSE otherwise.

---

os.Path.ishidden

Purpose:

Checks if a file is hidden.

Syntax:

```
CALL os.Path.ishidden(fname STRING) RETURNING result INTEGER
```

Notes:

1. *fname* is the file name.
2. *result* is TRUE if the file is hidden, FALSE otherwise.

---

os.Path.islink

Purpose:

Checks if a file is UNIX symbolic link.

Syntax:

```
CALL os.Path.islink(fname STRING) RETURNING result INTEGER
```

Notes:

1. *fname* is the file name.
2. *result* is TRUE if the file is a symbolic link, FALSE otherwise.

Warning: This method can only be used on UNIX!

---

os.Path.isroot

Purpose:

Checks if a file path is a root path.

Syntax:

```
CALL os.Path.isroot(path STRING) RETURNING result STRING
```

Notes:

1. *path* is the path to check.
2. *result* is TRUE if the path is a root path, FALSE otherwise.
3. On Unix the root path is '/'
4. On Windows the root path matches "[a-zA-Z]:\"

---

os.Path.type

Purpose:

Returns the file type as a string

Syntax:

```
CALL os.Path.type(fname STRING) RETURNING ftype STRING
```

Notes:

1. *fname* is the file name.
2. *ftype* can be one of:
    1. file : the file is a regular file
    2. directory : the file is a directory
    3. socket : the file is a socket
    4. fifo : the file is a fifo
    5. block : the file is a block device
    6. char : the file is a character device

Warning: On UNIX, this method follows symbolic links. You must use the islink() method to identify symbolic links.

---

os.Path.size

Purpose:

Returns the size of a file.

Syntax:

```
CALL os.Path.size(fname STRING) RETURNING size INTEGER
```

Notes:

1. *fname* is the file name.
2. *size* is the file size

---

os.Path.atime

Purpose:

Returns the time of the last file access.

Syntax:

```
CALL os.Path.atime(fname STRING) RETURNING atime STRING
```

Notes:

1. *fname* is the name of the file.
2. *atime* is a string containing the last access time for the specified file in the standard format 'YYYY-MM-DD HH:MM:SS'.
3. If the function failed, it returns NULL.

---

os.Path.mtime

Purpose:

Returns the time of the last file modification.

Syntax:

```
CALL os.Path.mtime(fname STRING) RETURNING mtime STRING
```

Notes:

1. *fname* is the name of the file.
2. *mtime* is a string containing the last modification time for the specified file in the standard format 'YYYY-MM-DD HH:MM:SS'.
3. If the function failed, it returns NULL.

---

os.Path.rwx

Purpose:

Returns the UNIX file permissions of a file.

Syntax:

```
CALL os.Path.rwx(fname STRING) RETURNING mode INTEGER
```

Notes:

1. *fname* is the name of the file.
2. *mode* is the combination of permissions for user, group and other.
3. Function returns -1 if it fails to get the permissions.

Warning: This method can only be used on UNIX!

Usage:

The *mode* is returned as a decimal value which is the combination of read, write and execution bits for the user, group and other part of the UNIX file permission. For example, if a file has the `-rwxr-xr-x` permissions, you get ((4+2+1) * 64 + (4+1) * 8) + (4+1)) = 493 from this method.

---

os.Path.chrwx

Purpose:

Changes the UNIX permissions of a file.

Syntax:

```
CALL os.Path.chrwx(fname STRING, mode INTEGER) RETURNING res INTEGER
```

Notes:

1. *fname* is the name of the file.
2. *mode* is the UNIX permission combination in decimal (not octal!).
3. Function returns TRUE on success, FALSE otherwise.

Warning: This method can only be used on UNIX!

Usage:

The *mode* must be a decimal value which is the combination of read, write and execution bits for the user, group and other part of the UNIX file permission. Make sure to pass the *mode* as the decimal version of permissions, not as octal (the chrwx UNIX command takes an octal value as parameter). For example, to set `-rw-r--r--` permissions, you must pass (((4+2) * 64) + (4 * 8) + 4) = 420 to this method.

---

os.Path.chown

Purpose:

Changes the UNIX owner and group of a file.

Syntax:

```
CALL os.Path.chown(fname STRING, uid INT, gui INT) RETURNING res
INTEGER
```

Notes:

1. *fname* is the name of the file.
2. *uid* is the user id.
3. gui is the group id.
4. Function returns TRUE on success, FALSE otherwise.

Warning: This method can only be used on UNIX!

---

os.Path.uid

Purpose:

Returns the UNIX user id of a file.

Syntax:

```
CALL os.Path.uid(fname STRING) RETURNING id INTEGER
```

Notes:

1. *fname* is the name of the file.
2. *id* is the user id.
3. Function returns -1 if it fails to get the user id.

Warning: This method can only be used on UNIX!

---

os.Path.gid

Purpose:

Returns the UNIX group id of a file.

Syntax:

```
CALL os.Path.gid(fname STRING) RETURNING id INTEGER
```

Notes:

1. *fname* is the name of the file.
2. *id* is the group id.
3. Function returns -1 if it fails to get the user id.

Warning: This method can only be used on UNIX!

---

os.Path.homedir

Purpose:

Returns the path to the HOME directory of the current user.

Syntax:

```
CALL os.Path.homedir() RETURNING homedir STRING
```

Notes:

1. *homedir* Path to the HOME directory of the user.

---

os.Path.rootdir

Purpose:

Returns the root directory of the current working path.

Syntax:

```
CALL os.Path.rootdir() RETURNING rootdir STRING
```

Notes:

1. *rootdir* is the root directory of the current working path.
2. On Unix, it always returns '/'
3. On Windows it returns the current working drive as "[a-zA-Z]:\"

---

os.Path.dirfmask

Purpose:

Defines a filter mask for diropen.

Syntax:

```
CALL os.Path.dirfmask(mask INTEGER)
```

Notes:

1. *mask* defines the filter mask (see below for possible values).

Usage:

When you call this function, you define the filter mask for any subsequent diropen call.

By default, all kinds of directory entries are selected by the diropen function. You can restrict the number of entries by using a filter mask.

The parameter of the `dirfmask` function must be a combination of the following bits:

- `0x01` = Exclude hidden files (.*)
- `0x02` = Exclude directories
- `0x04` = Exclude symbolic links
- `0x08` = Exclude regular files

For example, to retrieve only regular files, you must call:

```
01 CALL os.Path.dirfmask( 1 + 2 + 4 )
```

---

os.Path.dirsort

Purpose:

Defines the sort criteria and sort order for diropen.

Syntax:

```
CALL os.Path.dirsort(criteria STRING, order INTEGER)
```

Notes:

1. *criterie* is the sort criteria (see below for possible values).
2. *order* defines ascending (1) or descending (-1) order.

Usage:

When you call this function, you define the sort criteria and sort order for any subsequent diropen call.

The *criteria* parameter must be one of the following strings:

- `"undefined"` = No sort. This is the default. Entries are read as returned by the OS functions.
- `"name"` = Sort by file name.
- `"size"` = Sort by file size.
- `"type"` = Sort by file type (directory, link, regular file).
- `"atime"` = Sort by access time.
- `"ctime"` = Sort by modification time.
- `"extension"` = Sort by file extension.

When sorting by name, directory entries will be ordered according to the current locale.

When sorting by any criteria other than the file name, entries having the same value for the given criteria are ordered by name in ascending order.

---

os.Path.diropen

Purpose:

Opens a directory and returns an integer handle to this directory.

Syntax:

```
CALL os.Path.diropen(dname STRING) RETURNING dirhandle INTEGER
```

Notes:

1. *dname* is the name of the directory.
2. *dirhandle* is the directory handle. A dirhandle value of 0 indicates a failure when opening the directory.

Usage:

This function creates a list of directory

See also: dirfmask, dirsort

---

os.Path.dirclose

Purpose:

Closes the directory referenced by the directory handle *dirhandle*.

Syntax:

```
CALL os.Path.dirclose(dirhandle INTEGER)
```

Notes:

1. *dirhandle* is the directory handle of the directory to close.

---

os.Path.dirnext

Purpose:

Reads the next entry in the directory.

Syntax:

```
CALL os.Path.dirnext(dirhandle INTEGER) RETURNING direntry STRING
```

Notes:

1. *dirhandle* is the directory handle of the directory to read.
2. *direntry* is the name of the entry read or NULL if all entries have been read.

---

os.Path.pwd

Purpose:

Returns the current working directory.

Syntax:

```
CALL os.Path.pwd() RETURNING cwd STRING
```

Notes:

1. *cwd* is the current working directory.

---

os.Path.chdir

Purpose:

Changes the current working directory.

Syntax:

```
CALL os.Path.chdir(newdir STRING) RETURNING result INTEGER
```

Notes:

1. *newdir* is the directory to select.
2. *result* is TRUE if the current directory could be successfully selected.

---

os.Path.volumes

Purpose:

Returns the available volumes.

Syntax:

```
CALL os.Path.volumes() RETURNING volumes STRING
```

Notes:

1. *volumes* contains the list of all available volumes separated by "|".

---

os.Path.chvolume

Purpose:

Changes the current working volume.

Syntax:

```
CALL os.Path.chvolume(newvolume STRING) RETURNING result INTEGER
```

Notes:

1. *newvolume* is the volume to select as the new current working volume.
2. *result* is TRUE if the current working volume could be successfully changed.
3. Sample : CALL os.Path.chvolume("C:\\") RETURNING result

---

os.Path.mkdir

Purpose:

Creates a new directory.

Syntax:

```
CALL os.Path.mkdir(dname STRING) RETURNING result INTEGER
```

Notes:

1. *dname* is the name of the directory to create
2. *result* is TRUE if the directory has been successfully created, FALSE otherwise.

---

os.Path.delete

Purpose:

Deletes a file or a directory.

Syntax:

```
CALL os.Path.delete(dname STRING) RETURNING result INTEGER
```

Notes:

1. *fname* is the name of the file or directory to delete

2. *result* is TRUE if the file or directory has been successfully deleted, FALSE otherwise.
3. A directory can only be deleted if it is empty.

---

os.Path.rename

Purpose:

Renames a file or a directory.

Syntax:

```
CALL os.Path.rename(oldname STRING, newname STRING) RETURNING result
INTEGER
```

Notes:

1. *oldname* is the current name of the file or directory to be renamed.
2. *newname* is the new name to assign to the file or directory.
3. *result* is TRUE if the file or directory has been successfully renamed, FALSE otherwise.

---

os.Path.copy

Purpose:

Copies a regular file or.

Syntax:

```
CALL os.Path.copy(source STRING, dest STRING) RETURNING result INTEGER
```

Notes:

1. *source* is the name of the file to copy.
2. *dest* is the destination name of the copied file.
3. *result* is TRUE if the file has been successfully copied, FALSE otherwise.

Examples:

Example 1: Extracting the parts of a file name

This program uses the file functions to extract the directory name, the base name, the root name, and the file extension:

```
01 IMPORT os
02 MAIN
03   DISPLAY "Dir name   = ", os.Path.dirname(arg_val(1))
04   DISPLAY "Base name  = ", os.Path.basename(arg_val(1))
05   DISPLAY "Root name  = ", os.Path.rootname(arg_val(1))
06   DISPLAY "Extension  = ", os.Path.extension(arg_val(1))
07 END MAIN
```

Example results:

Path	os.Path.dirname	os.Path.basename	os.Path.rootname	os.Path.extension
.	.	.		NULL
..	.	..	.	NULL
/	/	/	/	NULL
/usr/lib	/usr	lib	/usr/lib	NULL
/usr/	/	usr	/usr/	NULL
usr	.	usr	usr	NULL
file.xx	.	file.xx	file	xx
/tmp.yy/file.xx	/tmp.yy	file.xx	/tmp.yy/file	xx
/tmp.yy/file.xx.yy	/tmp.yy	file.xx.yy	/tmp.yy/file.xx	yy
/tmp.yy/	/	tmp.yy	/tmp.yy/	NULL
/tmp.yy/.	/tmp.yy	.	/tmp.yy/	NULL

Warning: The above examples use Unix file names. On Windows the result would be different, as the file name separator is '\'.

Example 2: Browsing directories.

This program takes a directory path as an argument and scans the content recursively:

```
01 IMPORT os
02 MAIN
03   CALL showDir(arg_val(1))
04 END MAIN
05 FUNCTION showDir(path)
06   DEFINE path STRING
07   DEFINE child STRING
08   DEFINE h INTEGER
09   IF NOT os.Path.exists(path) THEN
10      RETURN
11   END IF
12   IF NOT os.Path.isdirectory(path) THEN
13      DISPLAY " ", os.Path.basename(path)
```

```
14      RETURN
15   END IF
16   DISPLAY "[", path, "]"
17   CALL os.Path.dirsort("name", 1)
18   LET h = os.Path.diropen(path)
19   WHILE h > 0
20      LET child = os.Path.dirnext(h)
21      IF child IS NULL THEN EXIT WHILE END IF
22      IF child == "." OR child == ".." THEN CONTINUE WHILE END IF
23      CALL showDir( os.Path.join( path, child ) )
24   END WHILE
25   CALL os.Path.dirclose(h)
26 END FUNCTION
```

Mathematical functions Class

Summary:

- Basics
- Syntax
- Methods

See also: Built-in Functions

Basics

The Math class provides mathematical functions.

This API is provided as a Dynamic C Extension library; it is part of the standard package.

To use this extension, you must import the **util** package in your program:

```
IMPORT util
```

Syntax

The **Math** class is a provides an interface for mathematical functions.

Syntax:

```
util.Math
```

Notes:

1. This class does not have to be instantiated; it provides class methods for the current program.

Methods:

Class Methods	
Name	**Description**
sqrt	This function computes the square root of its argument.
pow	This function computes the value of x raised to the power y.

exp	This function computes the base- e exponential of x.
srand	This function intializes the pseudo-random number generator.
rand	This function returns a pseudo-random number.
sin	This function computes the sine of their argument x, measured in radians.
cos	This function computes the cosine of their argument x, measured in radians.
tan	This function computes the tangent of their argument x, measured in radians.
asin	This function computes the arc sine of their argument x, measured in radians.
acos	This function computes the arc cosine of their argument x, measured in radians.
atan	This function computes the arc tangent of their argument x, measured in radians.
log	This function computes the natural logarithm of the argument x.
toDegrees	Converts an angle measured in radians to an equivalent angle measured in degrees.
toRadians	Converts an angle measured in degrees to an equivalent angle measured in radians.
pi	This function returns the FLOAT value of PI.

---

util.Math.sqrt

Purpose:

Returns the square root of the argument provided.

Syntax:

```
CALL util.Math.sqrt(FLOAT f) RETURNING rv FLOAT
```

Notes:

1. Returns NULL if the argument provided is invalid.

---

util.Math.pow

Purpose:

This function computes the value of x raised to the power y.

Syntax:

```
CALL util.Math.pow(FLOAT x, FLOAT y) RETURNING rv FLOAT
```

Notes:

1. Returns NULL if one of the argument provided is invalid.
2. If x is negative, the caller should ensure that y is an integer value.

---

util.Math.exp

Purpose:

This function computes the base- e exponential of x.

Syntax:

```
CALL util.Math.exp(FLOAT x) RETURNING rv FLOAT
```

Notes:

1. Returns NULL if the argument provided on error.

---

util.Math.srand

Purpose:

This function initializes the pseudo-random numbers generator.

Syntax:

```
CALL util.Math.srand()
```

---

util.Math.rand

Purpose:

This function returns a pseudo-random number between 0 and x

Syntax:

```
CALL util.Math.rand(x INTEGER) RETURNING rv INTEGER
```

---

util.Math.sin

Purpose:

This function computes the sine of the argument x, measured in radians.

Syntax:

```
CALL util.Math.sin(FLOAT f) RETURNING rv FLOAT
```

Notes:

1. Returns NULL if the argument provided is invalid.

---

util.Math.cos

Purpose:

This function computes the cosine of the argument x, measured in radians.

Syntax:

```
CALL util.Math.cos(FLOAT f) RETURNING rv FLOAT
```

Notes:

1. Returns NULL if the argument provided is invalid.

---

util.Math.tan

Purpose:

This function computes the tangent of the argument x, measured in radians.

Syntax:

```
CALL util.Math.tan(FLOAT f) RETURNING rv FLOAT
```

Notes:

1. Returns NULL if the argument provided is invalid.

---

util.Math.asin

Purpose:

This function computes the arc sine of the argument x, measured in radians.

Syntax:

```
CALL util.Math.asin(FLOAT f) RETURNING rv FLOAT
```

Notes:

1. Returns NULL if the argument provided is invalid.

---

util.Math.acos

Purpose:

This function computes the arc cosine of the argument x, measured in radians.

Syntax:

```
CALL util.Math.acos(FLOAT f) RETURNING rv FLOAT
```

Notes:

1. Returns NULL if the argument provided is invalid.

---

util.Math.atan

Purpose:

This function computes the arc tangent of the argument x, measured in radians.

Syntax:

```
CALL util.Math.atan(FLOAT f) RETURNING rv FLOAT
```

Notes:

1. Returns NULL if the argument provided is invalid.

---

util.Math.log

Purpose:

This function computes the natural logarithm of the argument x.

Syntax:

```
CALL util.Math.log(FLOAT f) RETURNING rv FLOAT
```

Notes:

1. Returns NULL if the argument provided is invalid.

---

util.Math.toDegrees

Purpose:

Converts an angle measured in radians to an approximately equivalent angle measured in degrees.

Syntax:

```
CALL util.Math.toDegrees(FLOAT f) RETURNING rv FLOAT
```

---

util.Math.toRadians

Purpose:

Converts an angle measured in degrees to an approximately equivalent angle measured in radians.

Syntax:

```
CALL util.Math.toRadians(FLOAT f) RETURNING rv FLOAT
```

---

util.Math.pi

Purpose:

This function returns the FLOAT value of PI.

Syntax:

```
CALL util.Math.pi() RETURNING rv FLOAT
```

Implementing C-Extensions

Summary:

- Basics
- Creating C-Extensions
  - Creating ESQL/C extensions
    - ESQL/C Extensions with non-Informix databases
    - ESQL/C Extensions with Informix
- C interface file
- Linking programs using C-Extensions
- Loading C-Extensions
  - Using the IMPORT instruction
  - Using the default extension name
  - Using the -e fglrun option
- Stack Functions
- C Data Types and Structures
- Available Macros
  - VARCHAR Macros
  - DECIMAL Macros
  - DATETIME/INTERVAL Macros
  - DATETIME/INTERVAL Constants
- Calling C functions from BDL
- Calling BDL functions from C
- Sharing global variables
- Example
- C API Functions
- Formatting directives

See also: Programs, Installation and Setup.

Basics

With C-Extensions, you can integrate your own C libraries in the runtime system, to call C function from the BDL code. This feature allows you to extend the language with custom libraries, or existing standard libraries, just by writing some 'wrapper functions' to interface with BDL.

Warning: Using C functions in your applications can cause problems when you port the application to another platform. For example, you can expect problems when porting an application from UNIX to Windows and vice versa. Portability problems can also occur when using incompatible C data types or when using platform-specific system calls.

C-Extensions are implemented with shared libraries. Using shared libraries avoids the need to re-link the fglrun program and simplifies deployment.

Function parameters and returned values are passed/returned on the legal BDL stack, using pop/push functions. Be sure to pop and push the exact number of parameters/returns expected by the caller; otherwise, a fatal stack error will be raised at runtime.

Warning: In earlier versions of Genero, static C extensions could define global variables sharing between the runtime system and the extension; This was done by using the "userData" structure in the extension interface file. With Dynamic C extensions, global variables cannot be shared any longer. You must use functions to pass global variable values. For an example, see Sharing global variables.

Creating C-Extensions

Custom C Extensions must be provided to the runtime system as Shared Objects (**.so**) on UNIX, and as Dynamically Loadable Libraries (**.DLL**) on Windows.

In order to create a C-Extension, you must:

1. Define the list of user functions in the C interface file.
2. Compile the C interface file and the C modules with the position-independent code option, by including the **fglExt.h** header file.
3. Create the shared library with the compiled C interface file and C modules by linking with the **libfgl** library.

In your C source files, you must include the **fglExt.h** header file in the following way:

```
01 #include <f2c/fglExt.h>
```

Warning: When migrating from Informix 4GL, it is possible that existing C extension sources include Informix specific headers like sqlhdr.h or decimal.h. You must include Informix specific header files before the fglExt.h header file, in order to let fglExt.h detect that typedefs such as dec_t or dtime_t are already defined by Informix headers. If you include Informix headers after fglExt.h, you will get a compilation error. As fglExt.h defines all Informix-like typedef structures, you can remove the inclusion of Informix specific header files.

Your C functions must be known by the runtime system. To do so, each C extension library must publish its functions in a **UsrFunction array**, which is read by the runtime system when the module is loaded. The UsrFunction array describes the user functions by specifying the name of the function, the C function pointer, the number of parameters and the number of returned values. You typically define the UsrFunction array the in the C interface file.

After compiling the C sources, you must link them together with the **libfgl** runtime system library. See below for examples.

Warning: Carefully read the man page of the ld dynamic loader, and any documentation of your operating system related to shared libraries. Some platforms require specific configuration and command line options when linking a shared library, or when linking a program using a shared library (+s option on HP for example).

Linux command-line example:

```
gcc -c -I $FGLDIR/include -fPIC myext.c
gcc -c -I $FGLDIR/include -fPIC cinterf.c
gcc -shared -o myext.so myext.o cinterf.o -L$FGLDIR/lib -lfgl
```

Windows command-line example using Visual C 7.1:

```
cl /DBUILDDLL /I%FGLDIR%/include /c myext.c
cl /DBUILDDLL /I%FGLDIR%/include /c cintref.c
link /dll /out:myext.dll myext.obj cinterf.obj %FGLDIR%\lib\libfgl.lib
```

Windows command-line example using Visual C 8.0 (you must create a manifest file for the DLL!):

```
cl /DBUILDDLL /I%FGLDIR%/include /c myext.c
cl /DBUILDDLL /I%FGLDIR%/include /c cintref.c
link /dll /manifest /out:myext.dll myext.obj cinterf.obj
%FGLDIR%\lib\libfgl.lib
mt -manifest myext.dll.manifest -outputresource:myext.dll
```

Creating ESQL/C Extensions

You can create user extension libraries from ESQL/C sources, as long as you have an ESQL/C compiler which is compatible with your Genero runtime system. In order to create these user extensions, you must first compile the **.ec** sources to object files by including the **fglExt.h** header file. Then you must create the shared library by linking with additional SQL libraries to resolve the functions used in the **.ec** source to execute SQL statements.

ESQL/C extensions with non-Informix databases

If you need to use ESQL/C extensions with database clients such as Oracle, SQL Server or Genero DB, you must use the **fesqlc** compiler, and link the objects with the FESQLC libraries in order to use the same SQL API as the FGL runtime system database interface.

For more details about Genero's ESQL/C compiler and .ec extension creation, see FESQLC page.

ESQL/C extensions with Informix

You can compile .ec extensions with the native Informix esql compiler, or with the fesqlc compiler provided by Genero. This section describes how to use the Informix esql

compiler. For more details about creating an .ec extension library with Genero's ESQL/C compiler, see FESQLC page.

The following example shows how to compile and link an extension library with Informix **esql** compiler:

Linux command-line example:

```
esql -c -I$FGLDIR/include myext.ec
gcc -c -I$FGLDIR/include -fPIC cinterf.c
gcc -shared -o myext.so myext.o cinterf.o -L$FGLDIR/lib -lfgl \
    -L$INFORMIXDIR/lib -L$INFORMIXDIR/lib/esql `esql -libs`
```

Windows command-line example (using VC 7.1):

```
esql -c myext.ec -I%FGLDIR%/include
cl /DBUILDDLL /I%FGLDIR%/include /c cintref.c
esql -target:dll -o myext.dll myext.obj cinterf.obj
%FGLDIR%\lib\libfgl.lib
```

When using Informix **esql**, you link the extension library with *Informix client libraries*. These libraries will be shared by the extension module and the Informix database driver loaded by the runtime system. Since both the extension functions and the runtime database driver use the same functions to execute SQL queries, you can share the current SQL connection opened in the Genero program to execute SQL queries in the extension functions.

C Interface File

To make your C functions visible to the runtime system, you must define all the functions in the **C Interface File**. This is a C source file that defines the **usrFunctions** array. This array defines C functions that can be called from BDL. The last record of each array must be a line with all the elements set to 0, to define the end of the list.

The first member of a **usrFunctions** element is the BDL name of the function, provided as a character string. The second member is the C function symbol. Therefore, you typically do a forward declaration of the C functions before the **usrFunctions** array initializer. The third member is the number of BDL parameters passed thru the stack to the function. The last member is the number of values returned by the function; you can use -1 to specify a variable number of arguments.

Example:

```
01 #include <f2c/fglExt.h>
02
03 int c_log(int);
04 int c_cos(int);
05 int c_sin(int);
```

```
06
07 UsrFunction usrFunctions[]={
08   {"log",c_log,1,1},
09   {"cos",c_cos,1,1},
10   {"sin",c_sin,1,1},
11   {0,0,0,0}
12 };
```

Linking programs using C-Extensions

When creating a 42r program or 42x library, the FGL linker needs to resolve all function names, including C extension functions. Thus, if extension modules are not specified explicitly in the source files with the IMPORT directive, you must give the extension modules with the **-e** option in the command line:

```
fgllink -e myext1,myext2,myext3 myprog.42r moduleA.42m moduleB.42m ...
```

The **-e** option is not needed when using the default userextension module.

Loading C-Extensions

The runtime system can load several C-Extensions libraries, allowing you to properly split your libraries by defining each group of functions in separate C interface files.

Directories are searched for the C-Extensions libraries according to the FGLLDPATH environment variable rules: If the module cannot be found in the directory where the .42r program resides, FGLLDPATH is scanned. If the module could not be found with FGLLDPATH, then $FGLDIR/lib is searched. If the module is still not found, the current directory is searched.

There are three ways to bind a C Extension with the runtime system:

1. Using the `IMPORT` instruction in sources.
2. Using the default C Extension name.
3. Using the -e option of fglrun.

Using the IMPORT instruction

The `IMPORT` instruction allows you to declare an external module in a .4gl source file. It must appear at the beginning of the source file.

The compiler and the runtime system automatically know which C extensions must be loaded, based on the `IMPORT` instruction:

```
01 IMPORT mylib1
```

```
02 MAIN
03   CALL myfunc1("Hello World")  -- defined in mylib1
04 END MAIN
```

When the IMPORT instruction is used, no other action has to be taken at runtime: The module name is stored in the **42m** pcode and is automatically loaded when needed.

For more details, see Importing modules.

Using the default C Extension name

Normally, all FGL modules using a function from a C extension should now use the IMPORT instruction. This could be a major change in your sources.

To simplify migration of existing C extensions, the runtime system loads by default a module with the name **userextension.** Create this shared library with your existing C extensions, and the runtime system will load it automatically if it is in the directories specified by FGLLDPATH.

Using the -e fglrun option

In some cases you need several C extension libraries, which are used by different group of programs, so you can't use the default **userextension** solution. However, you don't want to review all your sources in order to use the IMPORT instruction.

You can specify the C Extensions to be loaded by using the **-e** option of fglrun. The -e option takes a comma-separated list of module names, and can be specified multiple times in the command line. The next example loads five extension modules:

```
fglrun -e myext1,myext2,myext3 -e myext4,myext5 myprog.42r
```

By using the **-e** option, the runtime system loads the modules specified in the command line instead of loading the default **userextension** module.

Stack Functions

To pass values between a C function and a program, the C function and the runtime system use the BDL stack. The **int** parameter of the C function defines the number of input parameters passed on the stack, and the function must return an **int** value defining the number of values returned on the stack. The parameters passed to the C function must be popped from the stack at the beginning of the C function, and the return values expected by the 4gl call must be pushed on the stack before leaving the C function. If you don't pop / push the specified number of parameters / return values, you corrupt the stack and get a fatal error.

The runtime system library includes a set of functions to retrieve the values passed as parameters on the stack. The following table shows the library functions provided to pop values from the stack into C buffers:

Function	BDL Type	Notes
void popdate(int4 *dst);	DATE	4-byte integer value corresponding to days since 12/31/1899.
void popint(mint *dst);	INTEGER	System dependent integer value
void popshort(int2 *dst);	SMALLINT	2-byte integer value
void poplong(int4 *dst);	INTEGER	4-byte integer value
void popflo(float *dst);	SMALLFLOAT	4-byte floating point value
void popdub(double *dst);	FLOAT	8-byte floating point value
void popdec(dec_t *dst);	DECIMAL	See structure definition in $FGLDIR/include/f2c headers
void popquote(char *dst, int len);	CHAR(n)	len = strlen(val)+1 (for the '/0')
void popvchar(char *dst, int len);	VARCHAR(n)	len = strlen(val)+1 (for the '/0')
void popdtime(dtime_t *dst, int size);	DATETIME	See structure definition in $FGLDIR/include/f2c headers
void popinv(intrvl_t* dst, int size);	INTERVAL	See structure definition in $FGLDIR/include/f2c headers
void poplocator(loc_t **dst);	BYTE, TEXT	See structure definition in $FGLDIR/include/f2c headers Warning: this function pops the pointer of a loc_t object!

When using a pop function, the value is copied from the stack to the local C variable and the value is removed from the stack.

In BDL, Strings (CHAR, VARCHAR) are not terminated by '\0'. Therefore, the C variable must have one additional character to store the '\0'. For example, the equivalent of a VARCHAR(100) in BDL is a char [101] in C.

To return a value from the C function, you must use one of the following functions provided in the runtime system library:

Function	BDL Type	Notes
void pushdate(const int4 val);	DATE	4-byte integer value corresponding to days since 12/31/1899.
void pushdec(const dec_t* val, const unsigned decp);	DECIMAL	See structure definition in $FGLDIR/include/f2c headers
void pushint(const mint val);	INTEGER	System dependent integer value
void pushlong(const int4 val);	INTEGER	4-byte integer value

void pushshort(const int2 val);	SMALLINT	2-byte integer value
void pushflo(const float* val);	SMALLFLOAT	4-byte floating point value. Warning: function takes a pointer!
void pushdub(const double* val);	FLOAT	8-byte floating point value. Warning: function takes a pointer!
void pushquote(const char *val, int l);	CHAR(n)	len = strlen(val) (without '\0')
void pushvchar(const char *val, int l);	VARCHAR(n)	len = strlen(val) (without '\0')
void pushdtime(const dtime_t *val);	DATETIME	See structure definition in $FGLDIR/include/f2c headers
void pushinv(const intrvl_t *val);	INTERVAL	See structure definition in $FGLDIR/include/f2c headers

When using a push function, the value of the C variable is copied at the top of the stack.

C Data Types and Structures

The fglExt.h header file defines the following C types:

Type name	Description
int4	signed integer with a size of 4 bytes
uint4	unsigned integer with a size of 4 bytes
int2	signed integer with a size of 2 bytes
uint2	unsigned integer with a size of 2 bytes
int1	signed integer with a size of 1 byte
uint1	unsigned integer with a size of 1 byte
mint	signed machine-dependent C int
muint	unsigned machine-dependent C int
mlong	signed machine-dependent C long
mulong	unsigned machine-dependent C long
dec_t	DECIMAL data type structure
dtime_t	DATETIME data type structure
intrvl_t	INTERVAL data type structure
loc_t	TEXT / BYTE Locator structure

Basic data types

Basic data types such as **int4** and **int2** are provided for Informix compatibility.

You can use these types to define variables that must hold SMALLINT (int2), INTEGER (int4) and DATE (int4) values.

DATE

No specific typedef exists for DATEs; you can use the **int4** type to store a DATE value.

DECIMAL/MONEY

The **dec_t** structure is provided to hold DECIMAL and MONEY values.

The internals of dec_t structure can be ignored during C extension programming, because decimal API functions are provided to manipulate any aspects of a decimal.

DATETIME

The **dtime_t** structure holds a DATETIME value.

Before manipulating a dtime_t, you must initialize its qualifier qt_qual, by using the TU_DTENCODE macro, as in the following example:

```
01 dtime_t dt;
02 dt.dt_qual = TU_DTENCODE(TU_YEAR, TU_SECOND);
03 dtcvasc( "2004-02-12 12:34:56", &dt );
```

INTERVAL

The **intrvl_t** structure holds an INTERVAL value.

Before manipulating a intrvl_t, you must initialize its qualifier in_qual, by using the TU_IENCODE macro, as in the following example:

```
01 intrvl_t in;
02 in.in_qual = TU_IENCODE(5, TU_YEAR, TU_MONTH);
03 incvasc( "65234-02", &in );
```

TEXT/BYTE Locator

The **loc_t** structure is used to declare host variables for a TEXT/BYTE values (simple large objects). Because the potential size of the data can be quite large, this is a locator structure that contains information about the size and location of the TEXT/BYTE data, rather than containing the actual data.

The fields of the loc_t structure are:

Field name	Data Type	Description
loc_indicator	int4	Null indicator; a value of -1 indicates a null TEXT/BYTE value. Your program can set the field to indicate the insertion of a null value. FESQLC or ESQL/C libraries set the value for selects and fetches.

loc_type	int4	Data type - SQLTEXT (for TEXT values) or SQLBYTES (for BYTE values).
loc_size	int4	Size of the TEXT/BYTE value in bytes; your program sets the size of the large object for insertions. FESQLC or ESQL/C libraries set the size for selects and fetches.
loc_loctype	int2	Location - LOCMEMORY (in memory) or LOCFNAME (in a named file). Set loc_loctype after you declare the locator variable and before this declared variable receives the large object value.
loc_buffer	char *	If loc_loctype is LOCMEMORY, this is the location of the TEXT/BYTE value; your program must allocate space for the buffer and store its address here.
loc_bufsize	int4	IF loc_loctype is LOCMEMORY, this is the size of the buffer loc_buffer; If you set loc_bufsize to -1, FESQLC or ESQL/C libraries will allocate the memory buffer for selects and fetches. Otherwise, it is assumed that your program will handle memory allocation and de-allocation.
loc_fname	char *	IF loc_loc_type is LOCFNAME, tthis is the address of the pathname string that contains the file.

Example:

```
01 loc_t *pb1
02 double ratio;
03 char *source = NULL, *psource = NULL;
04 int size;
05
06 if (pb1->loc_loctype == LOCMEMORY) {
07     psource = pb1->loc_buffer;
08     size = pb1->loc_size;
09 } else if (pb1->loc_loctype == LOCFNAME) {
10     int fd;
11     struct stat st;
12     fd = open(pb1->loc_fname, O_RDONLY);
13     fstat(fd, &st);
14     size = st.st_size;
15     psource = source = (char *) malloc(size);
16     read(fd, source, size);
17     close(fd);
18 }
```

Available Macros

Varchar type related macros

The following macros allow you to obtain the size information stored by the database server for a VARCHAR column:

Macro	Description
MAXVCLEN (255)	Returns maximum number of characters allowed in a VARCHAR column
VCMIN(size)	Returns minimum number of characters that you can store in a VARCHAR column
VCLENGTH(len)	Returns length of the host variable
VCMAX(size)	Returns maximum number of characters that you can store in a VARCHAR column
VCSIZ(max, min)	Returns encoded size for the VARCHAR column

Decimal type related macros

Decimals are defined by an encoded **length** - the total number of significant digits (**precision**), and the significant digits to the right of the decimal (**scale**). The following macros handle decimal length:

Macro	Description
DECLEN(m,n)	Calculates the minimum number of bytes necessary to hold a decimal (m = precision, n = scale)
DECLENGTH(length)	Calculates the minimum number of bytes necessary to hold a decimal, given the encoded length
DECPREC(size)	Calculates a default precision, given the size (the number of bytes used to store a number)
PRECTOT(length)	Returns the precision from an encoded length
PRECDEC(length)	Returns the scale from an encoded length
PRECMAKE(p,s)	Returns the encoded decimal length from a precision and scale

Datetime/Interval related macros

Datetime and Interval need qualifiers (ex: YEAR TO MONTH) to complete the type definition. The following macros can be used to manage those qualifiers and set the qt_qual or in_qual members of dtime_t and intrvl_t structures.

Macro	Description
TU_YEAR	Defines the YEAR qualifier
TU_MONTH	Defines the MONTH qualifier
TU_DAY	Defines the DAY qualifier
TU_HOUR	Defines the HOUR qualifier
TU_MINUTE	Defines the MINUTE qualifier

TU_SECOND	Defines the SECOND qualifier
TU_FRAC	Defines default FRACTION(3) qualifier
TU_F1	Defines the FRACTION(1) qualifier
TU_F2	Defines the FRACTION(2) qualifier
TU_F3	Defines the FRACTION(3) qualifier
TU_F4	Defines the FRACTION(4) qualifier
TU_F5	Defines the FRACTION(5) qualifier
TU_END(q)	Returns the end qualifier of a composed qualifier
TU_START(q)	Returns the start qualifier of a composed qualifier
TU_LEN(q)	Returns the length in digits of a datetime qualifier
TU_DTENCODE(q1,q2)	Build a datetime qualifier as DATETIME q1 TO q2
TU_IENCODE(len,q1,q2)	Build an interval qualifier as INTERVAL q1(len) TO q2
TU_CURRQUAL	Default qualifier used by current

Data type identification macros

The following macros are used by functions like rsetnull() and risnull():

Macro name	Description
SQLCHAR	SQL CHAR data type
SQLSMINT	SQL SMALLINT data type
SQLINT	SQL INTEGER data type
SQLFLOAT	SQL FLOAT data type
SQLSMFLOAT	SQL SMALLFLOAT data type
SQLDECIMAL	SQL DECIMAL data type
SQLSERIAL	SQL SERIAL data type
SQLDATE	SQL DATE data type
SQLMONEY	SQL MONEY data type
SQLDTIME	SQL DATETIME data type
SQLBYTES	SQL BYTE data type
SQLTEXT	SQL TEXT data type
SQLVCHAR	SQL VARCHAR data type
SQLINTERVAL	SQL INTERVAL data type
SQLNCHAR	SQL NCHAR data type
SQLNVCHAR	SQL NVARCHAR data type
SQLINT8	SQL INT8 data type
SQLSERIAL8	SQL SERIAL8 data type
CCHARTYPE	C char data type
CSHORTTYPE	C short int data type
CINTTYPE	C int4 data type
CLONGTYPE	C long data type
CFLOATTYPE	C float data type
CDOUBLETYPE	C double data type
CDECIMALTYPE	C dec_t data type
CFIXCHARTYPE	C fixchar data type
CSTRINGTYPE	C string data type
CDATETYPE	C int4/date data type

CMONEYTYPE	C dec_t data type
CDTIMETYPE	C dtime_t data type
CLOCATORTYPE	C loc_t data type
CVCHARTYPE	C varchar data type
CINVTYPE	C intrvl_t data type

Calling C functions from BDL

With C-Extensions you can call C functions from the program in the same way that you call normal functions.

The C functions that can be called from BDL must use the following signature:

```
int function-name( int )
```

Warning: *function-name* must be written in lowercase letters. The fglcomp compiler converts all BDL functions names to lowercase.

The C function must be declared in the **usrFunctions** array in the C Interface File, as described below.

Calling BDL functions from C

It is possible to call an BDL function from a C function, by using the **fgl_call** macro in your C function, as follows:

```
fgl_call ( function-name, nbparams );
```

function-name is the name of the BDL function to call, and *nbparams* is the number of parameters pushed on the stack for the BDL function.

Warning: *function-name* must be written in lowercase letters (The fglcomp compiler converts all BDL functions names to lowercase)

The fgl_call macro is converted to a function that returns the number of values returned on the stack. The BDL function parameters must be pushed on the stack before the call, and the return values must be popped from the stack after returning. See Calling C functions from BDL for more details about push and pop library functions.

Example:

```
01 #include <stdio.h>
02 #include <f2c/fglExt.h>
03 int c_fct( int n )
04 {
```

```
05    int rc, r1, r2;
06       pushint(123456);
07    rc = fgl_call( fgl_fct, 1 );
08    if (rc != 2) ... error ...
09    popint(&r1);
10    popint(&r2);
11    return 0;
12 }
```

Sharing global variables

Prior to version 2.02, it was not possible to share global variables with C Extensions.

In order to share the global variables declared in your BDL program, you must do the following:

1. Generate the .c and .h interface files by using **fglcomp -G** with the BDL module defining the global variables:

```
01 GLOBALS
02 DEFINE g_name CHAR(100)
03 END GLOBALS
fglcomp -G myglobals.4gl
```

This will produce two files named **myglobals.h** and **myglobals.c**.

2. In the C module, include the generated header file and use the global variables directly:

```
01 #include <string.h>
02 #include <f2c/fglExt.h>
03 #include "myglobals.h"
04
05 int myfunc1(int c)
06 {
07                              strcpy(g_name, "new name");
08                              return 0;
09 }
```

3. When creating the C Extension library, compile and link with the myglobals.c generated file.

Tip: It is not recommended to use global variables, because it makes your code much more difficult to maintain. If you need persistent variables, use BDL module variables and write set/get functions that you can interface with.

Example

Warning: This example shows how to create a C extension library on Linux using gcc. The command line options to compile and link shared libraries can change depending on the operating system and compiler/linker used.

Create the "split.c" file:

```
01 #include <string.h>
02 #include <f2c/fglExt.h>
03
04 int fgl_split( int in_num );
05 int fgl_split( int in_num )
06 {
07                              char c1[101];
08                              char c2[101];
09                              char z[201];
10                              char *ptr_in;
11                              char *ptr_out;
12                              popvchar(z, 200); /* Getting input
parameter */
13                              strcpy(c1, "");
14                              strcpy(c2, "");
15                              ptr_out = c1;
16                              ptr_in = z;
17                              while (*ptr_in != ' ' && *ptr_in != '\0')
18                              {
19                                                                  *ptr_out =
*ptr_in;
20                                                                  ptr_out++;
21                                                                  ptr_in++;
22                              }
23                              *ptr_out=0;
24                              ptr_in++;
25                              ptr_out = c2;
26                              while (*ptr_in != '\0')
27                              {
28                                                                  *ptr_out =
*ptr_in;
29                                                                  ptr_out++;
30                                                                  ptr_in++;
31                              }
32                              *ptr_out=0;
33                              pushvchar(c1, 100); /* Returning the
first output parameter */
34                              pushvchar(c2, 100); /* Returning the
second output parameter */
35                              return 2; /* Returning the number of
output parameters (MANDATORY) */
36 }
```

Create the "splitext.c" C interface file:

```
01 #include <f2c/fglExt.h>
```

```
02
```

```
03 int fgl_split(int);
04
```

```
05 UsrFunction usrFunctions[]={
06   { "fgl_split", fgl_split, 1, 2 },
07   { 0,0,0,0 }
08 };
```

Compile the C Module and the interface file:

```
gcc -c -I $FGLDIR/include -fPIC split.c
gcc -c -I $FGLDIR/include -fPIC splitext.c
```

Create the shared library:

```
gcc -shared -o libsplit.so split.o splitext.o -L$FGLDIR/lib -lfgl
```

Create the BDL program "split.4gl":

```
01 IMPORT libsplit
02 MAIN
03       DEFINE str1, str2 VARCHAR(100)
04       CALL fgl_split("Hello World") RETURNING str1, str2
05       DISPLAY "1: ", str1
06       DISPLAY "2: ", str2
07 END MAIN
```

Compile the 4gl module:

```
fglcomp split.4gl
```

Run the program without the -e option:

```
fglrun split
```

C API Functions

Function	Description
bycmpr()	Compares two groups of contiguous bytes
byleng()	Returns the number of bytes as significant characters in the specified string
bycopy()	Copies a specified number of bytes to another location in

	memory
byfill()	Fills a specified number of bytes in memory
risnull()	Checks whether a given value is NULL
rsetnull()	Sets a variable to NULL for a given data type
rgetmsg()	Returns a message text
rgetlmsg()	Returns a message text
rtypalign()	Returns the position to align a variable at the proper boundary for its data type
rtypmsize()	Returns the size in bytes required for a specified data type
rtypname()	Returns the name of a specified data type
rtypwidth()	Returns the minimum number of characters required to convert a specified data type to a character data type
rdatestr()	Converts a date to a string
rdayofweek()	Returns the week day of a date
rdefmtdate()	Converts a string to a date by using a specific format
ifx_defmtdate()	Converts a string to a date by using a specific format, with century option
rfmtdate()	Converts a date to a string by using a specific format
rjulmdy()	Extracts month, day and year from a date
rleapyear()	Checks whether a year is a leap year
rmdyjul()	Builds a date from month, day, year
rstrdate()	Converts a string to a date
ifx_strdate()	Converts a date to a string by using a specific format, with the century option
rtoday()	Returns the current date
ldchar()	Copies a fixed-length string into a null-terminated string without trailing spaces
rdownshift()	Converts a string to lowercase characters
rfmtdouble()	Formats a double value in a specified format
rfmtint4()	Formats a 4-byte int value in a specified format
rstod()	Converts a string to a double
rstoi()	Converts a string to a 2-byte integer (but it takes an int pointer as parameter!)
rstol()	Converts a string to a 4-byte integer (but it takes an long pointer as parameter!)
rupshift()	Converts a string to uppercase characters
stcat()	Concatenates a null-terminated string to another
stchar()	Concatenates a null-terminated string to a fixed char
stcmpr()	Compares two strings
stcopy()	Copies a string into another
stleng()	Returns the number of bytes of significant characters, including

	trailing spaces
decadd()	Adds two decimals
deccmp()	Compares two decimals
deccopy()	Copies one decimal into another
deccvasc()	Converts a string to a decimal
deccvdbl()	Converts a double to a decimal
deccvflt()	Converts a float to a decimal
deccvint()	Converts a 4-byte integer to a decimal (parameter is a machine-dependent int pointer)
deccvlong()	Converts a 4-byte integer to a decimal
decdiv()	Divides a decimal by another decimal
dececvt()	To convert a decimal value to a string value, specifying the length of the string (the total number of digits)
decfcvt()	To convert a decimal value to a string value, specifying the number of digits to the right of the decimal point
decmul()	Multiplies a decimal by another
decround()	Rounds a decimal to the specified number of digits
decsub()	Subtracts two decimals
dectoasc()	Formats a decimal
dectodbl()	Converts a decimal to a double
dectoflt()	Converts a decimal to a float
dectoint()	Converts a decimal to a 2-byte integer (parameter is machine-dependent int pointer)
dectolong()	Converts a decimal to a 4-byte integer
dectrunc()	Truncates a decimal to a given number of digits
rfmtdec()	Formats a decimal
dtaddinv()	Adds an interval to a datetime
dtcurrent()	Returns the current datetime
dtcvasc()	Converts a string in ISO format to a datetime
ifx_dtcvasc()	Converts a string in ISO format to a datetime, with century option
dtcvfmtasc()	Converts a string to a datetime by using a specific format
ifx_dtcvfmtasc()	Converts a string to a datetime by using a specific format, with the century option
dtextend()	Extends a datetime
dtsub()	Subtracts a datetime from another datetime
dtsubinv()	Subtracts an interval from a datetime
dttoasc()	Formats a datetime in ISO format
dttofmtasc()	Formats a datetime in a specified format
ifx_dttofmtasc()	Formats a datetime in a specified format, with the century option
incvasc()	Converts a string in ISO format to an interval

incvfmtasc()	Converts a string to an interval by using a specific format
intoasc()	Formats an interval in ISO format
intofmtasc()	Formats an interval in a specified format
invdivdbl()	Divides an interval by using a double
invdivinv()	Divides an interval by using another interval
invextend()	Extends an interval
invmuldbl()	Multiplies an interval by a double

---

decadd()

Purpose:

To add two decimal values

Syntax:

```
mint decadd(dec_t *n1, struct decimal *n2, struct decimal *n3);
```

Notes:

1. *n1* is a pointer to the decimal structure of the first operand.
2. *n2* is a pointer to the decimal structure of the second operand.
3. *n3* is a pointer to the decimal structure that contains the sum (*n1* + *n2*).

Returns:

Code	Description
0	The operation was successful.
-1200	The operation resulted in overflow.
-1201	The operation resulted in underflow.

---

decsub()

Purpose:

To subtract two decimal values

Syntax:

```
mint decsub(dec_t *n1, struct decimal *n2, struct decimal *n3);
```

Notes:

1. *n1* is a pointer to the decimal structure of the first operand.
2. *n2* is a pointer to the decimal structure of the number to be subtracted.
3. *n3* is a pointer to the decimal structure that contains the result of (*n1* minus *n2*).

Returns:

Code	Description
0	The operation was successful.
-1200	The operation resulted in overflow.
-1201	The operation resulted in underflow.

---

decmul()

Purpose:

To multiply two decimal values

Syntax:

```
int decmul(dec_t *n1, struct decimal *n2, struct decimal *n3);
```

Notes:

1. *n1* is a pointer to the decimal structure of the first operand.
2. *n2* is a pointer to the decimal structure of the second operand.
3. *n3* is a pointer to the decimal structure that contains the product of (*n1* times *n2*).

Returns:

Code	Description
0	The operation was successful.
-1200	The operation resulted in overflow.
-1201	The operation resulted in underflow.

---

decdiv()

Purpose:

To divide one decimal value by another(*n1* divided by *n2*)

Syntax:

```
mint decdiv(dec_t *n1, struct decimal *n2, struct decimal *n3);
```

Notes:

1. *n1* is a pointer to the decimal structure of the number to be divided.
2. *n2* is a pointer to the decimal structure of the number that is the divisor.
3. *n3* is a pointer to the decimal structure that contains the quotient of (*n1* divided by *n2*).

Returns:

Code	Description
0	The operation was successful.
-1200	The operation resulted in overflow.
-1201	The operation resulted in underflow.
-1202	The operation attempted to divide by zero.

---

deccmp()

Purpose:

To compare two decimal values

Syntax:

```
mint deccmp(dec_t *n1, struct decimal *n2);
```

Notes:

1. *n1* is a pointer to the decimal structure of the first number to compare.
2. *n2* is a pointer to the decimal structure of the second number to compare.

Returns:

Code	Description
0	The two values are identical.
1	The first value is greater than the second value.
-1	The first value is less than the second value.
-2	Either value is null.

---

deccopy()

Purpose:

To copy one decimal value to another

Syntax:

```
void deccopy(dec_t *n1, struct decimal *n2);
```

Notes:

1. *n1* is a pointer to the value held in the source decimal structure.
2. *n2* is a pointer to the target decimal structure.

---

deccvasc()

Purpose:

To convert a string value to a decimal

Syntax:

```
mint deccvasc(char *cp, mint len, dec_t *np);
```

Notes:

1. *cp* is a pointer to a string to be converted to a decimal value.
2. *len* is the length of the string.
3. *np* is a pointer to the decimal structure which contains the result of the conversion.

Returns:

Code	Description
0	The conversion was successful.
-1200	The number is too large to fit into a decimal type structure.

---

deccvdbl()

Purpose:

To convert a double value to a decimal

Syntax:

```
mint deccvdbl(double dbl, dec_t *np);
```

Notes:

1. *dbl* is the double value to convert to a decimal type value.
2. *np* is a pointer to a decimal structure containing the result of the conversion.

Returns:

Code	Description
0	The conversion was successful.
<0	The conversion failed.

---

deccvint()

Purpose:

To convert a 4-byte integer value to a decimal

Syntax:

```
mint deccvint(mint in, dec_t *np);
```

Notes:

1. *in* is the mint value to convert to a decimal type value.
2. *np* is a pointer to a decimal structure to contain the result of the conversion.

Returns:

Code	Description
0	The conversion was successful.
<0	The conversion failed.

---

deccvlong()

Purpose:

To convert a 4-byte integer value to a decimal

Syntax:

```
mint deccvlong(int4 lng, dec_t *np);
```

Notes:

1. *lng* is the int4 value to convert to a decimal type value.
   np is a pointer to a decimal structure to contain the result of the conversion.

Returns:

Code	Description
0	The conversion was successful.
<0	The conversion failed.

Warning: Even if the function name is "deccvlong", it takes a 4-byte int as argument.

---

dececvt()

Purpose:

To convert a decimal value to a string value, specifying the length of the string (the total number of digits)

Syntax:

```
char *dececvt(dec_t *np, mint ndigit, mint *decpt, mint *sign);
```

Notes:

1. *np* is a pointer to a decimal structure that contains the decimal value you want to convert.
2. *ndigit* is the length of the ASCII string.
3. *decpt* is a pointer to an integer that is the position of the decimal point relative to the start of the string. A negative or zero value for
   decpt means to the left of the returned digits.
4. *sign* is a pointer to the sign of the result. If the sign of the result is negative, *sign* is nonzero; otherwise, *sign* is zero.

Returns:

This function returns a pointer to the result string. This string is temporary and has to be copied into your own string buffer.

---

decfcvt()

Purpose:

To convert a decimal value to a string value, specifying the number of digits to the right of the decimal point

Syntax:

```
char *decfcvt(dec_t *np, mint ndigit, mint *decpt, mint *sign);
```

Notes:

1. *np* is a pointer to a decimal structure that contains the decimal value you want to convert.
2. *ndigit* is the number of digits to the right of the decimal point.
3. *decpt* is a pointer to an integer that is the position of the decimal point relative to the start of the string. A negative or zero value for
   decpt means to the left of the returned digits.
4. *sign* is a pointer to the sign of the result. If the sign of the result is negative, *sign* is nonzero; otherwise, *sign* is zero.

Returns:

This function returns a pointer to the result string. This string is temporary and has to be copied into your own string buffer.

---

decround()

Purpose:

To round a decimal value, specifying the number of digits to the right of the decimal point

Syntax:

```
void decround(dec_t *np, mint dec_round);
```

Notes:

1. *np* is a pointer to a decimal structure whose value is to be rounded. Use a positive number for the *np* argument.
2. *dec_round* is the number of fractional digits to which the value is rounded.

---

dectoasc()

Purpose:

To convert a decimal value to an ASCII string, specifying the length of the string and the number of digits to the right of the decimal point

Syntax:

```
mint dectoasc(dec_t *np, char *cp, mint len, mint right);
```

Notes:

1. *np* is a pointer to the decimal structure to convert to a text string.
2. *cp* is a pointer to the first byte of the character buffer to hold the text string.
3. *len* is the size of strng_val, in bytes, minus 1 for the null terminator.
4. *right* is an integer that indicates the number of decimal places to the right of the decimal point.
5. Because the character string that dectoasc() returns is not null terminated, your program must add a null character to the string before you print it.

Returns:

Code	Description
0	The conversion was successful.
<0	The conversion failed.

---

dectodbl()

Purpose:

To convert a decimal value to a double value

Syntax:

```
mint dectodbl(dec_t *np, double *dblp);
```

Notes:

1. *np* is a pointer to the decimal structure to convert to a double type value.
2. *dblp* is a pointer to a double type where dectodbl() places the result of the conversion.

Returns:

Code	Description

0	The conversion was successful.
<0	The conversion failed.

---

dectoint()

Purpose:

To convert a decimal value to a SMALLINT equivalent (2-byte integer)

Syntax:

```
mint dectoint2(dec_t *np, mint *ip);
```

Notes:

1. *np* is a pointer to the decimal structure to convert to a mint type value.
2. *ip* is a pointer to a mint value where dectoint() places the result of the conversion.

Returns:

Code	Description
0	The conversion was successful.
<0	The conversion failed.
-1200	The magnitude of the decimal type number is greater than 32767.

Warning: This functions takes a machine-dependent int pointer as argument (usually 4 bytes), but converts the decimal to a SMALLINT equivalent, with possible overflow errors.

---

dectolong()

Purpose:

To convert a decimal value to a long integer

Syntax:

```
mint dectolong(dec_t *np, int4 *lngp);
```

Notes:

1. *np* is a pointer to the decimal structure to convert to an int4 integer.

2. *lngp* is a pointer to an int4 integer where dectolong() places the result of the conversion.

Returns:

Code	Description
0	The conversion was successful.
<0	The conversion failed.
-1200	The magnitude of the decimal type number is greater than 2,147,483,647.

Warning: Even if the function name is "dectolong", it takes a 4-byte int pointer as argument, and converts the decimal to an INTEGER equivalent.

---

dectrunc()

Purpose:

To truncate a decimal value, specifying the number of digits to the right of the decimal point

Syntax:

```
void dectrunc(dec_t *np, mint trunc);
```

Notes:

1. *np* is a pointer to the decimal structure for a rounded number to truncate.
2. *trunc* is the number of fractional digits to which dectrunc() truncates the number. Use a positive number or zero for this argument.

---

deccvflt()

Purpose:

To convert a float to a decimal value

Syntax:

```
mint deccvflt(float source, dec_t *destination);
```

Notes:

1. *source* is the float value to be converted.
2. *destination* is a pointer to the structure where the decimal value is placed.

Returns:

Code	Description
0	The conversion was successful.
<0	The conversion failed.

---

dectoflt()

Purpose:

To convert a decimal value to a float

Syntax:

```
mint dectoflt(dec_t *source, float *destination);
```

Notes:

1. *source* is a pointer to the decimal value to convert.
2. *destination* is a pointer to the resulting float value.

Returns:

Code	Description
0	The conversion was successful.
<0	The conversion failed.

---

rfmtdec()

Purpose:

To convert a decimal value to a string having a specified format

Syntax:

```
int rfmtdec(dec_t *dec, char *format, char *outbuf);
```

Notes:

1. *dec* is a pointer to the decimal value to format.
2. *format* is a pointer to a character buffer that contains the formatting mask to use.
3. *outbuf* is a pointer to a character buffer that receives the resulting formatted string.

Returns:

Code	Description
0	The conversion was successful.
<0	The conversion failed.
-1211	The program ran out of memory (memory-allocation error).
-1217	The format string is too large.

---

bycmpr()

Purpose:

To compare two groups of contiguous bytes, for a specified length, byte-by-byte.

Syntax:

```
mint bycmpr(char *st1, char *st2, mint count);
```

Notes:

1. *st1* is a pointer to the location where the first group of bytes starts.
2. *st2* is a pointer to the location where the second group of bytes starts.
3. *count* is the number of bytes to compare.

Returns:

Code	Description
0	The two groups of bytes are identical.
1	The *st1* group of bytes is less than the *st2* group.
-1	The *st1* group of bytes is greater than the *st2* group.

---

bycopy()

Purpose:

To copy a specified number of bytes to another location in memory

Syntax:

```
void bycopy(char *s1, char *s2, mint n);
```

Notes:

1. *s1* is a pointer to the first byte of the group of bytes that you want to copy.
2. *s2* is a pointer to the first byte of the destination group of bytes.
   If the location pointed to by *s2* overlaps the location pointed to by *s1*, the function will not preserve the value of *s1*.
3. *n* is the number of bytes to be copied.

Warning: Do not overwrite the memory areas adjacent to the destination area.

---

byfill()

Purpose:

To fill a specified number of bytes with a specified character

Syntax:

```
void byfill(char *s1, mint n, char c);
```

Notes:

1. *s1* is a pointer to the first byte of the memory area that you want to fill.
   n is the number of times that you want to repeat the character within the area.
   c is the character that you want to use to fill the area.

Warning: Do not overwrite the memory areas adjacent to the destination area.

---

risnull()

Purpose:

To check whether a variable is null

Syntax:

```
int risnull(int vtype, char *pcvar);
```

Notes:

1. *vtype* is an integer corresponding to the data type of the variable.
   This parameter must be one of the Date Type Constants defined in fglExt.h.
2. *pcvar* is a pointer to the C variable.

Returns:

Code	Description
1	The variable does contain a null value.
0	The variable does not contain a null value.

---

rsetnull()

Purpose:

To set a variable to NULL

Syntax:

```
mint rsetnull(mint vtype, char *pcvar);
```

Notes:

1. *type* is an integer corresponding to the data type of the variable.
   This parameter must be one of the Date Type Constants defined in fglExt.h.
2. *pcvar* is a pointer to the variable.

Returns:

Code	Description
0	The operation was successful.
<0	The operation failed.

---

rgetmsg()

Purpose:

Returns the error message for a specified error number, restricted to two-byte integers.

Syntax:

```
mint rgetmsg(int msgnum, char *s, mint maxsize);
```

Notes:

1. *msgnum* is the error number, restricted to error numbers between -32768 and +32767.
2. *s* is a pointer to the buffer that receives the message string (the output buffer).
3. *maxsize* is the size of the output buffer. This value should be set to the size of the largest message that you expect to retrieve.
4. The Informix message text files are used to retrieve the message.

Returns:

Code	Description
0	The operation was successful.
-1227	Message file not found.
-1228	Message number not found in message file.
-1231	Cannot seek within message file.
-1232	Message buffer is too small.

Warning: This function returns Informix specific messages; it will not work properly if the Informix client software is not installed.

---

rgetlmsg()

Purpose:

Returns the error message for a specified error number, which can be a 4-byte integer.

Syntax:

```
int4 rgetlmsg(int msgnum, char *s, mint maxsize, mint *msg_length);
```

Notes:

1. *msgnum* is the error number. The four-byte parameter provides for the full range of Informix-specific error numbers.

2. *s* is a pointer to the buffer that receives the message string (the output buffer).
3. *maxsize* is the size of the msgstr output buffer. Make this value the size of the largest message that you expect to retrieve.
4. *msg_length* is a pointer to the mint that contains the actual length of the message that rgetlmsg() returns.

Returns:

Code	Description
0	The operation was successful.
-1227	Message file not found.
-1228	Message number not found in message file.
-1231	Cannot seek within message file.
-1232	Message buffer is too small.

Warning: This function returns Informix specific messages; it will not work properly if the Informix client software is not installed.

---

rtypalign()

Purpose:

Returns the position to align a variable at the proper boundary for its data type

Syntax:

```
mlong rtypalign(mlong pos, mint datatype)
```

Notes:

1. *pos* is the current position in the buffer.
2. *datatype* is an integer code defining the data type.
   This parameter must be one of the Date Type Constants defined in fglExt.h.

Returns:

Code	Description
>0	The return value is the offset of the next proper boundary for a variable of type data type.

---

rtypmsize()

Purpose:

Returns the size in bytes required for a specified data type

Syntax:

```
mint rtypmsize(mint datatype, mint length)
```

Notes:

1. *datatype* is an integer code defining the data type.
   This parameter must be one of the Date Type Constants defined in fglExt.h.
2. *length* is the number of bytes in the data file for the specified type.

Returns:

Code	Description
0	The *datatype* is not a valid SQL type.
>0	The return value is the number of bytes that the data type requires.

---

rtypname()

Purpose:

Returns a pointer to a null-terminated string containing the name of the data type

Syntax:

```
char *rtypname(mint datatype)
```

Notes:

1. *datatype* is an integer code defining the data type.
   This parameter must be one of the Date Type Constants defined in fglExt.h.

Returns:

The rtypname function returns a pointer to a string that contains the name of the data type specified *datatype*.

If *datatype* is an invalid value, rtypname() returns a null string (" ").

rtypwidth()

Purpose:

Returns the minimum number of characters required to convert a specified data type to a character data type

Syntax:

```
mint rtypwidth(mint datatype, mint length)
```

Notes:

1. *datatype* is an integer code defining the data type.
   This parameter must be one of the Date Type Constants defined in fglExt.h.
2. *length* is the number of bytes in the data file for the specified data type.

Returns:

Code	Description
0	The sqltype is not a valid SQL data type.
>0	Minimum number of characters that the sqltype data type requires.

rdatestr()

Purpose:

To convert a date that is in the native database date format to a string

Syntax:

```
mint rdatestr(int4 jdate, char *str);
```

Notes:

1. *jdate* is the internal representation of the date to format.
2. *str* is a pointer to the buffer that receives the string for the date value.

Returns:

Code	Description
0	The conversion was successful.

<0	The conversion failed.
-1210	The internal date could not be converted to the character string format.
-1212	Data conversion format must contain a month, day, or year component.

Usage:

The DBDATE environment variable specifies the data conversion format.

---

rdayofweek()

Purpose:

Returns the day of the week of a date that is in the native database format

Syntax:

```
mint rdayofweek(int4 jdate);
```

Notes:

1. *jdate* is the internal representation of the date.

Returns:

Code	Description
<0	Invalid date value
0	Sunday
1	Monday
2	Tuesday
3	Wednesday
4	Thursday
5	Friday
6	Saturday

---

rdefmtdate()

Purpose:

To convert a string in a specified format to the native database date format

Syntax:

```
mint rdefmtdate(int4 *pdate, char *fmt, char *input);
```

Notes:

1. *pdate* is a pointer to an int4 integer value that receives the internal DATE value for the *input* string.
2. *fmt* is a pointer to the buffer that contains the formatting mask for the string.
3. *input* is a pointer to the buffer that contains the string to convert.

Returns:

Code	Description
0	The operation was successful.
-1204	The *input* parameter specifies an invalid year.
-1205	The *input* parameter specifies an invalid month.
-1206	The *input* parameter specifies an invalid day.
-1209	Because *input* does not contain delimiters between the year,month,and day, the length of *input* must be exactly six or eight bytes.
-1212	*fmt* does not specify a year, a month, and a day.

---

rfmtdate()

Purpose:

To convert a date that is in the native database date format to a string having a specified format

Syntax:

```
mint rfmtdate(int4 jdate, char *fmt, char *result);
```

Notes:

1. *jdate* is the internal representation of a date to convert.
2. *fmt* is a pointer to the buffer containing the formatting mask.
3. *result* is a pointer to the buffer that receives the formatted string.

Returns:

Code	Description
0	The operation was successful.
-1210	The internal date cannot be converted to month-day-year format.
-1211	The program ran out of memory (memory-allocation error).
-1212	Format string is NULL or invalid.

---

rjulmdy()

Purpose:

To create an array of short integer values representing year, month, and day from a date that is in the native database date format

Syntax:

```
mint rjulmdy(int4 jdate, int2 mdy[3]);
```

Notes:

1. *jdate* is the internal representation of a date.
2. *mdy* is an array of short integers, where mdy[0] is the month (1 to 12), mdy[1] is the day (1 to 31), and mdy[2] is the year (1 to 9999).

Returns:

0 The operation was successful.
< 0 The operation failed.
-1210 The internal date could not be converted to the character string format.

---

rleapyear()

Purpose:

To determine whether the value passed as a parameter is a leap year; returns 1 when TRUE.

Syntax:

```
mint rleapyear(mint year);
```

Notes:

1. year is an integer, representing the year component of a date, in the full form yyyy (ie, 2004).

Returns:

Code	Description
1	The year is a leap year.
0	The year is not a leap year.

---

rmdyjul()

Purpose:

To create a value in the native database date format from an array of short integer values representing month, day, and year

Syntax:

```
mint rmdyjul(int2 mdy[3], int4 *jdate);
```

Notes:

1. *mdy* is an array of short integer values, where *mdy*[0] is the month (1 to12), *mdy*[1] is the day (1 to 31), and *mdy*[2] is the year (1 to 9999).
2. *jdate* is a pointer to a long integer that receives the internal DATE value for the *mdy* array.

Returns:

Code	Description
0	The operation was successful.
-1204	The *mdy*[2] variable contains an invalid year.
-1205	The *mdy*[0] variable contains an invalid month.
-1206	The *mdy*[1] variable contains an invalid day.

---

rstrdate()

Purpose:

To convert a character string to the native database date format.

Syntax:

```
mint rstrdate(char *str, int4 *jdate);
```

Notes:

1. *str* is a pointer to a char string containing the date to convert.
2. *jdate* is a pointer to an int4 integer that receives the converted date value.

Returns:

Code	Description
0	The conversion was successful.
<0	The conversion failed.
-1204	The *str* parameter specifies an invalid year.
-1205	The *str* parameter specifies an invalid month.
-1206	The *str* parameter specifies an invalid day.
-1212	Data conversion format must contain a month, day, or year component. DBDATE specifies the data conversion format.
-1218	The date specified by the *str* argument does not properly represent a date.

Usage:

The DBDATE environment variable specifies the data conversion format.

---

rtoday()

Purpose:

Returns the system date in the internal database date format

Syntax:

```
void rtoday(int4 *today);
```

Notes:

1. *today* is a pointer to an int4 value that receives the internal date.

---

ifx_defmtdate()

Purpose:

To convert a string in a specified format to the native database date format; allows you to specify the century setting for two-digit dates.

Syntax:

```
mint ifx_defmtdate(int4 *pdate, char *fmt, char *input, char c);
```

Notes:

1. *pdate* is a pointer to an int4 integer value that receives the internal DATE value for the *input* string.
2. *fmt* is a pointer to the buffer that contains the formatting mask to use for the *input* string.
3. *input* is a pointer to the buffer that contains the date string to convert.
4. *c* is one of CENTURY characters, which determines which century to apply to the year portion of the date.

Returns:

Code	Description
0	The conversion was successful.
<0	The conversion failed.
-1204	The *str* parameter specifies an invalid year.
-1205	The *str* parameter specifies an invalid month.
-1206	The *str* parameter specifies an invalid day.
-1212	Data conversion format must contain a month, day, or year component. DBDATE specifies the data conversion format.
-1209	Because *input does not contain delimiters between the year, month, and day, the length of *input must

be exactly six or eight bytes.

---

ifx_strdate()

Purpose:

To convert a character string to the native database date format; allows you to specify the century setting for two-digit dates.

Syntax:

```
mint ifx_strdate(char *str, int4 *jdate, char c);
```

Notes:

1. *str* is a pointer to the string that contains the date to convert.
2. *jdate* is a pointer to a int4 integer that receives the internal DATE value for the *str* string.
3. *c* is one of CENTURY characters, which determines which century to apply to the year portion of the date.

Returns:

Code	Description
0	The conversion was successful.
<0	The conversion failed.
-1204	The *str* parameter specifies an invalid year.
-1205	The *str* parameter specifies an invalid month.
-1206	The *str* parameter specifies an invalid day.
-1212	Data conversion format must contain a month, day, or year component. DBDATE specifies the data conversion format.
-1218	The date specified by the *str* argument does not properly represent a date.

Usage:

The DBDATE environment variable specifies the data conversion format.

---

byleng()

Purpose:

Returns the number bytes as significant characters in the specified string; omitting trailing blanks

Syntax:

```
mint byleng(char *s1, mint count);
```

Notes:

1. *s1* is a pointer to a fixed-length string, not null-terminated.
2. *count* is the number of bytes in the fixed-length string.

Returns:

Number of bytes.

---

ldchar()

Purpose:

To copy a fixed-length string into a null-terminated string without trailing blanks

Syntax:

```
void ldchar(char *from, mint count, char *to);
```

Notes:

1. *from* is a pointer to a fixed-length source string.
2. *count* is the number of bytes in the source string.
3. *to* is a pointer to the first byte of a null-terminated destination string. If the *to* argument points to the same location as the *from* argument, or to a location that overlaps the *from* argument, ldchar() does not preserve the original value.

---

rdownshift()

Purpose:

To convert all the characters in a null-terminated string to lowercase.

Syntax:

```
void rdownshift(char *s);
```

Notes:

1. *s* is a pointer to a null-terminated string.

---

rupshift()

Purpose:

To convert all the characters in a null-terminated string to uppercase.

Syntax:

```
void rupshift(char *s);
```

Notes:

1. *s* is a pointer to a null-terminated string.

---

stcat()

Purpose:

To concatenate one null-terminated string to another (*src* is added to the end of *dst)*.

Syntax:

```
void stcat(char *src, char *dst);
```

Notes:

1. *src* is a pointer to the start of the string that is put at the end of the destination string.
2. *dst* is a pointer to the start of the null-terminated destination string.
3. The resulting string is *dstsrc.*

---

stcopy()

Purpose:

To copy a string to another location

Syntax:

```
void stcopy(char *src, char *dst);
```

Notes:

1. *src* is a pointer to the string that you want to copy.
2. *dst* is a pointer to a location in memory where the string is copied.

---

stleng()

Purpose:

Returns the number of bytes of significant characters, including trailing blanks

Syntax:

```
mint stleng(char *src);
```

Notes:

1. *src* is a pointer to a null-terminated string.
2. The length does not include the null terminator.

Returns:

Number of bytes.

---

stcmpr()

Purpose:

To compare two strings

Syntax:

```
mint stcmpr(char *s1, char *s2);
```

Notes:

1. *s1* is a pointer to the first null-terminated string.
2. *s2* is a pointer to the second null-terminated string.
3. *s1* is greater than *s2* when *s1* appears after *s2* in the ASCII collation sequence.

Returns:

Code	Description
0	The two strings are identical.
<0	The first string is less than the second string.
>0	The first string is greater than the second string.

---

stchar()

Purpose:

To copy a null-terminated string into a fixed-length string

Syntax:

```
void stchar(char *from, char *to, mint count);
```

Notes:

1. *from* is a pointer to the first byte of a null-terminated source string.
2. *to* is a pointer to a fixed-length destination string. If this argument points to a location that overlaps the location to which the *from* argument points, the *from* value is discarded.
3. *count* is the number of bytes in the fixed-length destination string.

---

rstod()

Purpose:

To convert a string to a double.

Syntax:

```
mint rstod(char *str, double *val);
```

Notes:

1. *str* is a pointer to a null-terminated string.
2. *val* is a pointer to a double value that holds the converted value.

Returns:

Code	Description
0	The conversion was successful.
<0	The conversion failed.

---

rstoi()

Purpose:

To convert a string to a 2-byte integer.

Syntax:

```
mint rstoi(char *str, mint *val);
```

Notes:

1. *str* is a pointer to a null-terminated string.
2. *val* is a pointer to a mint value that holds the converted value.

Warning: The function takes a machine-dependent int pointer (usually 4 bytes), but the function converts to a 2 byte interger (overflow error may occur if string does not represent a valid 2-byte integer).

Returns:

Code	Description
0	The conversion was successful.
<0	The conversion failed.

---

rstol()

Purpose:

To convert a string to a 4-byte integer (machine dependent)

Syntax:

```
mint rstol(char *str, mlong *val);
```

Notes:

1. *str* is a pointer to a null-terminated string.
2. *val* is a pointer to an mlong value that holds the converted value.

Warning: The function takes a machine-dependent long pointer (4 bytes on 32b and 8 bytes on 64b architectures), but the function converts to a 4-byte integer (overflow error may occur if string does not represent a valid 2 byte integer).

Returns:

Code	Description
0	The conversion was successful.
<0	The conversion failed.

---

rfmtdouble()

Purpose:

To convert a double value to a character string having a specified format.

Syntax:

```
mint rfmtdouble(double dvalue, char *format, char *outbuf);
```

Notes:

1. *dvalue* is the double value to format.
2. *format* is a pointer to a char buffer that contains the formatting mask.
3. *outbuf* is a pointer to a char buffer that receives the formatted string.

Returns:

Code	Description
0	The conversion was successful.
<0	The conversion failed.
-1211	The program ran out of memory (memory-allocation error).
-1217	The format string is too large.

---

rfmtint4()

Purpose:

To convert a 4-byte integer to a character string having a specified format

Syntax:

```
mint rfmtint4(int4 lvalue, char *format, char *outbuf);
```

Notes:

1. *lvalue* is the int4 integer to convert.
2. *format* is a pointer to the char buffer that contains the formatting mask.
3. *outbuf* is a pointer to a char buffer that receives the formatted string for the integer value.

Returns:

Code	Description
0	The conversion was successful.
<0	The conversion failed.
-1211	The program ran out of memory (memory-allocation error).
-1217	The format string is too large.

---

dtaddinv()

Purpose:

To add an interval value to a datetime value

Syntax:

```
mint dtaddinv(dtime_t *d, intrvl_t *i, dtime_t *r);
```

Notes:

1. *d* is a pointer to the initialized datetime host variable. The variable must include all the fields present in the interval value.
2. *i* is a pointer to the initialized interval host variable. The interval value must be in the year to month or the day to fraction(5) range.
3. *r* is a pointer to the result. The result inherits the qualifier of *d*.
4. Failure to initialize the host variables can produce unpredictable results.

Returns:

Code	Description
0	The addition was successful.
<0	The addition failed.

---

dtcurrent()

Purpose:

Returns the current date and time

Syntax:

```
void dtcurrent(dtime_t *d);
```

Notes:

1. *d* is a pointer to the initialized datetime host variable.
2. The function extends the current date and time to agree with the qualifier of the host variable.

---

dtcvasc()

Purpose:

To convert an ASCII-standard character string to a datetime value

Syntax:

```
mint dtcvasc(char *str, dtime_t *d);
```

Notes:

1. *str* is a pointer to the buffer that contains the ASCII-standard datetime string.
2. *d* is a pointer to a datetime variable, initialized with the qualifier that you want the datetime value to have.

Returns:

Code	Description
0	The conversion was successful.
<0	The conversion failed.
-1260	It is not possible to convert between

	the specified types.
-1261	Too many digits in the first field of datetime or interval.
-1262	Non-numeric character in datetime or interval.
-1263	A field in a datetime or interval value is out of range or incorrect.
-1264	Extra characters exist at the end of a datetime or interval.
-1265	Overflow occurred on a datetime or interval operation.
-1266	A datetime or interval value is incompatible with the operation.
-1267	The result of a datetime computation is out of range.
-1268	A parameter contains an invalid datetime qualifier.

---

ifx_dtcvasc()

Purpose:

To convert a character string to a datetime value; allows you to specify the century setting for 2-digit years

Syntax:

```
mint ifx_dtcvasc(char *str, dtime_t *d, char c);
```

Notes:

1. *str* is a pointer to a buffer that contains an ANSI-standard datetime string.
2. *d* is a pointer to a datetime variable, initialized with the desired qualifier.
3. *c* is one of CENTURY characters, which determines which century to apply to the year portion of the date.

Returns:

Code	Description
0	The conversion was successful.
<0	The conversion failed.
-1260	It is not possible to convert between the specified types.
-1261	Too many digits in the first field of datetime or interval.
-1262	Non-numeric character in datetime

	or interval.
-1263	A field in a datetime or interval value is out of range or incorrect.
-1264	Extra characters exist at the end of a datetime or interval.
-1265	Overflow occurred on a datetime or interval operation.
-1266	A datetime or interval value is incompatible with the operation.
-1267	The result of a datetime computation is out of range.
-1268	A parameter contains an invalid datetime qualifier.

---

dtcvfmtasc()

Purpose:

To convert a character string to a datetime value, specifying the format of the string

Syntax:

```
mint dtcvfmtasc(char *input, char *fmt, dtime_t *d);
```

Notes:

1. *input* is a pointer to a buffer that contains the string to convert.
2. *fmt* is a pointer to a buffer containing the formatting mask.
   The default date format conforms to the standard ANSI SQL format: `%Y-%m-%d %H:%M:%S`
3. *d* is a pointer to the datetime variable, which must be initialized with the desired qualifier.

Returns:

Code	Description
0	The conversion was successful.
<0	The conversion failed.

---

ifx_dtcvfmtasc()

Purpose:

To convert a character string to a datetime value, specifying the format of the string

Syntax:

```
mint ifx_dtcvfmtasc(char *input, char *fmt, dtime_t *d, char c);
```

Notes:

1. *input* is a pointer to a buffer that contains the string to convert.
2. *fmt* is a pointer to a buffer containing the formatting mask.
   The default date format conforms to the standard ANSI SQL format: `%Y-%m-%d %H:%M:%S`
3. *d* is a pointer to the datetime variable, which must be initialized with the desired qualifier.
4. *c* is one of CENTURY characters, which determines which century to apply to the year portion of the date.

Returns:

Code	Description
0	The conversion was successful.
<0	The conversion failed.

---

dtextend()

Purpose:

To copy a datetime value *id* to the datetime value *od*, adding or dropping fields based on the qualifier of *od*

Syntax:

```
mint dtextend(dtime_t *id, dtime_t *od);
```

Notes:

1. *id* is a pointer to the datetime variable to extend.
2. *od* is a pointer to the datetime variable containing a valid qualifier to use for the extension.

Returns:

Code	Description
0	The operation was successful.
<0	The operation failed.
-1268	A parameter contains an invalid datetime qualifier

---

dtsub()

Purpose:

To subtract one datetime value from another

Syntax:

```
mint dtsub(dtime_t *d1, dtime_t *d2, intrvl_t *i);
```

Notes:

1. *d1* is a pointer to an initialized datetime host variable.
2. *d2* is a pointer to an initialized datetime host variable.
3. *i* is a pointer to the interval host variable that contains the result.

Returns:

Code	Description
0	The subtraction was successful.
<0	The subtraction failed.

---

dtsubinv()

Purpose:

To subtract an interval value from a datetime value

Syntax:

```
mint dtsubinv(dtime_t *d, intrvl_t *i, dtime_t *r);
```

Notes:

1. *d* is a pointer to an initialized datetime host variable. This must include all the fields present in the interval value *i*.
2. *i* is a pointer to an initialized interval host variable.
3. *r* is a pointer to the datetime host variable that contains the result.

Returns:

Code	Description
0	The subtraction was successful.
<0	The subtraction failed.

---

dttoasc()

Purpose:

To convert a datetime value to an ANSI-standard character string

Syntax:

```
mint dttoasc(dtime_t *d, char *str);
```

Notes:

1. *d* is a pointer to the initialized datetime variable to convert.
2. *str* is a pointer to the buffer that receives the ANSI-standard DATETIME string for the value in *d*.

Returns:

Code	Description
0	The conversion was successful.
<0	The conversion failed.

Usage:

The *str* parameter includes one character for each delimiter, plus the fields, which are of the following sizes:

Field	Field Size
year	four digits
fraction of datetime	as specified by precision
all other fields	two digits

For example, datetime year to fraction(5):

```
YYYY-MM-DD HH:MM:SS.FFFFF
```

---

dttofmtasc()

Purpose:

To convert a datetime value to a character string, specifying the format

Syntax:

```
mint dttofmtasc(dtime_t *d, char *output, mint len, char *fmt);
```

Notes:

1. *d* is a pointer to the initialized datetime variable to convert.
2. *output* is a pointer to the buffer that receives the string for the value in *d*.
3. *len* is the length of the *output* buffer.
4. *fmt* is a pointer to a buffer containing the formatting mask.
   The default date format conforms to the standard ANSI SQL format: `%Y-%m-%d %H:%M:%S`

Returns:

Code	Description
0	The conversion was successful.
<0	The conversion failed.

---

ifx_dttofmtasc()

Purpose:

To convert a datetime value to a character string, specifying the format

Syntax:

```
mint ifx_dttofmtasc(dtime_t *d, char *output, mint len, char *fmt, char c);
```

Notes:

1. *d* is a pointer to the initialized datetime variable to convert.
2. *output* is a pointer to the buffer that receives the string for the value in *d*.
3. *len* is the length of the *output* buffer.
4. *fmt* is a pointer to a buffer containing the formatting mask.
   The default date format conforms to the standard ANSI SQL format: `%Y-%m-%d %H:%M:%S`
5. *c* is one of CENTURY characters, which determines which century to apply to the year portion of the date.

Returns:

Code	Description
0	The conversion was successful.
<0	The conversion failed.

---

incvasc()

Purpose:

To convert an ANSI-standard character string to an interval value

Syntax:

```
mint incvasc(char *str, intrvl_t *i);
```

Notes:

1. *str* a pointer to a buffer containing an ANSI-standard INTERVAL string.
2. *i* is a pointer to an initialized interval variable.

Returns:

Code	Description
0	The conversion was successful.
<0	The conversion failed.
-1260	It is not possible to convert between the specified types.
-1261	Too many digits in the first field of datetime or interval.

---

incvfmtasc()

Purpose:

To convert a character string having the specified format to an interval value

Syntax:

```
mint incvfmtasc(char *input, char *fmt, intrvl_t *intvl);
```

Notes:

1. *input* is a pointer to the string to convert.
2. *fmt* is a pointer to the buffer containing the formatting mask to use for the input string.
   It must be either in year to month, or in day to fraction ranges.
3. *intvl* is a pointer to the initialized interval variable.

Returns:

Code	Description

0	The conversion was successful.
<0	The conversion failed.

---

intoasc()

Purpose:

To convert an interval value to an ANSI-standard character string

Syntax:

```
mint intoasc(intrvl_t *i, char *str);
```

Notes:

1. *i* is a pointer to the initialized interval variable.
2. *str* is a pointer to the buffer containing the ANSI-standard interval string.

Returns:

Code	Description
0	The conversion was successful.
<0	The conversion failed.

---

intofmtasc()

Purpose:

To convert an interval value to a character string, specifying the format

Syntax:

```
mint intofmtasc(intrvl_t *i, char *output, mint len, char *fmt);
```

Notes:

1. *i* is a pointer to an initialized interval variable to convert.
2. *output* is a pointer to the buffer that receives the string for the value in i.
3. *strlen* is the length of the outbuf buffer.
4. *fmt* is a pointer to the buffer containing the formatting mask.
   It must be either in year to month, or in day to fraction ranges.

Returns:

Code	Description
0	The conversion was successful.
<0	The conversion failed.

---

invdivdbl()

Purpose:

To divide an interval value by a numeric value

Syntax:

```
mint invdivdbl(intrvl_t *iv, double dbl, intrvl_t *ov);
```

Notes:

1. iv is a pointer to an initialized interval variable to be divided.
2. *dbl* is a numeric divisor value, which can be a positive or negative number.
3. *ov* is a pointer to an interval variable with a valid qualifier in the year to month class or the day to fraction(5) class.
4. Both the *iv* and *ov* qualifiers must belong to the same qualifier class

Returns:

Code	Description
0	The division was successful.
<0	The division failed.
-1200	A numeric value is too large (in magnitude).
-1201	A numeric value is too small (in magnitude).
-1202	The *dbl* parameter is zero (0).

---

invdivinv()

Purpose:

To divide one interval value by another

Syntax:

```
mint invdivinv(intrvl_t *i1, intrvl_t *i2, double *res);
```

Notes:

1. *i1* is a pointer to an initialized interval variable that is the dividend.
2. *i2* is a pointer to an initialized interval variable that is the divisor.
3. *res* is a pointer to the double value that is the quotient.
4. The qualifiers for *i1* and *i2* must belong to the same interval class, either year to month or day to fraction(5).

Returns:

Code	Description
0	The division was successful.
<0	The division failed.
-1200	A numeric value is too large (in magnitude).
-1201	A numeric value is too small (in magnitude).
-1266	An interval value is incompatible with the operation.
-1268	A parameter contains an invalid interval qualifier.

---

invextend()

Purpose:

To copy an interval value *i* to the interval value *o*, adding or dropping fields based on the qualifier of *o*

Syntax:

```
mint invextend(intrvl_t *i, intrvl_t *o);
```

Notes:

1. *i* a pointer to the initialized interval variable to extend.
2. *o* is a pointer to the interval variable with a valid qualifier to use for the extension.

Returns:

Code	Description
0	The conversion was successful.
<0	The conversion failed.
-1266	An interval value is incompatible with the operation.
-1268	A parameter contains an invalid

	interval qualifier.

---

invmuldbl()

Purpose:

To multiply an interval value by a numeric value

Syntax:

```
mint invmuldbl(intrvl_t *iv, double dbl, intrvl_t *ov);
```

Notes:

1. *iv* is a pointer to the interval variable to multiply.
2. *dbl* is the numeric double value, which can be a positive or negative number.
3. *ov* is a pointer to the resulting interval variable containing a valid qualifier.
4. Both *iv* and *ov* must belong to the same interval class, either year to month or day to fraction(5).

Returns:

Code	Description
0	The multiplication was successful.
<0	The multiplication failed.
-1200	A numeric value is too large (in magnitude).
-1201	A numeric value is too small (in magnitude).
-1266	An interval value is incompatible with the operation.
-1268	A parameter contains an invalid interval qualifier.

Formatting Directives

Numeric Formatting Mask

A numeric-formatting mask specifies a format to apply to some numeric value.

This mask is a combination of the following formatting directives:

Character	Description
*	This character fills with asterisks any

positions in the display field that would otherwise be blank.

&	This character fills with zeros any positions in the display field that would otherwise be blank.
#	This character changes leading zeros to blanks. Use this character to specify the maximum leftward extent of a field.
<	This character left-justifies the numbers in the display field. It changes leading zeros to a null string.
,	This character indicates the symbol that separates groups of three digits (counting leftward from the units position) in the whole-number part of the value. By default, this symbol is a comma. You can set the symbol with the DBMONEY environment variable. In a formatted number, this symbol appears only if the whole-number part of the value has four or more digits.
.	This character indicates the symbol that separates the whole-number part of a money value from the fractional part. By default, this symbol is a period. You can set the symbol with the DBMONEY environment variable. You can have only one period in a format string.
-	This character is a literal. It appears as a minus sign when the expression is less than zero. When you group several minus signs in a row, a single minus sign floats to the rightmost position that it can occupy; it does not interfere with the number and its currency symbol.
+	This character is a literal. It appears as a plus sign when the expression is greater than or equal to zero and as a minus sign when expr1 is less than zero. When you group several plus signs in a row, a single plus or minus sign floats to the rightmost position that it can occupy; it does not interfere with the number and its currency symbol.
(	This character is a literal. It appears as a left parenthesis to the left of a negative number. It is one of the pair of accounting

	parentheses that replace a minus sign for a negative number. When you group several in a row, a single left parenthesis floats to the rightmost position that it can occupy; it does not interfere with the number and its currency symbol.
)	This is one of the pair of accounting parentheses that replace a minus sign for a negative value.
$	This character displays the currency symbol that appears at the front of the numeric value. By default, the currency symbol is the dollar sign ($). You can set the currency symbol with the DBMONEY environment variable. When you group several dollar signs in a row, a single currency symbol floats to the rightmost position that it can occupy; it does not interfere with the number.

Any other characters in the formatting mask are reproduced literally in the result.

Examples:

Mask	Numeric value	Formatted String
-##,###.##	-12345.67	-12,234.67
	12345.67	b12,345.67
	113.11	bbbb113.11
##,###.##	-12345.67	12,345.67
	12345.67	12,345.67
--,---.&&	-445.67	bb-445.67
$$,$$$.&&	2345.67	$2,345.67
	445.67	bb$445.67

Date Formatting Mask

A date-formatting mask specifies a format to apply to some date value.

This mask is a combination of the following formatting directives:

Character	Description

dd	Day of the month as a two-digit number (01 through 31)
ddd	Day of the week as a three-letter abbreviation (Sun through Sat)
mm	Month as a two-digit number (01 through 12)
mmm	Month as a three-letter abbreviation (Jan through Dec)
yy	Year as a two-digit number (00 through 99)
yyyy	Year as a four-digit number (0001 through 9999)
ww	Day of the week as a two-digit number (00 for Sunday, 01 for Monday, 02 for Tuesday … 06 for Saturday)

Any other characters in the formatting mask are reproduced literally in the result.

Datetime Formatting Mask

A datetime-formatting mask specifies a format to apply to some datetime value.

This mask is a combination of the following formatting directives:

Character	Description
	Date related directives
%a	Identifies abbreviated weekday name as defined in locale.
%A	Identifies full weekday name as defined in locale.
%b	Identifies abbreviated month name as defined in locale.
%B	Identifies full month name as defined in locale.
%C	Identifies century number (year divided by 100 and truncated to an integer)
%d	Identifies the day of the month (01 to 31). Single digit is preceded by zero.
%D	Identifies commonly used date format (%m/%d/%y).
%e	Identifies the day of the month as a number (1 to 31). Single digit is preceded by space.
%h	Same as %b.
%iy	Identifies the year as a 2-digit number (00 to 99).
%iY	Identifies the year as a 4-digit number (0000 to

	9999).
%m	Identifies the month as a number (01 to 12). Single digit is preceded by zero.
%w	Identifies the weekday as a number (0 to 6), where 0 is the locale equivalent of Sunday.
%x	Identifies a special date representation that the locale defines.
%y	Identifies the year as a 2-digit number (00 to 99).
%Y	Identifies the year as a 4-digit number (0000 to 9999).
Time related directives	
%c	Identifies special date/time representation that locale defines.
%Fn	Identifies value of the fraction of a second, with precision specified by integer *n*. Range of *n* is 0 to 5.
%H	Identifies the hour as 24-hour clock integer (00-23).
%I	Identifies the hour as 12-hour clock integer (00-12).
%M	Identifies the minute as an integer (00-59).
%p	Identifies A.M. or P.M. equivalent as defined in locale.
%r	Identifies commonly used time representation for a 12-hour clock.
%R	Identifies commonly used time representation for a 24-hour clock (%H:%M).
%S	Identifies the second as an integer (00-61). Second can be up to 61 instead of 59 to allow for the occasional leap second and double leap second.
%T	Identifies commonly used time format (%H:%M:%S).
%X	Identifies commonly used time representation as defined in the locale.
Specials	
%%	Identifies the % character.
%n	Identifies a new-line character.
%t	Identifies a TAB character.

Any other characters in the formatting mask are reproduced literally in the result.

Interval Formatting Mask

An interval-formatting mask specifies a format to apply to some interval value.

This mask must be combination of Class 1 interval or Class 2 interval formatting directives:

Character	Description
Class 1 formatting directives (YEAR to MONTH)	
%C	Identifies century number (year divided by 100 and truncated to an integer)
%iy	Identifies the year as a 2-digit number (00 to 99).
%iY	Identifies the year as a 4-digit number (0000 to 9999).
%m	Identifies the month as a number (01 to 12). Single digit is preceded by zero.
%y	Identifies the year as a 2-digit number (00 to 99).
%Y	Identifies the year as a 4-digit number (0000 to 9999).
Class 2 formatting directives (DAY to FRACTION)	
%a	Identifies abbreviated weekday name as defined in locale.
%A	Identifies full weekday name as defined in locale.
%d	Identifies the day of the month (01 to 31). Single digit is preceded by zero.
%e	Identifies the day of the month as a number (1 to 31). Single digit is preceded by space.
%Fn	Identifies value of the fraction of a second, with precision specified by integer *n*. Range of *n* is 0 to 5.
%H	Identifies the hour as 24-hour clock integer (00-23).
%I	Identifies the hour as 12-hour clock integer (00-12).
%M	Identifies the minute as an integer (00-59).
%S	Identifies the second as an integer (00-61). Second can be up to 61 instead of 59 to allow for the occasional leap second and double leap second.
%w	Identifies the weekday as a number (0 to 6), where 0 is the locale equivalent of Sunday.
Specials	
%%	Identifies the % character.
%n	Identifies a new-line character.
%t	Identifies a TAB character.

Genero FESQLC

- Overview
- Prerequisites
- Embedding SQL Statements
- Using Host Variables
  - Character variables
  - Decimal variables
  - Interval and datetime variables
- Using Data structures
  - Arrays
  - Structures (struct)
  - Type definitions (typedef)
  - Function parameters
  - Pointers
- Using Indicator Variables
- Using Database Cursors
- Using Dynamic SQL
  - Preparing and Executing SQL Statements
  - Using Input parameters in Dynamic SQL
  - Using database cursors with Prepared statements
  - Freeing Prepared statements
  - Optimizing Prepared statements
- Supported Data Types
- Compiling with fesqlc
- Using Preprocessor Directives
  - Including files - **include**
  - Creating FESQLC macros - **define, undefine**
  - Compiling conditionally - **ifdef, ifndef, endif, else, elif**
- Handling Exceptions
  - Using SQLSTATE
  - Using SQLCODE
  - Using WHENEVER
  - Using GET DIAGNOSTICS
  - Example program
- Migration Notes

See also: Implementing C Extensions

Overview

Genero FESQLC allows you to embed SQL statements in your C-language programs in order to communicate with a relational database. The keywords "**EXEC SQL**" (preferred ANSI standard) or the **$** symbol precede SQL statements and FESQLC pre-processor directives.

Host variables can substitute for literal values in your programs. The host variable stores the values retrieved from a database table or serves as a parameter for SQL statements.

You can create dynamic SQL statements that are constructed at runtime, based on some program conditions or the user's interaction.

FESQLC supports the common SQL data types, but some, such as DECIMAL, do not have corresponding C data types. FESQLC provides additional data types that you can use in your programs.

The **fesqlc** tool compiles and links C programs that contain **FESQLC** source code files, creating an executable C program.

Exception handling in FESQLC allows you to obtain information about the execution of an SQL statement and to handle program errors.

The FESQLC preprocessor directives allow you to:

- include additional files in your FESQLC program
- create compile-time definitions
- specify conditional compilation

Prerequisites

Before you begin using Genero FESQLC, make sure the following has been done:

- FESQLC and a C compiler are installed on your system.
- The database client software is installed.
- The environment variables required by the database client software are set.
- The PATH environment variable includes **$FGLDIR/bin**, the bin directory of the software installation directory.

Embedding SQL statements

SQL statements that communicate with a relational database can be embedded in your program.

Statements must be preceded with the keywords "EXEC SQL" (preferred ANSI standard) or the $ symbol:

```
EXEC SQL CONNECT TO "testdb";

EXEC SQL CREATE TABLE t1 (
  k2 integer NOT NULL PRIMARY KEY,
  t1 date DEFAULT TODAY NOT NULL,
```

```
  c char(10)
);
EXEC SQL DELETE FROM customer WHERE store_num = '101';
```

Case sensitivity

The following components of an embedded SQL statement are case-sensitive:

- **Names of host variables**. The variable **ordernum** is not the same as **Ordernum**. FESQLC considers these to be two different variables.
- **Data types**. If you use the data type **INT** in a variable declaration it would not be recognized - **int** is the correct name. See Supported data types for the names of the FESQLC data types.
- **Cursor names and statement names**

Usually, the values in variables are case-sensitive. All other components are not case-sensitive.

Escape characters and quotation marks

Both C and FESQLC use the backslash character \ as the escape character, to specify that the following character is to be considered literal and not interpreted. FESQLC allows you to enclose strings in either single or double quotes. Special care must be taken when the WHERE clause of your embedded SQL statement contains a backslash or quotation mark.

- To search for a backslash character, escape the special significance of the backslash with the \ escape character: For example, to search for the company name **The \\ Store:**

  ```
  EXEC SQL DECLARE c1 CURSOR FOR SELECT * FROM customer WHERE
  store_name = 'The \\\\ Store';
  ```

  The first backslash causes the second backslash to be interpreted literally, and the third backslash causes the fourth backslash to be interpreted literally, resulting in the desired "The \\ Store".

- ANSI standards do not allow you to use the same quotation character in an expression as the string delimiter and as a literal. Therefore, to search for a value containing an embedded single quote, enclose the value with double quotes, and use the \ escape character to cause the single quote to be interpreted literally. For example, to search for the company name **The ' Store**:

  ```
  EXEC SQL DECLARE c1 CURSOR FOR SELECT * FROM customer WHERE
  store_name = "The \' Store";
  ```

Comments

You can use the standard C comment indicator in your FESQLC statements:

```
EXEC SQL DELETE FROM customer;  /* deletes all rows */
```

Using Host Variables

Host variables are C variables that you can use in SQL statements as if they were literal values. Host variables can store:

- Parameter values for SQL statements
- Result set values fetched from the database
- Cursor or statement identifiers

Variable name

As in C, the host variable name can consist of letters, digits, and underscores, consistent with the requirements of your C compiler. Begin the name with a letter rather than an underscore, to avoid potential conflicts with internal FESQLC names.

```
int x;
char str1[11];
```

A variable name is case-sensitive; **firstname** is not the same as **FirstName**.

Variable declaration

Choose a data type for the host variable compatible with the SQL data type of the database column, if the variable is used to transfer data between the program and the database.

See Supported Data Types for a list of the SQL data types and the compatible FESQLC or C data types.

You can also use data structures such as arrays, struct, and pointers as host variables in your program.

Declare the host variable in your program, using the same syntax that you use to declare a C variable. Put the statements in an FESQLC declare section preceded with the keywords EXEC SQL BEGIN DECLARE SECTION, and terminate it with EXEC SQL END DECLARE SECTION (preferred ANSI standard):

```
EXEC SQL BEGIN DECLARE SECTION;
    int x;
    char str1[11];
    char str4[11];
EXEC SQL END DECLARE SECTION;
```

For backward compatibility with Informix ESQL/C, the FESQLC compiler also supports host variable declaration with the $ symbol:

```
$int x;
$char str1[11];
```

The variable can be declared only once within a C block.

Data conversion

We recommend that you use the FESQLC data type that corresponds to the SQL data type of the database column being referenced in your statements.

When necessary, FESQLC will try to convert the data type of a value if a discrepancy exists in your program.

Scope

The *scope* of an FESQLC host variable is the same as that of a C variable, which is dependent on the placement of the declaration within the file:

- Host variables declared within a program block or declared by a function are **local**, accessible only within that block or function.
- Host variables declared outside a function are **modular**, accessible from all program blocks that occur after that point, until the end of the file.
- If a local host variable has the same identifier as an external variable, the local variable takes precedence inside the program block in which it is declared.

Initialization

FESQLC allows you to use normal C initializer expressions with host variables:

```
EXEC SQL BEGIN DECLARE SECTION;
  int var1 = 128;
  string var2[21] = "A test string";
EXEC SQL END DECLARE SECTION;
```

Local datetime or interval variables are automatically initialized to set the datetime or interval qualifier; local variables of any other data type have an undefined value and must always be initialized before use.

```
EXEC SQL BEGIN DECLARE SECTION;
  datetime year to second dt;
  interval hour to second t1_iv1;
EXEC SQL END DECLARE SECTION;
```

The FESQLC preprocessor does not check the validity of initialization statements; the C compiler handles this.

Host variables in use

Precede the variable with an indicator (:) whenever it is used in an embedded SQL statement:

```
EXEC SQL BEGIN DECLARE SECTION;
   varchar firstname[31];
   int stateid;
   varchar statename[51];
EXEC SQL END DECLARE SECTION;
EXEC SQL select fname into :firstname from custtab;
EXEC SQL insert into statetab values (:stateid,
:statename);
```

Character variables

Although you can use **char/varchar/string/fixchar** data types for host variables in your program, we recommend that you match the data type of the host variable to the data type of the corresponding database column.

Genero C Extensions provide character and string functions to manipulate character data.

Declaring Size

The **char/varchar/string** host variables include the string terminator; you must allow for the terminator when declaring one of these variables. For example, to correspond to a database CHAR data type, declare the size of the **char** host variable as (dblength+1).

```
  EXEC SQL BEGIN DECLARE SECTION;
  char p_state_code[3]  /* length of the column state.state_code is 2
*/
```

Although **char**, **varchar**, and **string** host variables contain null terminators, FESQLC never inserts these characters into a database column.

Using char/varchar Pointers

It is strongly recommended that you use fixed-size **char** and **varchar** variables instead of pointers. The problem with **char/varchar** pointers is that the database interface cannot determine the size of the **char/varchar** input parameter or fetch buffer, which is required for binding host variables to SQL statements.

Host variables defined as **char** or **varchar** pointers cannot be used to fetch data. If you must use a **char/varchar** pointer to insert data, use an sqlda structure and specify the exact size of the corresponding CHAR/VARCHAR SQL type in `sqlvar->sqllen`.

Fetching and inserting CHAR data

When a value from a CHAR database column is fetched into a **char** host variable, FESQLC pads the value with trailing blanks up to the size of the host variable, leaving one space for the null terminator. If the column data being fetched does not fit into the character host variable, FESQLC truncates the data, setting the SQLSTATE variable to 01004, and setting the value of any indicator variable to the size of the character data in the column.

When inserting a value into a CHAR database column, FESQLC pads or truncates the value to the size of that column, if necessary.

Fetching and inserting VARCHAR data

When a value from a VARCHAR database column is fetched into a **char** host variable, FESQLC truncates and null terminates the value if the source is longer, and sets any indicator variable; if the destination is longer, FESQLC pads the value with trailing spaces and null terminates it.

When a character value is inserted into a VARCHAR database column, any trailing spaces are not counted.

Using the fixchar data type

The **fixchar** data type is a character string data type that does not have a null terminator. When a value from a CHAR or VARCHAR column is fetched into a fixchar, FESQLC pads the value with trailing blanks.

Do not use **fixchar** to insert data into a VARCHAR column. Even if the length of the data is shorter than the **fixchar**, trailing blanks will be stored by the database server.

Do not copy a null-terminated C string into a **fixchar** variable. When the variable value is inserted into the database column, the null character at the end of the string will also be inserted, complicating subsequent searches for the data value.

Using the string data type

The FESQLC **string** data type holds character data terminated by a null character, and does not have trailing blanks.

When values from a CHAR/VARCHAR column are fetched into a **string** host variable, the value is usually stripped of trailing blanks and null-terminated. However, if the value is a string of blanks (" "), a single blank and the null-terminator are stored in the **string** host variable, to distinguish an empty string from a null string.

Decimal Variables

FESQLC supports the **dec_t** and **decimal** data types to handle DECIMALs. Although you can use the **dec_t** data type to fetch decimal data, it cannot be used to insert data as there is no precision or scale; we recommend the use of the FESQLC **decimal** data type, which supports the DECIMAL and MONEY SQL data types, for inserting and fetching data.

A **decimal** host variable must be defined with a precision and scale:

```
EXEC SQL BEGIN DECLARE SECTION;
   decimal(6,2) dec4;
```

Genero C Extensions provide decimal functions to manipulate decimal values. Use only decimal functions on decimal data types; otherwise, you may get unpredictable results.

Interval and Datetime Variables

The **interval** FESQLC data type encodes a span of time. The **datetime** FESQLC data type encodes a specific point in time. The accuracy of the data types is specified by a qualifier (year to second, etc.) Declare host variables for INTERVAL or DATETIME database columns using these FESQLC data types:

```
EXEC SQL BEGIN DECLARE SECTION;
  datetime year to second dtm2;
  interval hour(5) to second inv2;
EXEC SQL END DECLARE SECTION;
```

A **datetime** or **interval** data type is stored as a decimal number with a scale of zero and a precision dependent on the qualifier used when defining the data type.

Datetime and **interval** host variables must have the qualifier initialized. This is done by using datetime or **interval** data types as shown in the above example. If you use **dtime_t** or **intrvl_t** data types, use macros to initialize their qualifiers:

```
EXEC SQL BEGIN DECLARE SECTION;
   dtime _t dtml;
   intrvl_t inv1;
EXEC SQL END DECLARE SECTION;

dtml.dt_qual = TU_DTENCODE(TU_YEAR, TU_SECOND);
invl.in_qual = TU_TENCODE(5, TU_HOUR, TU_SECOND);
```

Genero C Extensions provide macros for the qualifiers, and functions to manipulate these data types.

Using data structures

Arrays

You can declare an *array* of host variables in your program; provide the number of elements in the array:

```
EXEC SQL BEGIN DECLARE SECTION;
  int ordid[20];
 EXEC SQL END DECLARE SECTION;
```

An array can be one-dimensional or two-dimensional.

Your program can reference an element of the array as follows:

```
EXEC SQL DELETE FROM ordertab WHERE order_num = :ordid[1];
```

Structures (struct)

You can declare a C language structure *struct* in your program, specifying the members of the structure:

```
EXEC SQL BEGIN DECLARE SECTION;
  struct {
    int ordid;
    date orddate;
    char ordname[26];
  } order_rec;
EXEC SQL END DECLARE SECTION;
```

Here struct **order_rec** defines a host variable with three members: **ordid**, **orddate**, and **ordname**.

The individual members of the structure are referenced in your program as *structname.membername*:

```
EXEC SQL insert into ordertab (order_num, order_date)
    values (:order_rec.ordid :order_rec.orddate);
```

If an SQL statement allows a list of host variables, you can specify the structure name and FESQLC will expand it:

```
EXEC SQL insert into ordertab values (:order_rec);
```

The values list in this SQL statement will expand to order_rec.ordid, order_rec.orddate, order_rec.ordname.

Type definitions (typedef)

FESQLC supports C typedef statements and allows the use of typedef names in the declaration of the types of host variables:

```
EXEC SQL BEGIN DECLARE SECTION;
typedef struct {
     int key;
     varchar name[21];
     datetime year to fraction(5) tstamp;
} mytype;
mytype myrecord;
EXEC SQL END DECLARE SECTION;
```

You cannot use a typedef statement that names a multidimensional array, a union, or a function pointer as the type of a host variable.

Function parameters

You can declare host variables as function parameters; precede the variable name with the PARAMETER keyword.

```
func1(ord_id, ord_date)
EXEC SQL BEGIN DECLARE SECTION;
  PARAMETER int ord_id;
  PARAMETER date ord_date;
EXEC SQL END DECLARE SECTION;
{
  ...
  EXEC SQL INSERT INTO oldorders VALUES (:ord_id,
:ord_date);
  ...
}
```

Notes:

- Parameters declared for host variables must be part of a function header only.
- You can have other function parameters that are not used as host variables inside the EXEC SQL declare section.

Pointers

Host variables can be declared as pointers to SQL identifiers, prepared statements for example:

```
EXEC SQL BEGIN DECLARE SECTION;
    char *stmt;
    int ord_num;
EXEC SQL END DECLARE SECTION;
...
stmt = "SELECT order_num FROM orders";
```

```
EXEC SQL PREPARE curs1 FROM :stmt;
EXEC SQL OPEN curs1;
EXEC SQL FETCH curs1 INTO :ord_num;
```

Host variables declared as pointers can also be used as input parameters in SQL insert statements. However, we strongly recommend that you use fixed-size char and varchar host variables instead of char/varchar pointers to hold SQL CHAR/VARCHAR data.

```
EXEC SQL BEGIN DECLARE SECTION;
  int *v_id;
  char v_name[26];
EXEC SQL END DECLARE SECTION;
...
INSERT INTO vendors values (:v_id, :v_name);
```

Using Indicator variables

You can specify an indicator variable in your SQL statement to obtain information about a value returned by the statement.

- Declare the indicator variable using the normal FESQLC syntax, and specify an appropriate data type; integer data types are used for indicators that detect null values or truncated values.

  ```
  EXEC SQL BEGIN DECLARE SECTION;
     char p_state_code[3];
     int i_state_code;
  EXEC SQL END DECLARE SECTION;
  ```

- Use the indicator variable in the SELECT statement. Specify the indicator variable to be associated with the host variable, preceding the indicator variable with the word INDICATOR:

  ```
  :p_state_code INDICATOR :i_state_code
  ```

  Or, separate the variables with a symbol (: or $). There can be one or more whitespaces between the host variable name and the indicator name:

  ```
  :p_state_code :i_state_code
  $p_state_code$i_state_code
  $p_state_code:i_state_code
  ```

  Example:

  ```
  EXEC SQL SELECT state_code INTO :p_state_code INDICATOR
  :i_state_code
       FROM state WHERE state_name = 'Illinois';
  ```

When FESQLC returns the value into the host variable, it will also set the corresponding indicator variable. Your program can check the values of any indicator variables before continuing.

To detect null values

If a database table column permits nulls, you may have a null value returned to the corresponding host variable specified in your SQL SELECT statement. If one of the values in a database column referenced for an aggregate function is null, you may also have a null value returned. Null (unknown) values can cause problems for your program. Use indicator variables to determine whether any values returned to the host variables are null:

- FESQLC will set the value of an indicator variable to **zero** if the value returned for the associated host variable is not null.
- If the value returned is null, FESQLC will set the indicator variable to **-1**.

Checking whether the variable is null before printing it:

```
printf("%s\n", i_state_code == 0 ? p_state_code : "<no code>");
```

Tip: The NULL keyword of an INSERT statement allows you to insert a null value into a table row. As an alternative, you can use a negative indicator variable with the host variable. Set the value of the indicator variable to **-1** when it is declared.

To detect truncated values

If the host variable referenced by an SQL SELECT statement is a character array, the value returned from the database may be truncated (the value returned is too large to fit into the array). Use indicator variables to determine whether any values returned are truncated.

- FESQLC will set the value of an indicator variable to **zero** if the value returned for the associated host variable is not truncated.
- If the value returned was truncated, FESQLC will set the value of the indicator variable to the original size in bytes of the host variable.

Example program using indicator variables:

```
01 #include <stdio.h>
02
03 int main()
04 {
05  EXEC SQL BEGIN DECLARE SECTION;
06  char p_state_code[3];
07  int i_state_code;
08  EXEC SQL END DECLARE SECTION;
09
10  EXEC SQL CONNECT TO "custdemo";
```

```
11  EXEC SQL SELECT state_code INTO :p_state_code INDICATOR
:i_state_code
12    FROM state
13    WHERE state_name = 'Illinois';
14
15  printf("%-20s\n ",
16         i_state_code == 0 ? p_state_code : "<no name>"
17  );
18  EXEC SQL DISCONNECT ALL;
19 }
```

Using database cursor statements

You can use sequential, scroll, hold, update, or insert cursors in FESQLC programs. For example, if a SELECT statement will retrieve more than one row:

- **Declare** the cursor for the select statement. This allocates storage to hold the cursor.
- **Open** the cursor. The active set associated with the cursor is identified, and the cursor is positioned before the first row of the set.
- **Fetch** the data into the host variable(s), until the last desired row is fetched.
- **Close** the cursor when the fetch is complete. This releases the active result set associated with the cursor. A closed cursor can be re-opened.
- **Free** the cursor when it is no longer needed by the program. A freed cursor must be declared again before it can be re-opened and used.

The cursor program statements must appear physically within the module in the order listed.

Example program using a database cursor:

```
01 #include <stdio.h>
02
03 EXEC SQL define NAME_LEN 20;  /* customer.store_name is a CHAR(20)
*/
04 EXEC SQL define ADDR_LEN 20;  /* customer.addr is a CHAR(20)
*/
05 EXEC SQL define STATE_LEN 2;  /* customer.state_code is a CHAR(2)
*/
06
07 int main()
08 {
09
10  EXEC SQL BEGIN DECLARE SECTION;
11    int p_num;
12    varchar p_name[ NAME_LEN + 1 ];
13    int2 i_name;
14    varchar p_addr[ ADDR_LEN + 1 ];
15    int2 i_addr;
16    varchar p_addr2[ ADDR_LEN + 1 ];
17    int2 i_addr2;
```

```
18    char p_state_code[STATE_LEN + 1];
19  EXEC SQL END DECLARE SECTION;
20
21  printf( "Connecting...\n\n");
22  EXEC SQL CONNECT TO 'custdemo';
23
24  EXEC SQL DECLARE c1 CURSOR FOR        /* Declaring the cursor */
25    SELECT store_num, store_name, addr, addr2
26     FROM customer
27     WHERE state = :p_state;
28
29  strcpy(p_state, "IL");
30  EXEC SQL OPEN c1;                  /* Opening the cursor */
31
32  for (;;)                           /* loop to fetch the rows */
33  {
34   EXEC SQL FETCH c1 INTO :p_num,
35    :p_name INDICATOR :i_name,
36    :p_addr INDICATOR :i_addr,
37    :p_addr2 INDICATOR :i_addr2;
38
39   if (strncmp(SQLSTATE, "02", 2) == 0) {
40     /* No more rows */
41     break;
42   }
43
44   printf("%6d %-20s\n %s %s\n",
45     p_num,
46     i_name == 0 ? p_name : "<no name>",
47     i_addr == 0 ? p_addr : "<no address>",
48     i_addr2 == 0 ? p_addr2 : ""
49   );
50  }
51
52  EXEC SQL CLOSE c1;     /* close the cursor */
53  EXEC SQL FREE c1       /* free the cursor  */
54
55  printf("\nDisconnecting...\n\n");
56  EXEC SQL DISCONNECT CURRENT;
57
58  return 0;
59 }
```

The code in lines 39-41 checks the SQLSTATE global variable to determine whether there are any more rows to retrieve. See Handling Exceptions

See Dynamic SQL for additional examples using cursors and the PREPARE statement.

Note: The FETCH ... INTO ... syntax is supported by all databases. Declaring a cursor for SELECT ... INTO ... is Informix-specific syntax, and not recommended for portability.

Using Dynamic SQL

Dynamic SQL statements allow you to construct an SQL statement at runtime, based on some program conditions or the user's interaction.

1. Your program assembles the SQL statement in a character string, and assigns it to a character string host variable.

   The SQL statement string cannot contain the names of any host variables. Input parameters (indicated by question mark placeholders in the statement string) can be used in the statement string, for example in a WHERE clause, to indicate that the value needed will be provided at a given position in the SQL statement.

2. Your program uses the prepare statement to assign a statement id and send the SQL statement string contained in the host variable to the database server for parsing. You can check the global variables SQLSTATE or SQLCODE to find out whether errors occurred in the parsing.

3. Your program executes the prepared statement one or more times.

   - If input parameters are used, the USING clause of the EXECUTE or OPEN instruction provides the values.
   - If the SQL statement produces a result set (like SELECT), or inserts multiple rows with an INSERT CURSOR, you must declare a cursor with the prepared statement handle in order to execute it.

4. When the prepared statement is no longer needed, your program frees the statement to release the allocated resources.

You can use Dynamic SQL for any SQL statement except SQL management instructions (such as PREPARE, DECLARE, OPEN, EXECUTE, FETCH, CLOSE, FREE), SQL connection instructions (such as CONNECT, DATABASE, DISCONNECT) and transaction control instructions (BEGIN WORK, COMMIT WORK, ROLLBACK WORK, SET ISOLATION, SET LOCK MODE).

Preparing and Executing SQL statements

Preparing statements

In order to prepare an SQL statement, you must do the following:

1. Declare a char host variable to hold the entire SQL string:

```
EXEC SQL BEGIN DECLARE SECTION;
  char sqlstring[128];    /* SQL statement */
EXEC SQL END DECLARE SECTION;
```

2. Assign the SQL string to the host variable:

```
strcpy(sqlstring, "DELETE FROM orders WHERE store_num IS
NULL");
```

3. Use the PREPARE statement to create a valid SQL statement from the string contained in the host variable:

```
EXEC SQL PREPARE stid FROM :sqlstring;
```

Once an SQL statement is prepared, use EXECUTE to execute the statement; the statement can be executed often as needed.

Executing prepared statements that do not return values (INSERT, UPDATE, DELETE)

Follow the steps described at the beginning of this section to prepare the statement. Then, use the EXECUTE instruction to perform the execution of the statement:

```
strcpy(sqlstring, "DELETE FROM orders WHERE store_num IS
NULL");
EXEC SQL PREPARE stid FROM :sqlstring;
EXEC SQL EXECUTE stid;
```

Executing prepared statement that return values (SELECT)

If the prepared SQL statement returns a single row, you can use the INTO clause with EXECUTE to specify the host variables to receive the database column values:

```
sprintf(sqlstring, "SELECT COUNT(*) FROM orders");
EXEC SQL PREPARE stid FROM :sqlstring;
EXEC SQL EXECUTE stid INTO :p_order_count;
```

You can use indicator variables with the EXECUTE statement, to determine if a value returned from the database is null:

```
EXEC SQL EXECUTE stid INTO :p_custname INDICATOR
:i_custname;
```

If the prepared statement returns more than one row, you must declare a database cursor and use cursor management statements instead of the EXECUTE statement. See Using cursors with prepared statements.

Executing dynamic SQL statements without parameters

If there are no input parameters, and the statement will not be executed multiple times, you can reduce the number of statements to be executed by using the EXECUTE IMMEDIATE statement to prepare and execute the statement string at once:

```
sprintf(sqlstring, "DELETE FROM orders WHERE store_num IS NULL");
EXEC SQL EXECUTE IMMEDIATE :sqlstring;
```

Example program using EXECUTE IMMEDIATE and also returning a single row:

```
01 #include <stdio.h>
02
03 int main()
04
05 {
06  EXEC SQL BEGIN DECLARE SECTION;
07    char sqlstring[128];   /* string to hold sql statement */
08    int p_store_num;       /* values for store_num column */
09    char p_store_name[21]; /* values for store_name, column length 20
*/
10    int2 i_store_name;      /* indicator variable for p_store_name */
11  EXEC SQL END DECLARE SECTION;
12
13  char whereinput[64];   /* variable for string containing where
clause */
14  strcpy(whereinput, "store_num > 200"); /* setting value of
whereinput */
15
16  EXEC SQL CONNECT TO 'custdemo';
17
18  /* Using EXECUTE IMMEDIATE  */
19  sprintf(sqlstring, "DELETE FROM orders WHERE %s", whereinput);
20  EXEC SQL EXECUTE IMMEDIATE :sqlstring;
21
22  /* Returning a single row - USING PREPARE and EXECUTE */
23  strcpy(whereinput, "store_num = 101");  /* setting value of
whereinput */
24  sprintf(sqlstring, "SELECT store_name FROM customer WHERE %s",
whereinput);
25  EXEC SQL PREPARE stid FROM :sqlstring;
26  EXEC SQL EXECUTE stid INTO :p_store_name INDICATOR :i_store_name;
27  /* Displaying store name */
28  printf("%-20s\n", i_store_name == 0 ? p_store_name : "<no name>");
29
30  EXEC SQL DISCONNECT CURRENT;
31
32  return 0;
33 }
```

Using Input Parameters in dynamic SQL

Input parameters are placeholders in an SQL statement string for host variables that contain the values for expressions. The actual values are provided at runtime. Placeholders are used since the database server cannot parse a dynamic SQL statement that contains host variable names.

Placeholders can be used anywhere within the SQL statement where an expression is valid. You cannot use an input parameter to represent an identifier such as a database name, a table name, or a column name.

Steps to use input parameters

1. Declare a host variable for each input parameter. The data type of the host variable must be compatible with the data type of the database column referenced in the SQL statement string.
2. Assemble the SQL statement string using ? as a placeholder for each input parameter.
3. PREPARE the statement string to create a valid SQL statement.
4. Provide the host variables corresponding to the input parameters with the USING clause of the EXECUTE instruction. The order in which the variables are listed must match the order in which the variables appear in the prepared statement.

Example using input parameters with a prepared DELETE statement:

```
EXEC SQL PREPARE stid FROM "DELETE FROM orders WHERE
store_num = ? AND order_num < ?";
EXEC SQL EXECUTE stid USING :p_store_num, :p_order_num;
```

Example using input parameters with a prepared SELECT statement that returns a single row:

```
EXEC SQL PREPARE stid FROM "SELECT COUNT(*) FROM orders
WHERE order_num < ?";
EXEC SQL EXECUTE stid USING :p_order_num INTO :p_count;
```

If the SELECT statement returns more than one row, a database cursor must be used.

Example program using input parameters:

```
01 #include <stdio.h>
02
03 int main()
04 {
05  EXEC SQL BEGIN DECLARE SECTION;
06    char sqlstring[128];
07    char p_state_code[3];       /* column length is 2  */
08    char p_state_name[16];      /* column length is 15 */
09  EXEC SQL END DECLARE SECTION;
10
11  EXEC SQL CONNECT TO 'custdemo';
```

```
12
13  /* Using placeholders for values to be inserted  */
14  strcpy(sqlstring, "INSERT INTO state VALUES ( ?, ? )");
15  EXEC SQL PREPARE stid2 FROM :sqlstring;
16  printf(" SQLSTATE = [%s]\n\n", SQLSTATE); /* checking SQL success
*/
17
18  strcpy(p_state_code, "AZ");
19  strcpy(p_state_name, "Arizona");
20
21  /* Executing the prepared statement */
22  EXEC SQL EXECUTE stid2 USING :p_state_code, :p_state_name;
23  printf(" SQLSTATE = [%s]\n\n", SQLSTATE);
24
25  EXEC SQL DISCONNECT CURRENT;
26
27  return 0;
28 }
```

Using a Database Cursor with Prepared statements

If an SQL SELECT statement generates a set of rows, you must handle the result set with a Database Cursor.

First, the cursor is declared for the prepared statement:

```
EXEC SQL PREPARE stid FROM "SELECT order_num FROM orders WHERE
store_num = ?";
EXEC SQL DECLARE c1 CURSOR FOR stid;
```

If input parameters are used in the WHERE clause, the USING clause of the OPEN cursor instruction provides the host variables containing the values needed:

```
EXEC SQL OPEN c1 USING :p_store_num;
```

The FETCH statement uses the INTO clause to retrieve the database values into host variables. This statement can be executed multiple times until there are no more values to be retrieved:

```
EXEC SQL FETCH c1 INTO :p_order_num;
```

The order in which the host variables are listed must match the order of the select list.

CLOSE the cursor when the fetch is complete. A closed cursor can be re-opened.

```
      EXEC SQL CLOSE c1;
```

When the cursor is no longer needed, it is FREEd, releasing the memory associated with it. Once freed, a cursor must be re-declared before it can be used again.

```
    EXEC SQL FREE c1;
```

Example program using a database cursor and prepared statement:

```
01 #include <stdio.h>
02
03 /* To check for SQL errors */
04 void errlog(void)
05 {
06  fprintf(stderr, "Error occurred:\n");
07  fprintf(stderr, " SQLSTATE = [%s] SQLCODE = %d\n\n",
08                  SQLSTATE, SQLCODE);
09  EXEC SQL DISCONNECT ALL;
10  exit(1);
11 }
12
13 int main()
14 {
15  EXEC SQL BEGIN DECLARE SECTION;
16    int p_store_num;        /* store_num column */
17    int p_order_num;        /* order_num column */
18    int2 i_order_num;       /* indicator for p_order_num */
19  EXEC SQL END DECLARE SECTION;
20
21  EXEC SQL WHENEVER ERROR CALL errlog; /* Set the error-handling */
22
23  EXEC SQL CONNECT TO 'custdemo';
24
25  EXEC SQL PREPARE stid FROM "SELECT order_num FROM orders WHERE
store_num = ?";
26  EXEC SQL DECLARE c1 cursor for stid;
27  p_store_num = 12;  /* Assign a value to store number */
28  EXEC SQL OPEN c1 USING :p_store_num;
29
30  for (;;)
31  {
32   EXEC SQL FETCH c1 INTO :p_order_num INDICATOR :i_order_num;
33   if (strncmp(SQLSTATE, "02", 2) == 0) break;
34   printf("%6d \n",
35          i_order_num == 0 ? p_order_num : "<no order number>"
36   );
37  }
38
39  EXEC SQL CLOSE c1;
40  EXEC SQL FREE c1;
41
42  EXEC SQL DISCONNECT CURRENT;
43
44  return 0;
45 }
```

Inserting multiple rows

Use an *insert cursor* to perform buffered row insertion in database tables. The insert cursor simply inserts rows of data; it cannot be used to fetch data.

A cursor is declared for the prepared statement, using input parameters for the values to be inserted:

```
EXEC SQL PREPARE s1 FROM "INSERT INTO manufact VALUES (?,?)"
EXEC SQL DECLARE c1 CURSOR FOR s1;
```

The insert cursor is opened:

```
EXEC SQL OPEN c1;
```

The PUT cursor instruction uses the USING clause to insert values from the host variables into the database columns. This statement is executed multiple times until there are no more values to be inserted.

```
EXEC SQL PUT c1 FROM :faccode, :facname;
```

Flush data rows, if required:

```
EXEC SQL FLUSH c1;
```

Close the cursor:

```
EXEC SQL CLOSE c1;
```

Free the statement handle and the cursor:

```
EXEC SQL FREE s1;
EXEC SQL FREE c1;
```

Freeing Prepared Statements

When the prepared statement is no longer needed, your program can free the statement to release the resources allocated:

```
EXEC SQL PREPARE stid FROM "SELECT order_num FROM orders
WHERE store_num = ?";
...
EXEC SQL FREE stid;
```

If a database cursor is used for the prepared statement, both the cursor and the statement can be freed:

```
EXEC SQL FREE c1;
EXEC SQL FREE stid;
```

Optimizing Prepared Statements

If an SQL statement is to be executed multiple times, you can reduce thetraffic between the client application and the database server if the statement is not also parsed and optimized each time. Use the PREPARE statement to parse the statement outside of the program loop, so the parsing only occurs once:

```
EXEC SQL PREPARE stid FROM "DELETE FROM orders WHERE
store_num = ?"
```

Then, use the EXECUTE ... USING statement inside the program loop.

```
EXEC SQL EXECUTE stid USING :p_store_num;
```

Supported Data Types

When your program accesses a database column you must declare a host variable of the appropriate FESQLC or C data type to hold the data.

Recommended Data Types for Host Variables

The table below lists the relationship between supported SQL data types, and the recommended FESQLC host variable data types that correspond to the SQL types. Also included are the corresponding data type constants, required by some FESQLC functions:

SQL Data Type	C Data Type	Description	Type Constant
char(*n*), character(*n*)	char(*n+1*)	character data, padded with blanks; null-terminated, specify	CCHARTYPE
	string(*n+1*)	database column length plus 1	CSTRINGTYPE
		character data, no trailing blanks; null-terminated, specify database column length plus 1	
date	date	4-byte integer representing the date	CDATETYPE
datetime	datetime (*qualifiers*)	calendar date and time of day; must specify accuracy as a qualifier such as *year to day*	CDTIMETYPE
decimal, dec,	decimal(*p,s*)	fixed point number; *p*	CDECIMALTYPE

numeric		(precision) and *s* (scale) must be defined	
money	decimal(*p,s*)	fixed point number; *p* (precision) and *s* (scale) must be defined	CMONEYTYPE
float, double precision	double	double-precision value with up to 17 significant digits	CDOUBLETYPE
integer, int	int, long	4-byte integer	CINTTYPE
interval	interval(*qualifiers*)	a span of time; must specify accuracy as a qualifier such as *day to second*	CINVTYPE
serial	int, long	4-byte integer	CINTTYPE
smallfloat, real	float	single-precision value with up to 9 significant digits	CFLOATTYPE
smallint	short	2-byte integer	CSHORTTYPE
varchar(m,x)	varchar(*m+1*)	variable length character data; null-terminated, specify maximum length plus 1	CVARCHARTYPE
	string(*n+1*)		CSTRINGTYPE
		character data, no trailing blanks; null-terminated, specify database column length plus 1	

See Using Host Variables for additional information about the use of these data types.

Additional FESQLC data types

These data types are not recommended for use in database operations, but may be used in FESQLC functions.

FESQLC Data Type	Description	Data Type Constant
fixchar(n)	character data, padded with blanks, no null terminator	CFIXCHARTYPE
dec_t	decimal value; cannot have precision or scale	CDECIMALTYPE
dtime_t	datetime value; cannot have qualifiers	CDTIMETYPE
intrvl_t	interval value; cannot have qualifiers	CINVTYPE

Specific-Length Data Types

The following data types automatically map correctly for 32-bit and 64-bit platforms. Some FESQLC functions use these data types instead of int, short, and long.

FESQLC Data Type	Description
int1	one-byte integer
int2	two-byte integer
int4	four-byte integer
mint	native integer data type for machine
mlong	native long integer data type for the machine
MSHORT	native short integer data type for the machine
MCHAR	native char data type for the machine

SQL Data Types not supported

SQL Data Type	Description
blob	binary large object; binary data in an undifferentiated byte stream
boolean	data type with values limited to TRUE, FALSE, and NULL
byte	binary data in an undifferentiated byte stream
clob	character large object; text data
int8	8-byte integer
lvarchar	character data of varying length, no larger than 2 kilobytes
list	a collection of elements that can be duplicate values and have ordered positions
multiset	a collection of elements that can be duplicate values and have no ordered positions
opaque	user-defined data type
row	complex data type with one or more members called fields
serial8	8-byte serial data type
set	a collection of elements that are unique values and have no ordered positions
text	any kind of text data

Compiling with fesqlc

The **fesqlc** tool compiles and links C programs that contain **Genero FESQLC** source code files, creating an executable C program. An **FESQLC** source file must be preprocessed before a C compiler can compile it. By default, your **FESQLC** source files are passed to the **FESQLC** preprocessor, and then to the C compiler. You can choose to preprocess only.

Syntax:

```
fesqlc [options] source.ec [othersrc.ec ...]
        [othersrc.c ...] [otherobj.o ...] [otherlib.a ...]
```

Notes:

1. *options* are described below.
2. The **FESQLC** source code files must have the extension **.ec**.
3. *othersrc.*c are C source files to be linked.
4. *otherobj*.o are C object files to be linked.
5. *otherlib.*a are C static libraries to be linked.

Options:

Option	Description
`-V`	Display version information
`-h`	Display this help
`-v`	Verbose mode (display information messages)
`-e`	Preprocess only
`-G`	No line numbers (for debugging purposes)
`-c`	Compile to object file
`-o name`	Output file specification
`-ED name`	Define a preprocessor macro for FESQLC macros
`-EU name`	Un-define a preprocessor macro for FESQLC macros
`-D name`	Define a preprocessor macro for C macros
`-U name`	Un-define a preprocessor macro for C macros
`-I path`	Specify a path for C and FESQLC includes
`-W option`	Warnings: *option* can be one

	of: *all*: enable all warnings
`-cpf "flag .."`	C compiler flags
`-lkf "flag .."`	Linker flags
`-usl "flag .."`	Utility and system libraries

Usage:

Options can be used for preprocessing only, or for preprocessing/compiling/linking.Options are global and affect all files.

The **-V** (display version information) and **-h** (display help) options do not require source file names:

```
fesqlc -V
fesqlc -h
```

The **-e** option suppresses compiling and linking of the source file. The file will be preprocessed by FESQLC and output as a C source code file (*filename*.c).

```
fesqlc -e mysource.ec
```

The **-c** option preprocesses the source file, and then compiles it to object code (*filename*.o). By default the '**cc**'compiler is used on Unix. You can modify the C compiler by setting the FGLCC environment variable. On Windows, the default is '**cl.exe**'.

```
fesqlc -c mysource.ec
```

Use the **-o** option to specify the output file name for the executable file that is the result of preprocessing, compiling and linking.

```
fesqlc -o myprog file1.ec
```

Use the option **-ED *name*** to define a global FESQLC macro. This has the same effect as an FESQLC define preprocessor directive at the top of your program.

```
fesqlc -ED CHECKED file2.ec
```

Use the option **-EU *name*** to un-define an FESQLC macro, removing it globally from the entire program.

```
fesqlc -EU CHECKED file2.ec
```

Use the **-D** and **-U** options only to define and un-define C macros for your program.

Example:

```
fesqlc -o prog1 -ED DEBUG -I ./include -d ora920 \
        file1.ec file2.ec file3.ec
```

In this example the executable output file will be named **prog1**, a global macro named **DEBUG** is defined, the include path is specified as **./include**, and the database type is **Oracle 9.20**. Files to be processed include **file1.ec**, **file2.ec** and **file3.ec**.

Creating a C Extensions written in ESQL/C

In Genero V 2, runtime system C extensions must be created as shared libraries. For more details, see C Extensions.

Steps:

1. Include the **fglExt.h** file to the **.ec** files implementing C functions called from Genero BDL.
   ```
   #include <f2c/fglExt.h>
   ```
2. Create the extension interface as described in C Extensions: C interface file.
3. Compile the extension interface to object files with you C compiler.
4. Compile all the **.ec** sources to object files with **fesqlc**, by using the **-c** option.
5. Create the shared library (OS specific command) from all the object files. Warning: You must link with the **libfesqlc** and **libfgl** libraries provided in FGLDIR/lib.
6. Declare the extension module in the 4gl source code by using the IMPORT instruction:
   ```
   IMPORT extension
   MAIN
     ...
   ```

Example:

This example uses three sources:

- The ESQL/C module doing some SQL.
- The extension interface file defining the list of C extension functions.
- The Genero program connecting to the database and calling the C Extension function of the ESQL/C module.

```
-- The module.ec source ----------------------------------
#include <f2c/fglExt.h>
exec sql include sqlca;

int insert_row(int c)
{
   exec sql insert into dbit2 values (1, 'aaaa', 'bbbbb');
   return 0;
}
```

```
-- The myext.c source ------------------------------------

#include "f2c/fglExt.h"

int insert_row(int);

UsrFunction usrFunctions[]={
  { "insert_row", insert_row, 0, 0 },
  { 0, 0, 0, 0 }
};

-- The prog.4gl source -----------------------------------

IMPORT myext
MAIN
   DATABASE stores
   CALL insert_row()
END MAIN
```

First, we link the extension module. This creates the object file (**module.o**):

```
fesqlc -c module.ec
```

Then, create the shared library from the compiled object file, by using the **libfesqlc** library:

Linux example:

```
gcc -static-libgcc -Xlinker --no-undefined -shared \
   -o myext.so \
   module.o \
   -L$FGLDIR/lib -lfesqlc -lfgl
```

MS Visual C 7.1 example (you must create a manifest file starting from VC 8.0):

```
link /DLL /O:myext.dll \
   %FGLDIR%\lib\libfesqlc.lib %FGLDIR%\lib\libfgl.lib
```

You could also create the shared library directly from **fesqlc**, by using the **-l** option:

```
fesqlc -o myext.so \
   -l "gcc -static-libgcc -Xlinker --no-undefined -shared" \
   source1.ec source2.ec source3.ec source4.ec \
   extinterface.c
```

Preprocessor Directives

The **fesqlc** compiler supports FESQLC preprocessor directives. This allows you to *include* other FESQLC files, *define macros* that can be used later in the code and use

ifdef conditional directives to compile only some part of the code, as you can do with the C preprocessor (cpp):

```
EXEC SQL include "myheader.h";
EXEC SQL define LEN 15;
EXEC SQL ifdef DEBUG;
    ...
EXEC SQL endif;
```

Warning: FESQLC macros must end with a semi-colon, like all FESQLC statements.

Standard C preprocessing

Standard C preprocessing takes effect after FESQLC preprocessing, during the C compilation step. In order to include system header files such as **stdio.h**, you must use the standard C **#include** directive. If C macros are defined, you must use standard C **#ifdef** directives. FESQLC will ignore any C macro definitions during the FESQLC preprocessing step.

Including files - the include directive

The **include** directive allows you to include other files into your FESQLC programs. You must use the FESQLC include directive if the file contains embedded SQL statements or other FESQLC statements.

The file will be read into the program at the location of the include directive. Use the keywords **EXEC SQL** (preferred ANSI standard) or the **$** symbol to precede the directive. Specify the exact name, including any extension, of the file:

```
EXEC SQL include "def_constants.h";
EXEC SQL include "/prog/def_constants.h";
```

If the filename includes the path, you must enclose the filename in quotationmarks. Otherwise, you may omit the quotation marks, but FESQLC will convert the filename to lowercase. If you omit the path, FESQLC will search the preprocessor path for the file.

To avoid the need for recompilation if a file location changes, omit the path from the filename and use the **-I** FESQLC compiler option to specify the path:

```
fesqlc -I ./myincludes prog1.ec
```

Note: Genero FESQLC automatically includes any header files that it requires in order to preprocess your program. Use the standard C #include directive to include C header files. The #include of C includes a file after FESQLC preprocessing.

Warning:

For migration purposes, **include** statements for Informix ESQL/C header files are permitted, although they are not required and the FESQLC

compiler will ignore them. However, if your program has such statements, you must use the EXEC SQL syntax. The following C syntax is not supported:

```
#include "sqlca.h"
```

FESQLC will consider this a normal C header file to be included. The statement must be replaced by:

```
EXEC SQL include sqlca;
```

Creating FESQLC macros - the define, undef directives

The FESQLC **define** directive allows you to create simple macros that are available only to the FESQLC preprocessor. The FESQLC preprocessor, rather than the C preprocessor, processes this directive.

Use the keywords **EXEC SQL** (preferred ANSI standard) or the **$** symbol to precede the directive:

```
EXEC SQL define CHECKED;       -- macro without a value
EXEC SQL define LEN 15;        -- macro with an integer
value of 15
EXEC SQL define NAME "scott";  -- macro with a string value
```

The scope of the macro is from the location of the **define** directive until the end of the file, or until the macro is removed. The FESQLC **undef** directive allows you to remove the macro:

```
EXEC SQL undef LEN;
```

You can use the FESQLC compiler options **-ED** and **-EU** to override these FESQLC macro definitions.

Note: C macros are defined and undefined using the #define and #undef C preprocessor directives in your source code, or the -D and -U FESQLC preprocessor options.

Macros can be used in conjunction with other **FESQLC** preprocessor directives to control conditional processing of a file.

Compiling conditionally - the ifdef,ifndef, endif, else, elif directives

FESQLC provides directives to conditionally compile a program. The directives test whether a macro has been created with an FESQLC define directive or the **-ED** FESQLC compiler option, and then process the file accordingly.

- **ifdef** - instructs FESQLC to compile the statements that follow, if the specified macro is defined.
- **ifndef** - instructs FESQLC to compile the statements that follow, if the specified macro is not defined.
- **endif** - indicates the close of an **ifdef** or **ifndef** condition.
- **else** - provides alternative processing instructions for an **ifdef** or **ifndef** condition.
- **elif** - is the same as "else if define", used to provide alternative processing instructions; instructs FESQLC to compile the statements that follow, if the specified macro has been defined.

Use the keywords **EXEC SQL** (preferred ANSI standard) or the **$** symbol to precede the directive:

```
EXEC SQL ifdef CHECKED;
EXEC SQL DELETE FROM cust_temp;  /* executed when CHECKED
is defined */
EXEC SQL endif;
EXEC SQL ifdef CHECKED;
EXEC SQL DELETE FROM cust_temp;
EXEC SQL else;  /* if CHECKED is not defined */
printf("no delete, CHECKED is not defined");
EXEC SQL endif;
EXEC SQL ifdef CHECKED;
EXEC SQL DELETE FROM cust_temp;
EXEC SQL elif PROBLEMS;  /* if CHECKED not defined and
PROBLEMS defined */
printf("no delete, PROBLEMS is defined");
EXEC SQL endif;
```

Note: There is no equivalent to the "if" C directive in FESQLC; the preprocessor supports only the **ifdef** and **ifndef** statements that test for the existence of a macro.

Example:

If your program uses both FESQLC macros and C macros, you can combine the -D and -ED preprocessor options in the same command line. The following lines in the FESQLC program **prog.ec** has two blocks of preprocessing directives. The first block uses FESQLC syntax and the second block uses the standard C syntax:

```
EXEC SQL ifdef CHECKED;
printf("CHECKED is defined");
EXEC SQL else;
printf("CHECKED is not defined");
EXEC SQL endif;

#ifdef MAXLEN
printf("MAXLEN is defined");
#else
printf("MAXLEN is not defined");
#endif
```

If you combine FESQLC **-ED** and **-D** command line options like this:

```
fesqlc -ED CHECKED -D MAXLEN prog.ec
```

FESQLC will create the FESQLC macro CHECKED, and will create the C macro MAXLEN.

When the program is run, the following lines would be executed:

```
printf("CHECKED is defined");
printf("MAXLEN is defined");
```

Handling Exceptions

Unless you explicitly include exception-handling code in your FESQLC application, the program continues when an SQL error occurs. Uncontrolled SQL errors may cause your program to have unexpected behavior.

Suggested exception-handling strategies are:

- Check after each SQL statement by testing the value of SQLSTATE (or SQLCODE).
- Use the WHENEVER statement to specify the action to be taken when an SQL statement is not successful.
- Use GET DIAGNOSTICS to obtain additional information about any exceptions.

Using SQLSTATE

Warning: Using SQLSTATE is the preferred ANSI method of checking for exceptions. However, not all database servers do support SQLSTATE. You must check the database server documentation to verify if SQLSTATE is supported.

This global variable defined by FESQLC returns the status of the SQL statement most recently executed. The status can be:

- Success
- Success, but no rows found
- Success, but warnings generated
- Failure, runtime error generated

FESQLC automatically copies the status information from the diagnostics area into the SQLSTATE global variable; you do not have to define the variable, and it is accessible anywhere in your program.

SQLSTATE can contain only digits and capital letters. The first two characters of the five-character SQLSTATE string indicate the class of the execution code; the remaining characters provide additional information. Check the first two characters to determine whether the most recently executed SQL statement was successful:

Class (first 2 chars)	Description
00	SQL statement was successful
01	SQL statement was successful, with warnings
02	SELECT or FETCH statement resulted in NOT FOUND condition
>02	SQL statement resulted in a runtime error

The following code checks SQLSTATE for the NOT FOUND case:

```
if (strncmp(SQLSTATE, "02", 2) == 0) {
  break;
}
```

If an SQL error occurred, you can get specific information from the other three characters of SQLSTATE. The characters 0 to 4 and A to H are used for the SQLSTATE class codes in ANSI and X/OPEN standard implementations:

First two characters	Remaining characters	Description
00	000	SQL statement was successful
01	000	SQL statement was successful, with warnings:
01	002	Warning: Disconnect error, transaction rolled back
01	003	Warning: Null value eliminated in set function
01	004	Warning: String data right truncated
01	005	Insufficient item descriptor areas
01	006	Privilege not revoked
02	000	SELECT or FETCH statement resulted in NOT FOUND condition
>02		SQL statement resulted in a runtime error
0A	000	Feature not supported.
0A	001	Multiple database server transactions
21	000	Cardinality violation
22	000	Data exception
22	001	String data, right truncation

22	002	Null value, no indicator parameter
22	003	Numeric value out of range
22	012	Division by zero
22	024	Un-terminated string
23	000	Integrity-constraint violation
24	000	Invalid cursor state
2B	000	Dependent privilege descriptors still exist
2D	000	Invalid transaction termination
2E	000	Invalid connection name
33	000	Invalid SQL descriptor name
34	000	Invalid cursor name
3C	000	Duplicate cursor name
40	000	Transaction rollback
40	003	Statement completion unknown
42	000	Syntax error or access violation

Check your database server documentation for additional server-specific implementations of SQLSTATE codes.

Using SQLCODE

SQLCODE is a global variable defined by FESQLC to determine the success of the most recently executed SQL statement.

SQLCODE holds the Informix SQL error code. Using SQLSTATE is the preferred method; SQLCODE is provided for backwards compatibility.

The database server returns information about the execution of an SQL statement to the SQL Communications Area (sqlca) C structure. FESQLC copies the value of the sqlca.sqlcode field to the SQLCODE global variable. Your program can check the value in the SQLCODE variable for the following information:

Value	Description
0	SQL statement was successful
100	SELECT or FETCH statement resulted in NOT FOUND condition
<0	SQL statement resulted in a runtime error. The number specifies the particular Informix error number.

Native SQL error codes

If an SQL error occurs, the database server specific error code is available in **sqlca.sqlerrd[1]**.

Warnings

If a warning was generated by the SQL statement, sqlca.sqlwarn.sqlwarn0 is set to "W". To test for warnings, check the first warning field. If this indicates the database server has generated a warning, you can check the values of the other fields in **sqlca.sqlwarn** to identify the specific condition.

See your database server documentation for additional information about specific errors reported in SQLCODE, and the sqlca structure.

Using WHENEVER

Your program can use the WHENEVER statement to trap exceptions that occur during the execution of SQL statements, and to specify the action to be taken when a specific class of error occurs:

```
WHENEVER <exception class> <exception action>
```

Exception classes:

- SQLERROR - exception occurs when an SQL statement has failed.
- SQLWARNING - exception occurs when an SQL statement has generated a warning.
- NOT FOUND - exception occurs when the NOTFOUND condition is returned by an SQL statement.

Using WHENEVER SQLERROR statements allows your program to react to errors and warnings, in the same way as checking the SQLSTATE or SQLCODE variables.

Actions to be taken:

- CONTINUE - the program ignores the exception and continues with the next instruction in the program. **This is the default**.
- STOP - the program stops executing immediately when an exception occurs.
- CALL *<function name>*- the specified function in the program is called if an exception occurs.
- GOTO *<label>* - control is transferred to the program code identified by the specified label. The program code specified by GOTO *<label>* must be included in every program block that contains SQL statements, or another WHENEVER statement in that program block must re-set the condition.

Examples:

```
WHENEVER SQLERROR STOP;
```

```
EXEC SQL CONNECT TO mydb;
...
WHENEVER SQLERROR GOTO error_handling_code;
EXEC SQL DELETE FROM orderstab WHERE cust_num = "101";
...
```

The default action is CONTINUE if no WHENEVER condition is set. The actions for the exception classes can be set independently.

The GET DIAGNOSTICS statement

The diagnostics area is an internal structure that the database server updates after the execution of each SQL statement. The GET DIAGNOSTICS statement can provide additional information about the most recently executed SQL statement.

Statement fields return overall information about the most recently executed SQL statement. For example, you can determine the number of exceptions generated by an SQL statement, indicating whether you need to seek additional information.

Syntax:

```
EXEC SQL GET DIAGNOSTICS :host-variable = field-name [,...]
;
```

The keywords for the statement fields of the diagnostic area, and the data types of the returned information, are:

Field name	Data type	Information type	Description
NUMBER	mint	Statement	Number of exceptions generated. Even a successful execution will generate one exception.
MORE	char[2]	Statement	Contains Y or N plus a null terminator; Y indicates that the diagnostics area contains all the exceptions information. N indicates that there is more information than the database server can store in the diagnostics area.
ROWCOUNT	mint	Statement	Number of rows that a successful SELECT, UPDATE, or DELETE

	statement has affected. ROWCOUNT for a SELECT statement is undefined.

GET DIAGNOSTICS can return multiple exceptions; each exception that is generated has a number. Exception number 1 is the exception generated by the last SQL statement executed in your program. No set order exists for any additional exceptions that are generated. To get information about a particular exception, use the following syntax:

```
EXEC SQL GET DIAGNOSTICS EXCEPTION exception-number
             :host-variable = field-name [,...] ;
```

Some of the keywords for the exception fields associated with a specific exception, and the data types of the information, are:

Field name	Data type	Information type	Description
RETURNED_SQLSTATE	char[6]	Exception	Status of the exception. FESQLC automatically copies the information from this field for exception 1 to the SQLSTATE global variable.
INFORMIX_SQLCODE	int4	Exception	Contents of sqlca.sqlcode; FESQLC automatically copies the information from this field to the SQLCODE global variable.
MESSAGE_TEXT	char[8191]	Exception	Variable length character string that describes the exception.
MESSAGE_LENGTH	mint	Exception	Number of characters that are in the message stored in

			MESSAGE_TEXT.
SERVER_NAME	char[255]	Exception	Variable length character string holding the name of the database server used in the most recent connection statement; will be blank if there is no server connected.
CONNECTION_NAME	char[255]	Exception	Variable length character string holding the name of the connection associated with the most recent connection statement; will be blank if there is no current connection.

See the SQL documentation for your database server for additional information about exception fields in the diagnostics area.

To return information about your SQL statement and any exceptions to your program:

First, declare a host variable in your program to contain the name of the field in the diagnostics area that you want to access:

```
EXEC SQL BEGIN DECLARE SECTION;
  mint numrows;
  mint numerr;
  char messagetext[8191];
  mint messagelength;
EXEC SQL END DECLARE SECTION;
```

To get statement information after the SQL statement has executed, specify the diagnostics area field name. For example, to get the number of rows updated, inserted, or deleted by the SQL statement, and the number of exceptions generated:

```
EXEC SQL DELETE FROM orders WHERE store_num = 111;
EXEC SQL GET DIAGNOSTICS :numrows = ROWCOUNT, :numerr = NUMBER;
```

Use the exception clause to get information about a specific exception generated by your SQL statement. Exception 1 is the exception associated with the last SQL statement executed.

```
EXEC SQL GET DIAGNOSTICS EXCEPTION 1 :messagetext =
MESSAGE_TEXT;
```

You do not have to use GET DIAGNOSTICS to check the RETURNED_SQLSTATE and INFORMIX_SQLCODE fields for exception 1, since FESQLC copies the values from those fields to the SQLSTATE and SQLCODE global variables.

Note: The GET DIAGNOSTICS statement does not change the contents of the diagnostics area.

Migration Notes

- Including Informix ESQL/C header files
- Using C-style macros for ESQL/C host variables
- Using char and varchar pointers
- Database and database object names
- Decimal host variables
- Literal constants in USING clause
- Comment indicator
- Features not supported

Including Informix ESQL/C header files

For migration purposes, include statements for Informix ESQL/C header files are permitted, although they are not required and the FESQLC compiler will ignore them. However, if your program has such statements, you must use the EXEC SQL syntax. The following C syntax is not supported:

```
#include "sqlca.h"
```

FESQLC will consider this a normal C header file to be included. The statement must be replaced by:

```
EXEC SQL include sqlca;
```

Using C-style macros for ESQL/C host variables

Old Informix ESQL/C compilers allowed you to use C-style macros in the declaration of host variables:

```
#define MAXLEN 15

$char myvar[MAXLEN];
```

This is not standard ESQL/C programming. Recent Informix ESQL/C compilers raise a warning in such a case:

```
Warning -33208: Runtime error is possible because size of 'myvar' is
unknown.
```

FESQLC does not support the usage of C-style macros in SQL host variables declarations. You must use ESQL/C macros instead:

```
EXEC SQL define MAXLEN 15;

$char myvar[MAXLEN];
```

Existing code using C-style macros may use the C macro in normal C variable declarations. Is such a case you can define the macro twice: Once for the C variables and once for the ESQL/C host variables:

```
#define MAXLEN 15
EXEC SQL define MAXLEN 15;

char myvar1[MAXLEN];
$char myvar2[MAXLEN];
```

This solution is valid ESQL/C programming. It works with both Informix ESQL/C and Genero FESQLC compilers.

Using char and varchar pointers

Normally you use char/varchar arrays to hold SQL CHAR or VARCHAR data:

```
$char cn[101];
$SELECT cust_name INTO $cn FROM customer ... ;
```

Informix ESQL/C also supports char/varchar pointers, but the length of the C variable is unknown.

```
$char *cn;
$SELECT cust_name INTO $cn FROM customer ... ;
```

The char/varchar pointers are typically used for function parameters:

```
int func(p1, p2)
EXEC SQL BEGIN DECLARE SECTION;
int p1;
char *p2;
EXEC SQL BEGIN DECLARE SECTION;
{
EXEC SQL INSERT INTO tab VALUES ( :p1, :p2 );
}
```

or as constants:

```
$const char *blank = " ";
$const char *undef = "Undefined value";
```

Database APIs must know the size of the SQL char/varchar type corresponding to the host variable, to check for overflow. (For example, you cannot insert a CHAR(100) value into a CHAR(9) in Oracle).

Char and varchar pointers can be used as input parameters in SQL statements, but the FESQLC API computes the size with an strlen() of the current value pointed by sqlvar->sqldata.

It is strongly recommended that you use a fixed size of char and varchars. If you MUST use a char/varchar pointer, use an sqlda structure and specify the exact size of the corresponding CHAR/VARCHAR SQL type in sqlvar->sqllen.

Host variables defined as char or varchar pointers cannot be used to fetch data:

```
$char *c;
$varchar *v;
$select c1, c2 into :c, :v from t;
```

Database and database object names

EXEC SQL DATABASE supports only a simple database name; for portability the Informix database specification ***dbname*@*server*** is not supported.

The database prefix in a table name specification (***dbname*:*tabname***) is not supported, for portability.

Using SQL keywords as database object names is not supported.

Decimal host variables

Decimal host variables must be declared with precision and scale if you want to use a database other than Informix:

```
$decimal(6,2) mydec;
```

In an **sqlda** structure, **sqlvar** elements defining a DECIMAL value must have the **sqllen** member initialized with the decimal dimension (i.e., PRECMAKE(precision,scale). This is different from Informix ESQL/C, where **sqllen** has the size of the **dec_t** structure for input parameters.

Literal constants in USING clause

Literal constants are not allowed in a USING clause:

```
OPEN cursor USING 123, "abc"
```

You must always use host variables.

Comment Indicator

The double hyphen (--) comment identifier is permitted within an SQL statement only.

```
$SELECT * -- all columns        compiles successfully
    FROM tab;
$SELECT * FROM tab;
    -- all columns              will not compile
```

Features Not Supported

- **DESCRIBE INTO sqlda** is not yet supported.
- System descriptors (**ALLOCATE DESCRIPTOR**, etc) are not yet supported.
- **INT8** and **SERIAL8** data types are not yet supported.
- **CLOB/BLOB** data types are not yet supported.
- **Multi-byte** character sets are not yet supported.
- **Opaque** and **LVARCHAR** data types will not be supported.
- Server communication functions (**sqlbreak sqlexit)** are not supported.
- **GLS** library functions will not be supported.
- **Collections** will not be supported.

Example program

```
01 #include <stdio.h>
02
03 EXEC SQL define NAME_LEN 20;  /* customer.store_name is a CHAR(20)
*/
04 EXEC SQL define ADDR_LEN 20;  /* customer.addr is a CHAR(20)
*/
05
06 void errlog(void)
07 {
08  fprintf(stderr, "Error occurred:\n");
09  fprintf(stderr, "  SQLSTATE = [%s]  SQLCODE = %d\n\n",
10                  SQLSTATE, SQLCODE);
11  EXEC SQL DISCONNECT ALL;
12  exit(1);
13 }
14
15 int main()
16 {
17  EXEC SQL BEGIN DECLARE SECTION;
18    int  p_num;
19    varchar p_name[ NAME_LEN + 1 ];
20    int2 i_name;
21    varchar p_addr[ ADDR_LEN + 1 ];
22    int2 i_addr;
23    varchar p_addr2[ ADDR_LEN + 1 ];
24    int2 i_addr2;
25    char p_state[3];
```

```
26  EXEC SQL END DECLARE SECTION;
27
28  EXEC SQL WHENEVER ERROR CALL errlog; /* set error handling */
29
30  printf( "Connecting...\n\n");
31  EXEC SQL CONNECT TO 'custdemo';
32
33  EXEC SQL DECLARE c1 CURSOR FOR
34    SELECT store_num, store_name, addr, addr2
35      FROM customer
36      WHERE state = :p_state;
37
38  strcpy(p_state, "IL");
39  EXEC SQL OPEN c1;
40
41  for (;;)
42  {
43
44   EXEC SQL FETCH c1 INTO :p_num,
45                          :p_name INDICATOR :i_name,
46                          :p_addr INDICATOR :i_addr,
47                          :p_addr2 INDICATOR :i_addr2;
48
49   if (strncmp(SQLSTATE, "02", 2) == 0) {
50        /* No more rows */
51        break;
52   }
53
54   printf("%6d %-20s\n      %s %s\n",
55            p_num,
56            i_name == 0 ? p_name : "<no name>",
57            i_addr == 0 ? p_addr : "<no address>",
58            i_addr2 == 0 ? p_addr2 : ""
59   );
60  }
61
62  EXEC SQL CLOSE c1;
63  EXEC SQL FREE c1;
64
65  printf("\nDisconnecting...\n\n");
66  EXEC SQL DISCONNECT CURRENT;
67
68  return 0;
69 }
```