

Lycia Development Suite



Lycia II Function Guide
Version 2.2 – August 2014
Revision number 3.03



Querix 'Lycia'

Function Guide

Part Number: 009-022-303-014

Querix 4GL Reference – Function Guide

Copyright 2006-2014 Querix (UK) Limited. All rights reserved.

Part Number: 009-060-303-014

Published by:

Querix (UK) Limited. 50 The Avenue, Southampton, Hampshire,
SO17 1XQ, UK

Publication history:

August 2004:	Beta edition
June 2005:	Updated for 'Hydra4GL' v 4.2
January 2008:	Updated for 'Hydra4GL' v 4.3
December 2008:	Web Services update 4.31
February 2009:	Updated for 'Hydra4GL' v 4.4
December 2010:	Updated for Lycia II
August 2014:	Updated for Lycia 2.2

Last Updated:

August 2014

Documentation written by:

Anthony Davey

Updated By Matt Davenport, Sean Sunderland, Gemma Davis, Elena Krivtsova, Svitlana Ostnek, Olga Gusarenko

Notices:

The information contained within this document is subject to change without notice. If you find any problems in the documentation please submit your comments to documentation@querix.com.

No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose without the express permission of Querix (UK) Ltd.

Other products or company names used within this document are for identification purposes only, and may be trademarks of their respective owners.

Table of Contents

IN THIS GUIDE	1
FUNCTIONS IN 4GL PROGRAMS	3
BUILT-IN 4GL FUNCTIONS.....	3
METHODS USED WITH OBJECT DATA TYPES.....	4
BUILT-IN AND EXTERNAL SQL FUNCTIONS AND PROCEDURES.....	5
C FUNCTIONS	5
ESQL/C FUNCTIONS.....	6
PROGRAMMER-DEFINED 4GL FUNCTIONS	6
INVOKING FUNCTIONS.....	8
<i>Passing Arguments and Returning Values.....</i>	<i>8</i>
<i>Invoking SQL Functions</i>	<i>8</i>
OPERATORS OF 4GL.....	9
AGGREGATE REPORT FUNCTIONS.....	12
THE GROUP KEYWORD	13
THE WHERE CLAUSE	13
THE MAX() AND MIN() FUNCTIONS.....	13
<i>max()</i>	<i>14</i>
<i>min()</i>	<i>14</i>
THE AVG() AND SUM() FUNCTIONS	15
<i>sum()</i>	<i>15</i>
<i>avg()</i>	<i>16</i>
THE COUNT (*) AND PERCENT (*) FUNCTIONS.....	16
<i>Differences between the 4GL and SQL Aggregates</i>	<i>18</i>
ARITHMETIC OPERATORS	19
UNARY ARITHMETIC OPERATORS	21
BINARY ARITHMETIC OPERATORS	21
EXPONENTIATION (**) OPERATOR.....	24
MOD	25
MULTIPLICATION (*) AND DIVISION (/) OPERATORS	26
ADDITION (+) AND SUBTRACTION (-) OPERATORS	26
SIZE OPERATOR.....	27

BOOLEAN OPERATORS.....	29
LOGICAL OPERATORS.....	30
BOOLEAN COMPARISONS.....	31
RELATIONAL OPERATORS.....	32
THE NULL TEST.....	35
LIKE	35
MATCHES	37
<i>Set Membership and Range Tests.....</i>	<i>39</i>
DATA TYPE COMPATIBILITY	40
EVALUATING BOOLEAN EXPRESSIONS	41
OPERATOR PRECEDENCE IN BOOLEAN EXPRESSIONS	41
BITWISE OPERATORS.....	44
AND (&)	44
OR ()	44
XOR (^).....	44
NOT (~)	44
DATABASE.....	47
FGL_COLUMN_INFO()	47
FGL_DRIVER_ERROR()	48
FGL_FIND_TABLE()	49
FGL_NATIVE_CODE()	50
FGL_NATIVE_ERROR()	51
MEMBERSHIP (.)	52
CURSOR.....	53
CURSOR_NAME()	53
CLOSE()	54
DECLARE()	55
FETCHING ROWS METHODS.....	56
<i>FetchNext().....</i>	<i>56</i>
<i>FetchPrevious().....</i>	<i>56</i>
<i>FetchFirst()</i>	<i>56</i>
<i>FetchLast().....</i>	<i>57</i>
<i>FetchRelative().....</i>	<i>57</i>
<i>FetchAbsolute().....</i>	<i>57</i>
FLUSH()	58

FREE()	59
GETNAME()	60
GETSTATEMENT()	61
OPEN()	62
PUT()	63
SETPARAMETERS()	64
SETRESULTS()	65
PREPARED STATEMENTS	66
EXECUTE()	66
FREE()	67
GETNAME()	68
GETSTATEMENT()	69
ISNATIVE()	70
PREPARE()	71
SETPARAMETERS()	72
SETRESULTS()	73
FILE I/O	74
CHANNEL	74
DDE FUNCTIONS	78
<i>Using DDE</i>	78
<i>DDEConnect</i>	78
<i>DDEExecute</i>	79
<i>DDEFinish</i>	79
<i>DDEFinishAll</i>	79
<i>DDEGetError</i>	79
<i>DDEPeek</i>	80
<i>DDEPoke</i>	80
FGL_BASENAME()	81
FGL_DIRNAME()	82
FGL_DOWNLOAD()	83
FGL_GRID_EXPORT()	84
FGL_GRID_VIEWEXPORT()	85
FGL_MK()	86
FGL_MKDIR()	87
FGL_RM()	88
FGL_RMDIR()	89

FGL_TEST()	90
FGL_UPLOAD()	92
BASE.CHANNEL CLASS	93
<i>Create()</i>	93
<i>SetDelimiter()</i>	93
<i>Opening a Channel</i>	94
<i>Closing a Channel</i>	96
<i>readLine()</i>	96
<i>writeLine()</i>	97
<i>isEOF()</i>	97
<i>read()</i>	97
<i>write()</i>	98
<i>Errors handling</i>	99
<i>Input/ Output formatting</i>	99
<i>Using line terminators in Read() and Write() methods</i>	100
<i>Using line terminators in ReadLine() and WriteLine() methods</i>	100
<i>Line terminators on different platforms</i>	101
INPUT.....	102
ARR_COUNT()	102
ARR_CURR()	105
FGL_BELL()	107
FGL_DIALOG_FIELDORDER()	108
FGL_DIALOG_SETCURRLINE().....	109
FGL_GETKEY()	112
FGL_KEY_QUEUE()	114
FGL_KEYNAME()	115
FGL_KEYVAL()	116
FGL_LASTKEY()	120
SCR_LINE()	123
SET_COUNT()	126
DIALOG METHODS	129
GENERAL METHODS.....	130
<i>ui.Dialog.GetCurrent()</i>	130
<i>accept()</i>	130
<i>getArrayLength()</i>	130
<i>nextField()</i>	130
<i>getCurrentRow()</i>	131
<i>setCurrentRow()</i>	131
<i>getCurrentItem()</i>	131
<i>getFieldBuffer()</i>	132

<i>getFieldTouched()</i>	132
<i>setFieldTouched()</i>	133
<i>getForm()</i>	133
<i>setActionActive()</i>	133
<i>setActionHidden()</i>	134
FIELD MANIPULATION METHODS	134
<i>setFieldActive()</i>	135
<i>insertRow()</i>	135
<i>insertNode()</i>	136
<i>appendRow()</i>	136
<i>appendNode()</i>	137
<i>deleteRow()</i>	137
<i>deleteNode()</i>	138
<i>deleteAllRows()</i>	138
<i>validate()</i>	139
<i>setDefaultUnbuffered()</i>	139
<i>setArrayAttributes()</i>	139
<i>setCellAttributes()</i>	140
<i>setSelectionMode()</i>	140
<i>isRowSelected()</i>	140
<i>setSelectionRange()</i>	141
<i>selectionToString()</i>	141
DYNAMIC AND STATIC ARRAYS METHODS	142
APPEND() / APPENDELEMENT()	142
CLEAR()	143
DELETE() / DELETEELEMENT()	144
GETSIZE() / GETLENGTH()	145
INSERT() / INSERTELEMENT()	146
RESIZE()	147
FGL_DIALOG_GETBUFFERSTART()	147
FGL_DIALOG_GETBUFFERLENGTH()	147
GUI CLIENT	148
FGL_FGLGUI()	148
FGL_GETPROPERTY()	149
<i>Read Only Properties</i>	150
FGL_GETUITYPE()	153
FGL_INIT4JS()	154
FGL_REPORT_TYPE()	155
FGL_SETPROPERTY()	157
<i>Editable Properties</i>	158

UI.INTERFACE CLASS	162
<i>GetFrontEndName()</i>	162
<i>GetFrontEndVersion()</i>	162
<i>FrontCall()</i>	162
<i>Refresh()</i>	164
GRAPHICAL THEME FILES	165
APPLY_THEME()	165
WINDOW	166
FGL_DRAWBOX()	166
FGL_DRAWLINE ()	169
FGL_GETWIN_HEIGHT	171
FGL_GETWIN_WIDTH	171
FGL_GETWIN_X	172
FGL_GETWIN_Y	172
FGL_SETSIZE() §	173
FGL_SETTITLE() §	174
FGL_WINDOW_CLEAR	175
FGL_WINDOW_CLOSE()	176
FGL_WINDOW_CURRENT()	177
FGL_WINDOW_GETOPTION()	178
FGL_WINDOW_OPEN()	180
FGL_WINMESSAGE()	181
FGL_WINPROMPT()	183
FGL_WINQUESTION()	185
FGL_WINSIZE()	187
CLEAR()	189
CLOSE()	190
DISPLAYAT()	191
ERROR()	192
MESSAGE()	193
MOVE()	194
OPEN()	195
OPENWITHFORM()	196
RAISE()	197
UI.WINDOW CLASS	198

<i>Initializing a ui.WINDOW Object</i>	198
<i>Using a ui.WINDOW Object</i>	198
<i>Example</i>	199
FORM	200
FGL_FORM_CLOSE()	200
FGL_FORM_DISPLAY()	201
FGL_FORM_OPEN()	202
FGL_GRID_HEADER()	203
FGL_SCR_SIZE()	204
CLOSE()	207
DISPLAY()	208
GETWIDTH()	209
GETHEIGHT()	210
OPEN()	211
UI.FORM CLASS	212
<i>LoadActionDefaults ()</i>	212
<i>LoadToolbar()</i>	212
<i>LoadTopMenu()</i>	212
<i>SetDefaultInitializer()</i>	213
<i>SetElementText()</i>	213
<i>SetElementImage()</i>	214
<i>SetFieldStyle()</i>	214
<i>EnsureFieldVisible()</i>	214
<i>SetFieldHidden()</i>	215
<i>Example</i>	215
UI.DRAGDROP CLASS	216
<i>SetOperation()</i>	216
<i>GetOperation ()</i>	216
<i>AddPossibleOperation ()</i>	216
<i>GetLocationRow ()</i>	217
<i>GetLocationParent ()</i>	217
<i>SetFeedback ()</i>	217
<i>SetMimeType ()</i>	218
<i>SetBuffer ()</i>	218
<i>SelectMimeType ()</i>	218
<i>GetSelectedMimeType ()</i>	219
<i>GetBuffer ()</i>	219
<i>DropInternal ()</i>	219
FIELD	221
FGL_BUFFERTOUCED()	221

FGL_DIALOG_INFIELD()	223
FGL_DIALOG_GETBUFFER()	224
FGL_DIALOG_GETCURSOR()	225
FGL_DIALOG_GETFIELDNAME()	226
FGL_DIALOG_SETBUFFER()	227
FGL_DIALOG_SETCURSOR()	228
FGL_DIALOG_SETFIELDORDER ()	229
FGL_DIALOG_SETSELECTION()	230
FGL_DIALOG_GETSELECTIONEND()	231
FGL_DIALOG_UPDATE_DATA()	232
FGL_FORMFIELD_GETOPTION()	233
FGL_OVERWRITE()	238
FIELD_TOUCHED()	239
GET_FLDBUF()	242
INFIELD()	247
COMBOBOX	251
FGL_LIST_CLEAR()	251
FGL_LIST_COUNT()	251
FGL_LIST_FIND()	252
FGL_LIST_GET()	252
FGL_LIST_INSERT()	252
FGL_LIST_SORT()	252
FGL_LIST_RESTORE()	253
FGL_LIST_SET()	253
UI.COMBOBOX CLASS	253
<i>SetDefaultInitializer()</i>	253
<i>ForName()</i>	254
<i>Clear()</i>	254
<i>AddItem()</i>	254
<i>GetTableName()</i>	255
<i>GetColumnName()</i>	255
<i>GetTag()</i>	255
<i>GetIndexOf()</i>	255
<i>GetItemCount()</i>	255
<i>GetItemName()</i>	255
<i>GetItemText()</i>	255
<i>GetTextOf()</i>	256
<i>RemoveItem()</i>	256

DATE/TIME	257
CURRENT	257
DATE	261
DAY()	264
EXTEND()	265
MDY()	269
MONTH()	270
TIME()	271
TODAY	272
UNITS	273
WEEKDAY()	275
YEAR()	278
STRING	280
CLIPPED	280
CONCATENATION () OPERATOR	283
DOWNSHIFT()	285
FGL_CHARBOOL_TO_INTBOOL()	286
FGL_INTBOOL_TO_CHARBOOL()	287
LENGTH()	288
ORF()	290
SFMT()	291
SPACE	292
SUBSTRING ([])	294
TRIM()	296
UPSHIFT()	297
USING	300
WORDWRAP	313
STRING METHODS	316
<i>append()</i>	316
<i>equals()</i>	316
<i>equalsIgnoreCase()</i>	316
<i>getCharAt()</i>	317
<i>getIndexOf()</i>	317
<i>getLength()</i>	317
<i>substring()</i>	318
<i>toLowerCase()</i>	318

<i>toUpperCase()</i>	318
<i>trim()</i>	318
<i>trimLeft()</i>	319
<i>trimRight()</i>	319
BASE.STRINGBUFFER CLASS	320
<i>create()</i>	320
<i>append()</i>	320
<i>clear()</i>	320
<i>toString()</i>	321
<i>equals()</i>	321
<i>equalsIgnoreCase()</i>	322
<i>getCharAt()</i>	322
<i>getIndexOf()</i>	322
<i>getLength()</i>	322
<i>substring()</i>	323
<i>toLowerCase()</i>	323
<i>toUpperCase()</i>	323
<i>trim()</i>	323
<i>trimLeft()</i>	324
<i>trimRight()</i>	324
<i>replaceAt()</i>	324
<i>replace()</i>	324
<i>insertAt()</i>	325
BASE.STRINGTOKENIZER CLASS	326
<i>create()</i>	326
<i>createExt()</i>	326
<i>countTokens()</i>	327
<i>hasMoreTokens()</i>	327
<i>nextToken()</i>	327
THE OS.PATH METHODS	328
OS.PATH.ATIME	328
OS.PATH.BASENAME	328
OS.PATH.CHDIR	329
OS.PATH.CHVOLUME	329
OS.PATH.COPY	329
OS.PATH.DELETE	330
OS.PATH.DIRCLOSE	330
OS.PATH.DIRFMASK	330
OS.PATH.DIRNAME	331
OS.PATH.DIRNEXT	331
OS.PATH.DIROPEN	331

OS.PATH.DIRSORT	332
OS.PATH.EXECUTABLE	333
OS.PATH.EXISTS	333
OS.PATH.EXTENSION.....	333
OS.PATH.HOMEDIR.....	333
OS.PATH.ISDIRECTORY	334
OS.PATH.ISFILE	334
OS.PATH.ISHIDDEN	334
OS.PATH.ISROOT	334
OS.PATH.JOIN.....	335
OS.PATH.MKDIR.....	335
OS.PATH.MTIME.....	335
OS.PATH.PATHSEPARATOR	336
OS.PATH.PATHTYPE.....	336
OS.PATH.PWD	336
OS.PATH.READABLE.....	336
OS.PATH.RENAME.....	337
OS.PATH.ROOTDIR.....	337
OS.PATH.ROOTNAME.....	337
OS.PATH.SEPARATOR.....	337
OS.PATH.SIZE	338
OS.PATH.TYPE.....	338
OS.PATH.VOLUMES	338
OS.PATH.WRITABLE.....	339
OS.PATH.CHOWN	339
OS.PATH.CHRWX	339
OS.PATH.GID.....	340
OS.PATH.ISLINK.....	340
OS.PATH.RWX	340
OS.PATH.UID.....	341
LARGE DATA TYPES METHODS.....	342
GETLENGTH()	342
READFILE()	342
WRITEFILE()	342

APPLICATION LAUNCH	343
RUN.....	343
EXEC_LOCAL()	345
EXEC_PROGRAM()	346
WINEXEC() AND WINEXECWAIT()	347
WINSHELLEXEC() AND WINSHELLEXECWAIT().....	348
TOOLBAR	349
FGL_DIALOG_KEYDIVIDER()	349
FGL_DIALOG_SETKEYLABEL().....	350
FGL_GETKEYLABEL()	351
FGL_KEYDIVIDER()	352
FGL_SETKEYLABEL()	353
FGL_SETKEYSTATIC()	354
RING MENU	355
CREATE_MENU()	355
MENU_ADD_OPTION().....	355
MENU_ADD_SUBMENU()	356
MENU_PUBLISH()	356
EXECUTE_MENU()	357
DIALOG BOX.....	358
FGL_FILE_DIALOG()	358
FGL_MESSAGE_BOX()	359
FGL_WINBUTTON() §.....	361
ENVIRONMENT	363
FGL_ARCH()	363
FGL_GETENV()	364
FGL_GETPID()	367
FGL_GETRESOURCE()	368
FGL_GETVERSION()	369
FGL_PUTENV().....	370
FGL_SETENV().....	371
FGL_USERNAME()	372

HELP	373
ERR_GET()	373
ERR_PRINT()	374
ERR_QUIT()	376
ERRORLOG()	378
FGL_GETHELP()	380
SHOWHELP()	381
STARTLOG()	383
4GL PROGRAM.....	386
ARG_VAL()	386
NUM_ARGS()	388
BASE.APPLICATION CLASS	390
<i>Command line arguments</i>	390
<i>Application Information</i>	391
<i>GetResourceEntry()</i>	391
<i>GetStackTrace()</i>	391
UI.INTERFACE CLASS USED FOR MDI MODE	393
SETNAME()	393
GETNAME()	393
SETTEXT()	393
GETTEXT ()	393
SETIMAGE()	393
GETIMAGE()	394
SETTYPE()	394
GETTYPE()	394
SETSIZE()	394
SETCONTAINER()	395
GETCONTAINER()	395
GETCHILDCOUNT()	395
GETCHILDINSTANCES()	395
LOADTOOLBAR()	395
LOADTOPMENU()	396
LOADSTARTMENU()	397
USING THE MDI MODE WITH UI.INTERFACE CLASS	398

4GL LANGUAGE	400
ASCII.....	400
COLUMN	402
FGL_COPYBLOB().....	405
FGL_RANDOM()	406
LINENO	407
PAGENO.....	409
SIZEOF()	410
 WEB SERVICES FUNCTIONS	 411
FGL_WSDL_CALL()	411
FGL_WS_CALL()	412
FGL_WS_CREATE()	413
FGL_WS_SETPARAMETERS()	414
FGL_WS_GETDATA()	415
FGL_WS_ERROR()	416
FGL_WS_EXECUTE()	417
FGL_WS_CLOSE().....	418
FGL_WS_SET_OPTION()	419
EXECUTE()	420
LOADMETA()	421
SELECTOPERATION().....	422

Table of Figures

Figure 1 - FUNCTION Program Block Syntax	6
Figure 3 - Expression: MAX.....	14
Figure 5 - Arithmetic Expression Returning an INTERVAL value	20
Figure 6 - Arithmetic Expression Returning DATE value.....	20
Figure 7 - Arithmetic Expression Returning DATETIME Value.....	20
Figure 9 - Logical Operators	30
Figure 10 - Relational Operators	32
Figure 11 - NULL Test.....	35
Figure 12 - LIKE Operator	35
Figure 13 - MATCHES Operator	37
Figure 14 - Set Membership Test	39
Figure 15 - Range Test.....	39
Figure 16 - Set Current Line Screenshot Example.....	111
Figure 17 - fgl_getkey().....	112
Figure 18 - fgl_keyval().....	116
Figure 19 - fgl_keyval Example Output.....	117
Figure 20 - fgl_keyval Return Value Example.....	118
Figure 21 - fgl_lastkey Function.....	120
Figure 22 - fgl_lastkey Message Box	120
Figure 23 - fgl_lastkey Return Value Example.....	120
Figure 24 - fgl_lastkey Message Box Output.....	121
Figure 25 - fgl_lastkey Confirmation Box.....	121
Figure 26 - fgl_report_type() Function.....	155
Figure 27 - fgl_drawbox() Function	166
Figure 28 - fgl_drawline().....	169
Figure 29 - fgl_setsize() Function	173
Figure 30 - fgl_settitle() Function	174
Figure 31 - fgl_winmessage()	181
Figure 32 - fgl_winprompt() Function.....	183
Figure 33 - fgl_winquestion()	185
Figure 34 - fgl_winsize() Message Box	187
Figure 35 - fgl_scr_size() Function.....	204
Figure 36 - Field Get Option Example	237
Figure 37 - FIELD_TOUCHED() Operator	239
Figure 38 - GET_FLDBUF()	242
Figure 39 - INFIELD() Operator	247
Figure 40 - Field Identifier Example.....	250
Figure 47 - Substring Operator	294
Figure 48 - UPSHIFT() Function	297
Figure 51 - fgl_message_box() Function	359
Figure 52 - fgl_getenv().....	364
Figure 53 - ERR_GET() Function.....	373
Figure 54 - ERR_PRINT() Function.....	374
Figure 55 - ERR_QUIT() Function	376
Figure 56 - ERRORLOG() Function	378
Figure 57 - fgl_gethelp()	380
Figure 58 - ARG_VAL().....	386

Figure 59 – Demonstration Output Code.....	408
--	-----

In This Guide

This volume of the Querix4GL Reference describes the kinds of built-in functions you can use in Querix4GL applications. It also describes 4GL operators. Information about the syntax and usage of these built-in functions and operators appears in the sections that follow, arranged in a categorised order according to their functionality.

Any references to demonstration applications mean that the code sample used can also be found as a file within the 4GL demonstration application 'functions' packaged with the Querix Lycia compiler application. Please refer to the 'Lycia Getting Started' guide for information on how to import projects from the CVS into LyciaStudio.

Functions in 4GL Programs

In 4GL, a *function* is a named collection of statements that perform a task. (In some programming languages, terms like *method*, *subroutine*, and *procedure* correspond to a *function* in 4GL.) If you need to repeat the same series of operations, you can call the same function several times, rather than specify the same steps for each repetition. This construct supports the structured programming design goal of segmenting source code modules into logical units, each of which has only a single entry point and controlled exit points.

4GL programs can invoke the following types of functions:

- Programmer-defined 4GL functions
- 4GL built-in functions
- SQL built-in functions (invoked by SQL statements)
- C functions
- ESQL/C functions (if you have ESQL/C modules)

The FUNCTION statement defines a function. Variables that are declared within a function are local to it, but functions that are defined in the same module can reference any module-scope variables that are declared in that module. See the descriptions in the "Programmer-Defined 4GL Functions" section of this volume, and "FUNCTION" in the 4GL Statements volume. The other types of functions that you can call from a 4GL program are briefly discussed on the next pages.

Built-In 4GL Functions

The *built-in functions* of 4GL are predefined functions that support features of Querix4GL. Except for the fact that no FUNCTION definition is required, built-in functions behave exactly like the 4GL functions that you define with the FUNCTION statement.

Built-in 4GL functions include the following features:

- They can be invoked with a CALL statement. (If they return a single value, they can be used without a CALL statement, but must instead use a LET statement.) For example: LET *variable* = *some_function()* is similar to
CALL *some_function()*
RETURNING *variable*
- They must have parentheses in them, even if there are no arguments used.
- They cannot be invoked from SQL statements.
- They can be invoked from C modules. Note that there are also Built-In Functions that can be used by C, but cannot be used by 4GL.

If a FUNCTION statement is used to define a function that has the same name as a built-in function, the built-in function will not work in your program.

Each of the following 4GL built-in functions is known not to function with character mode interfaces (§), to produce unexpected results. These functions are also marked with the symbols shown at their heading in the next chapter:

- fgl_file_to_client §
- fgl_getproperty §
- fgl_setproperty §
- fgl_setsize §
- fgl_settitle §
- fgl_winbutton §
- winexec §
- winshellexec §


Methods used with object data types

Methods used with the object data types such as CURSOR, FORM, PREPARED, WEBSERVICE, and WINDOW are similar to other functions, but have some peculiarities. They can be invoked in the same way as the other functions (i.e. by a CALL statement). The methods are case insensitive like any other 4GL elements.

The methods must be preceded by the variable of an object data type separated from a method by a full stop, e.g.:

```
CALL variable.Method()
```

Sometimes the names of the methods for different data types coincide, e.g. Open() method is used for CURSOR, FORM, AND WINDOW variables. However, the arguments and the returned values strongly depend on the data type of the variable a method is used with. Thus even though some methods have the same names, they are described separately for each data type.

	<p>You should not use these variables with the built-in functions and 4GL statements used to manipulate cursors, forms, windows or prepared statements; they can only be used with the appropriate methods.</p>
---	---

Built-In and External SQL Functions and Procedures

Some RDBMSs support built-in SQL functions, some of which have the same names as built-in 4GL functions or operators. Some RDBMS servers also support the syntax of the following statements:

CREATE FUNCTION	CREATE ROUTINE FROM
CREATE FUNCTION FROM	EXECUTE FUNCTION
CREATE PROCEDURE FROM	CREATE PROCEDURE

These SQL statements register and execute external functions and stored procedures; these are similar to 4GL functions but run on the database server.

Calls to SQL, SPL, and external functions can appear in SQL statements but not in other 4GL statements.

C Functions

To invoke C functions within a 4GL program, you can use a CALL statement or an expression. Some C functions can be helpful in completing tasks that are not easily written in 4GL, such as processing binary I/O.

Unlike 4GL identifiers, names of C functions and their arguments are case-sensitive. When they are used in a function call, they should be typed in lower case.

A C function can be called from the 4GL program using the CALL statement. To call a C function, the CALL statement must include the following information:

- The name of the C function, which is case sensitive.
- Any arguments you want to pass to this function enclosed in parentheses - they can be either literals or variables.
- The RETURNING clause with the variables for the returned values, if the function returns any.

The following example calls the C function MyCfunc(), passes two arguments to this function and returns two values:

```
CALL MyCfunc(var1, 100) RETURNING ret_v1, ret_v2
```

The C function receives an integer argument that specifies how many values were pushed on the argument stack (in this case, two arguments). This is the number of values to be popped off the stack in the C function. The function must test its integer argument to check whether the number of arguments passed to it is correct.

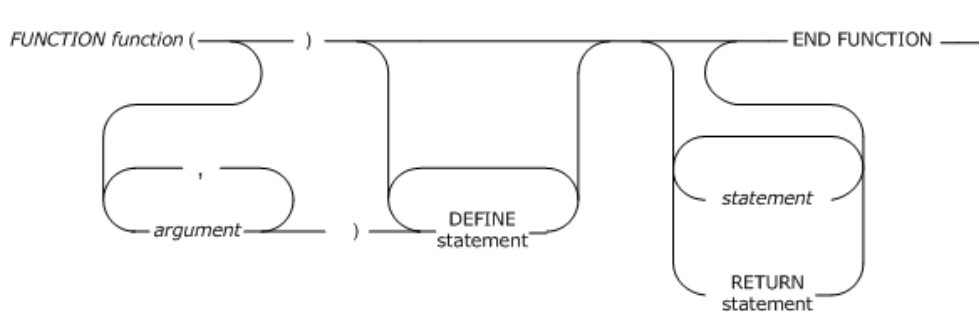
The switch statement is useful in popping the correct number of arguments from the stack. By arranging the valid cases in descending order, the correct number of arguments can be popped in the correct sequence with minimal coding.

ESQL/C Functions

The Lycia compiler can handle ESQL/C, allowing you to use this type of function in your 4GL program, as well as ESQL/C library functions.

Programmer-Defined 4GL Functions

The FUNCTION program block begins with the FUNCTION keyword and ends with the END FUNCTION keywords. These keywords enclose a program block of the 4GL statements that compose the function, and that are executed when the function is invoked.



Syntax of the FUNCTION program block

Figure 1 - FUNCTION Program Block Syntax

Element	Description
<i>argument</i>	This is the name of a formal argument to the function.
<i>function</i>	This is the identifier declared here for the function.
<i>statement</i>	This is a SQL statement or other 4GL statement.

The left-hand side of this diagram, comprising the function identifier and its list of arguments, is sometimes called the function prototype. This is described in detail in "The Prototype of the Function"

section of the 4GL Functions volume. In this specific example this portion resembles the prototype of a report.

The names of functions cannot be the same as any other reports or global variables within the program, or any of the formal arguments of the same function. More details of how to define 4GL functions are given in the "FUNCTION" section of the 4GL Functions volume.

The right-hand side of the diagram, which comprises the declarations of formal arguments and any local variables, and the *statement* block, is sometimes called the FUNCTION *program block*. This part can include any executable statement of SQL or 4GL except the report execution statements (which is described in "4GL Report Execution Statements" in the 4GL Statements volume).

The whole FUNCTION program block must be defined within a single source module. No other FUNCTION, REPORT, or MAIN program blocks can be included in a FUNCTION definition. However, that definition can include statements that create a report, call a function, or execute a RUN statement that invokes another program.

Invoking Functions

Within 4GL programs you can use a CALL statement to invoke any functions, except for SQL functions and stored procedures. These are called in SQL expressions.

In some contexts, functions can be called implicitly. This could occur in the following situations:

- If a function only returns a single value, it can be invoked by specifying its name and any arguments within an expression. This can only be done where a value of the returned data type is valid.
- 4GL has exception-handling features which can automatically invoke a function specified in a WHENEVER...CALL statement.

Passing Arguments and Returning Values

A program block that contains a CALL statement, or an expression that invokes a function, is called the *calling routine*. Functions can receive information from, and return values to the calling routine. Often, where this is a separate program block, values from the calling routine and from other program blocks are visible to the function only through global or module variables, or through the argument list of the calling statement.

For most data types, the RETURN statement in the function and the RETURNING clause of the CALL statement that specify any values returned to the calling routine. The operation of communicating between the function and its calling routine is called *passing by value*.


Being larger in size, BYTE or TEXT datatype arguments are processed differently, and this is called *passing by reference*. Although the BYTE or TEXT variables appear as arguments of the calling statement, what is actually passed to the function is a pointer to the variables.

The RETURN statement and RETURNING clause cannot include BYTE or TEXT variables. The 4GL built-in functions described in this volume all pass their arguments by value, rather than by reference.

Invoking SQL Functions

Predefined SQL functions and operators can only be invoked in 4GL programs within SQL statements. For example, the USER operator of SQL can appear in a SELECT statement:

```
DEFINE x DATETIME YEAR TO SECOND
SELECT CURRENT INTO x FROM DUAL
```

	<p>Some built-in 4GL functions and operators have the same names as SQL functions or operators. For example, CURRENT, DATE(), DAY(), EXTEND(), LENGTH(), MDY(), MONTH(), WEEKDAY(), YEAR(), and the relational operators are features of both 4GL and SQL. (For more information, see "Relational Operators" in this volume.)</p> <p>If a statement that is not an SQL statement references an SQL function or operator that is not also a 4GL function or operator, you may well see a compile time or link-time error.</p>
---	--

Built-in SQL functions and operators like USER cannot, however, appear in other 4GL statements that are not SQL statements. If a program requires the functionality of USER in a non-SQL statement like PROMPT, for example, you must first use FUNCTION to define an equivalent 4GL function. Querix 4GL provides the user function 'fgl_username()' to perform this task:

```
MAIN
  DISPLAY "This process is currently run under the username: ", fgl_username() at
  5,5
  CALL fgl_winmessage("Exit","Press any key to close this demo application","info")
END MAIN
```

Operators of 4GL

Informix 4GL operators differ from 4GL functions in several ways:

- With the exception of GET_FLDBUF(), the CALL statement cannot invoke operators
- Some operators are able to take non-alphanumeric symbols as operands
- Operators cannot be referenced from C programs

Although they are different, we have chosen to describe 4GL operators in this volume. This is because they are similar to 4GL functions in terms of syntax and behaviour. Operators that return a single value can be operands in expressions.

The FUNCTION statement can define (and CALL can invoke) a 4GL function that has the same name as a 4GL operator that can be defined by a FUNCTION statement, or invoked by a CALL function. In such cases, only the operator, not the function, is visible as an operand within a 4GL expression. For example:

```
MAIN
  DEFINE my_date DATE

  DISPLAY "*** MDY() Example ***" AT 1,5
```

```

DISPLAY "LET my_date = MDY(06/3,3+9,2005)" AT 3,5
LET my_date = MDY(06/3,3+9,2005)
DISPLAY my_date at 5,5

CALL fgl_winmessage("Exit","Press any key to close this demo application","info")
END MAIN

```

The keyword-based operators of 4GL listed below are all described in this volume. These operators, as well as other arithmetic and relational operators, are included here so that syntax articles can be found without the need to classify a given feature as a function or an operator.

AND	LINENO
ASCII <i>int-expr</i>	MATCHES <i>expr</i>
BETWEEN <i>expr</i> AND <i>expr</i>	<i>int-expr</i> MOD <i>int-expr</i>
<i>char-expr</i> CLIPPED	NOT
COLUMN <i>int-expr</i>	OR
CURRENT(<i>qualifier</i>)	PAGENO
DATE (<i>date-expression</i>)	<i>int-expr</i> SPACE
DAY (<i>date-expression</i>)	<i>int-expr</i> SPACES
EXTEND (<i>value, qualifier</i>)	TIME
FIELD_TOUCHED(<i>field-list</i>)	TODAY
GET_FLDBUF(<i>field-list</i>)	<i>int-expr</i> UNITS <i>time-keyword</i>
INFIELD(<i>field</i>)	<i>expression</i> USING <i>format-string</i>
IS NULL	WEEKDAY (<i>date-expression</i>)
LENGTH(<i>char-expression</i>)	<i>char-expr</i> WORDWRAP
LIKE	YEAR (<i>date-expression</i>)

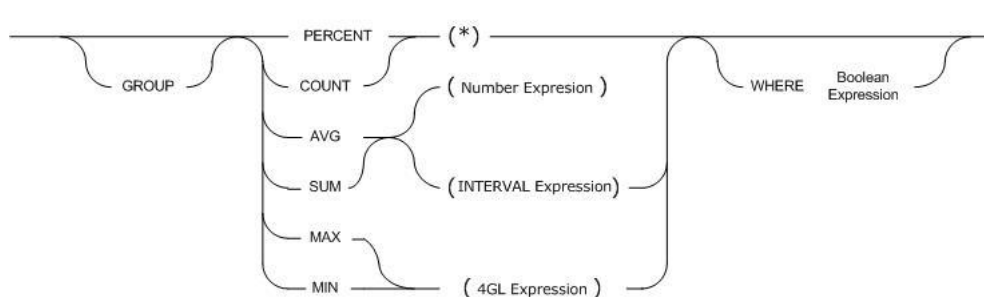
The following table lists the arithmetic and relational operators of 4GL.

Arithmetic Operators		Relational Operators	
Symbol	Description	Symbol	Description
+	Addition	<	Less than
/	Division	<=	Less than or equal to
**	Exponentiation	= or ==	Equal to
MOD	Modulus	!= or <>	Not equal to
*	Multiplication	>=	Greater than or equal to
-	Subtraction	>	Greater than
-	Unary negative		
+	Unary positive		

For a general discussion of 4GL operators, see "Operators in 4GL Expressions" in the Datatypes volume.

Aggregate Report Functions

Each *aggregate report function* of 4GL returns a value summarizing data from all the input records or from a specified group of input records. The 4GL report aggregates are not valid outside of a REPORT program block.



Aggregate Report Functions syntax

Figure 2 - Aggregate Report Function Syntax

Usage

Aggregate report functions return values based on the input records of a 4GL report. Generally these are statistical aggregates, except for MAX and MIN, where their extreme values are returned.

The GROUP keyword indicates that the aggregate function is to be performed on a sub-group of the report data, rather than all rows of the report. It is only valid in the AFTER GROUP section. The value of the function then corresponds to the aggregate function applied only to the group of data that has just been processed.

A WHERE clause can also be applied to an aggregate function. The values processed by the function are only those for which the WHERE clause evaluates to true. The percent operator is a special case; it indicates the percentage of the rows seen which matched the WHERE clause, if referenced. (If no WHERE clause is specified, this will be 100%.)

The 4GL report aggregates are similar to the SQL aggregates that can appear in SELECT statements, but their syntax is not the same.

The differences are described in detail in the "Differences Between the 4GL and SQL Aggregates" section of this chapter.

Aggregate report functions can appear as arguments in 4GL expressions, but they cannot be nested. That is, no expression within a report aggregate can include a report aggregate.

Using the name of an aggregate as an identifier will result in an error. Programmer-defined functions used to calculate the same statistics can be invoked either inside or outside REPORT definitions, but the names that you use for those functions must be different from the names of the identifiers.

You cannot use large or structured data type variables as arguments for these functions. It is possible, however, to specify the name of a simple variable that is a member of a record, or that is an element of an array.

AVG(), SUM(), MIN(), and MAX() ignore records with null values for their argument, but each returns NULL if all records have a null value. If one of the values used in an aggregate function is NULL then the result of the function will also be NULL. For example: SUM(x) + SUM(y) yields a different result to SUM(x+y), since null+value = a value, while null multiplied by a value = null.

If an aggregate value that depends on all records of the report appears anywhere except in the ON LAST ROW control block, each variable in that aggregate or WHERE clause must also appear in the list of formal arguments of the report. (Examples of aggregates that depend on all records include using GROUP COUNT(*) anywhere in a report, or using any aggregate without the GROUP keyword anywhere outside the ON LAST ROW control block.)

In order to evaluate aggregates, 4GL must store all report data in a temporary table before processing it. An error results if no database is open (because the temporary table cannot be created), or if you change or close the current database while evaluating an aggregate. The only case in which 4GL will not use a temporary table is if ORDER BY EXTERNAL is used. This will take all the data from the external source in the order it is presented.

The GROUP Keyword

This optional keyword causes the aggregate function to include data only for a group of records that have the same value on a variable that you specify in an AFTER GROUP OF control block. An aggregate can include the GROUP keyword only within an AFTER GROUP OF control block. If you need the value of a GROUP report aggregate elsewhere, you must use the LET statement within the AFTER GROUP OF control block to store the value in a variable that has the appropriate scope of reference.

The WHERE Clause

The optional WHERE keyword selects among the records passed to the report, including only those for which a Boolean expression is TRUE. Conditional aggregates are calculated on the first pass, when the records are read, and printed on the second pass. You cannot use aggregates in a loop, such as FOR or WHILE, where the Boolean expression changes dynamically.

The MAX() and MIN() Functions

These return the maximum value and minimum value (respectively) of their argument among all records, or among records qualified by the WHERE clause or by the GROUP keyword. For character data, greater than means after in the ASCII collation sequence, where *a* > *A* > 1, and *less than* means *before* in the

ASCII sequence, where $1 < A < a$. For DATE or DATETIME data, *greater than* means *later* and *less than* means *earlier* in time.

Where non-default locales are used, 4GL sorts character operands of MIN() or MAX() in code-set order, unless the locale files define a localized collation sequence in its COLLATION category, and **DBNLS** is set to 1. This order also applies to character variables whose values were retrieved from the database.

Unlike the database, 4GL makes no distinction between character strings whose values were retrieved from CHAR or VARCHAR columns and values from NCHAR or NVARCHAR columns of the database. To sort strings by the rules of the database, rather than by these 4GL rules, write your code so that the database performs the sorting.

max()

This built-in report function returns the maximum value for *expression* taken from all the records in the current database. The range of data from which the return value is drawn can be limited by the GROUP and WHERE specifiers.

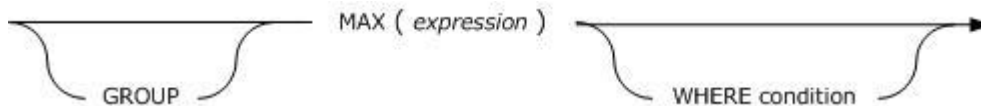


Figure 3 - Expression: MAX

Usage

For DATE and DATETIME data, limits can be placed using the greater than (<) and less than (>) symbols. 'Greater than' meaning after the date/time specified, 'less than' limiting the search to values earlier than that specified.

Character strings are sorted according to their first character.

min()

The MIN report function provides the lowest value for a report variable at the current stage within a 4GL report. With the exception of the operator itself, the syntax is identical to that of the MAX report function.

Usage

The following example uses the MIN operator to output the lowest value for the report input variable.

```
MAIN
  START REPORT agg TO SCREEN
  OUTPUT TO REPORT agg(10)
  OUTPUT TO REPORT agg(20)
  OUTPUT TO REPORT agg(30)
  OUTPUT TO REPORT agg(40)
  OUTPUT TO REPORT agg(50)
```

```
OUTPUT TO REPORT agg(60)
OUTPUT TO REPORT agg(70)
OUTPUT TO REPORT agg(80)
OUTPUT TO REPORT agg(90)
FINISH REPORT agg
END MAIN

REPORT agg(in_val)
  DEFINE in_val INTEGER

  FORMAT
    ON EVERY ROW
      PRINT "Input value: ", in_val
    ON LAST ROW
      PRINT "Smallest: ", MIN(in_val)
  END REPORT
```

The AVG() and SUM() Functions

These functions calculate the average and the sum (or total), respectively, of the expression for all records. The range of records on which the calculations are performed can be modified by using the WHERE clause or the GROUP keyword. The operand of AVG() or SUM() must be a 4GL expression of a number or INTERVAL data type.

sum()

The SUM() report function is used to total the values given to the report up to the point that the function is invoked.

Code Sample: sum_report_function.4gl

```
MAIN
  START REPORT agg TO SCREEN
  OUTPUT TO REPORT agg(10)
  OUTPUT TO REPORT agg(20)
  OUTPUT TO REPORT agg(30)
  OUTPUT TO REPORT agg(40)
  OUTPUT TO REPORT agg(50)
  OUTPUT TO REPORT agg(60)
  OUTPUT TO REPORT agg(70)
  OUTPUT TO REPORT agg(80)
  OUTPUT TO REPORT agg(90)
  FINISH REPORT agg
END MAIN
```

```
REPORT agg(in_val)
  DEFINE in_val INTEGER

  FORMAT
    ON EVERY ROW
      PRINT "Input value: ", in_val
    ON LAST ROW
      PRINT "Total: ", SUM(in_val)
END REPORT
```

avg()

The AVG() Report function is used to find the average of the given values up to the point that the function is invoked.

Code Sample: avg_report_function.4gl

```
MAIN
  START REPORT agg TO SCREEN
  OUTPUT TO REPORT agg(10)
  OUTPUT TO REPORT agg(20)
  OUTPUT TO REPORT agg(30)
  OUTPUT TO REPORT agg(40)
  OUTPUT TO REPORT agg(50)
  OUTPUT TO REPORT agg(60)
  OUTPUT TO REPORT agg(70)
  OUTPUT TO REPORT agg(80)
  OUTPUT TO REPORT agg(90)
  FINISH REPORT agg
END MAIN

REPORT agg(in_val)
  DEFINE in_val INTEGER

  FORMAT
    ON EVERY ROW
      PRINT "Input value: ", in_val
    ON LAST ROW
      PRINT "Average: ", AVG(in_val)
END REPORT
```

The COUNT (*) and PERCENT (*) Functions

These functions are most often used when the total number of records is qualified by a WHERE clause. Their results are the number of records that fall within the parameters of the WHERE clause, and the

percentage that the COUNT result is of the total number of records in the report. If no WHERE function is included, the COUNT figure will be the total number of records in a report.

You must include the (*) symbols. Like the other report aggregates, PERCENT(*) and COUNT(*) cannot be used within an expression.

The following fragment of a REPORT routine uses the ON LAST ROW control block to output the report count. These aggregate functions may be used in reference to any logical group within a 4GL report.

```
MAIN
START REPORT agg TO SCREEN
OUTPUT TO REPORT agg(10)
OUTPUT TO REPORT agg(20)
OUTPUT TO REPORT agg(30)
OUTPUT TO REPORT agg(40)
OUTPUT TO REPORT agg(50)
OUTPUT TO REPORT agg(60)
  OUTPUT TO REPORT agg(70)
OUTPUT TO REPORT agg(80)
OUTPUT TO REPORT agg(90)
FINISH REPORT agg
END MAIN

REPORT agg(in_val)
DEFINE in_val INTEGER

FORMAT
ON EVERY ROW
PRINT "Input value: ", in_val
ON LAST ROW
PRINT "Count: ", COUNT(*)
END REPORT
```

PERCENT Report Function

Code Sample: percent_report_function.4gl

```
MAIN
START REPORT agg TO SCREEN
OUTPUT TO REPORT agg(10)
OUTPUT TO REPORT agg(20)
OUTPUT TO REPORT agg(30)
OUTPUT TO REPORT agg(40)
OUTPUT TO REPORT agg(50)
OUTPUT TO REPORT agg(60)
OUTPUT TO REPORT agg(70)
```

```
OUTPUT TO REPORT agg(80)
OUTPUT TO REPORT agg(90)
FINISH REPORT agg
END MAIN

REPORT agg(in_val)
  DEFINE in_val INTEGER

  FORMAT
    ON EVERY ROW
      PRINT "Input value: ", in_val
    ON LAST ROW
      PRINT "Percent: ", PERCENT(*)
END REPORT
```

Differences between the 4GL and SQL Aggregates

The syntax of the SQL aggregate functions can be found in the *Informix Guide to SQL: Syntax*. The main differences between the 4GL report aggregates and aggregate functions are:

- Only 4GL report aggregates can use the PERCENT(*) and GROUP keywords.
- Only SQL aggregates can use ALL, DISTINCT, and UNIQUE as keywords.
- In 4GL reports, COUNT can only take an asterisk (*) as its argument, but in SELECT statements of SQL, the COUNT aggregate can also use a column name or an SQL expression as its argument.

Only SQL aggregate functions can use database column names as arguments, but this syntax difference is not of much practical importance. (Operands in the expressions that you specify as arguments for 4GL report aggregates can be program variables that contain values from database columns.)

References

LINENO, PAGENO, WORDWRAP

Arithmetic Operators

The 4GL arithmetic operators perform arithmetic operations on operands of number data types (and in some cases, on operands of time data types). Arithmetic expressions that return a number value have this syntax:

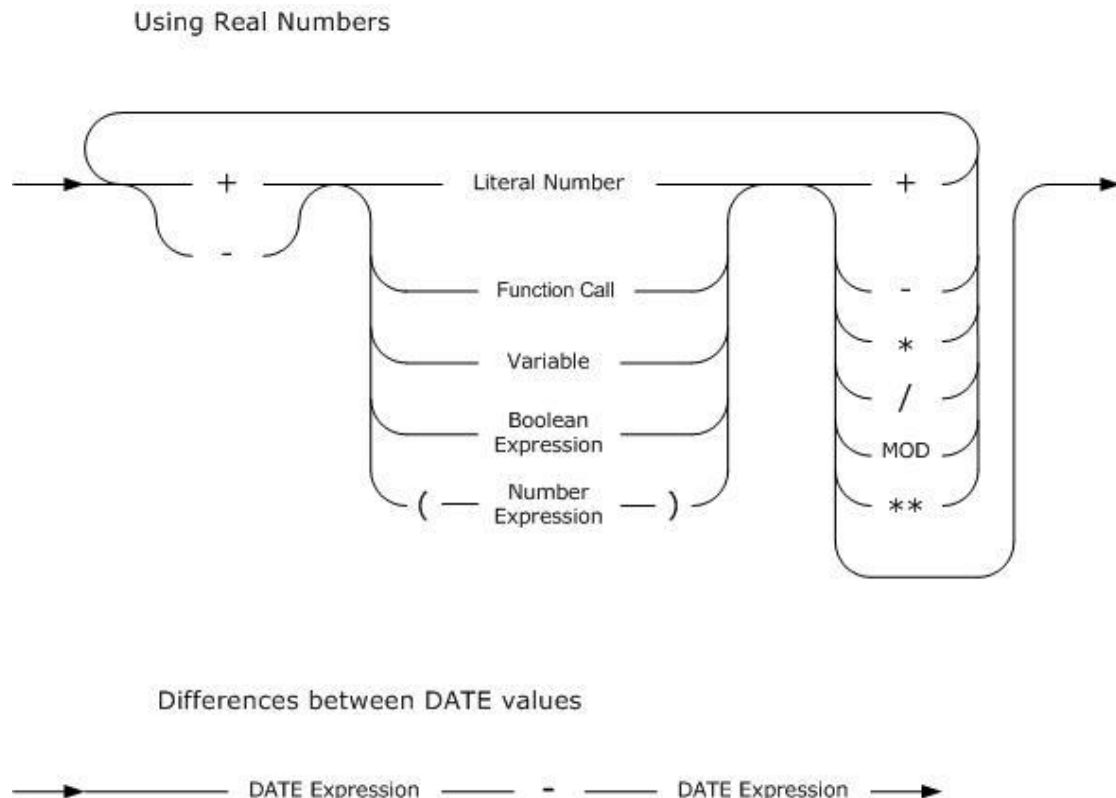


Figure 4 - Arithmetic Expressions Returning a Value

In the next three syntax diagrams, the DATETIME and INTERVAL expression segments are only a subset of time expressions, as described in the "Time Expressions" section of the Datatypes volume. A DATETIME expression or an INTERVAL expression used as an arithmetic operand can be any of the following items:

- A program variable of the DATETIME or INTERVAL data type
- A DATETIME or INTERVAL value that a function or operator returns
- A DATETIME literal or an INTERVAL literal (both described in the Datatypes volume)

A DATETIME or INTERVAL expression cannot be a quoted string or a numeric DATETIME value, or a numeric INTERVAL value that omits the DATETIME or INTERVAL keyword and the DATETIME qualifier or INTERVAL qualifier.

This is the syntax for arithmetic expressions that return an INTERVAL value.

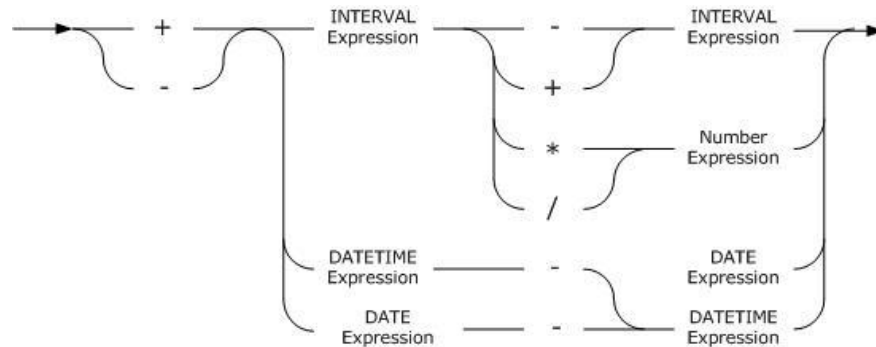


Figure 5 - Arithmetic Expression Returning an INTERVAL value

This is the syntax for arithmetic expressions that return a DATE value.

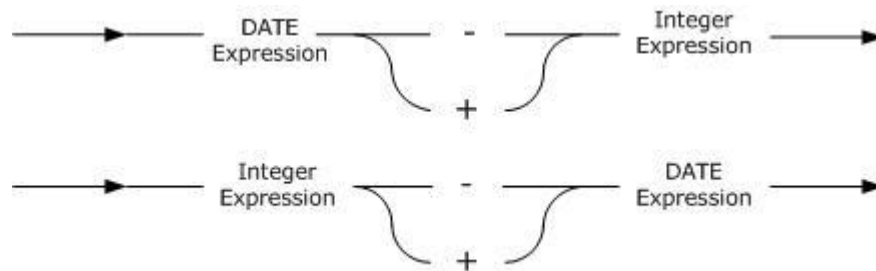


Figure 6 - Arithmetic Expression Returning DATE value

This is the syntax for arithmetic expressions that return a DATETIME value.

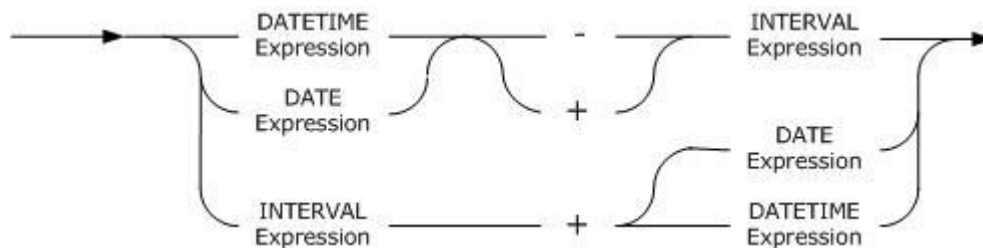


Figure 7 - Arithmetic Expression Returning DATETIME Value

Usage

The arithmetic operators of 4GL resemble the arithmetic functions of Informix database servers, but they are evaluated by the client system.

Arithmetic operands in Q4GL can be of simple or structured (ARRAY or RECORD) data types. Large (BYTE or TEXT) data types are not valid as operands.

The range of returned values is that of the returned data type.

In most situations, arithmetic operands using number values can be represented as CHAR or VARCHAR strings. However this imposes other complications and it is better to use number datatypes where possible. This usually provides greater efficiency in 4GL applications that include a large number of arithmetic operators.

If a Boolean expression is an arithmetic operand, 4GL evaluates it and converts it to an integer on the basis that TRUE = 1 and FALSE = 0.

If any component of an expression that includes an arithmetic operator is a null value, the entire expression returns NULL, unless the null component is an operand of the IS NULL or IS NOT NULL operators.

Unary Arithmetic Operators

At the left of expressions that return a number or INTERVAL value, plus (+) and minus (-) symbols can appear as *unary operators* to specify the sign. For unsigned values, the default is positive (+). The number data types of 4GL include DECIMAL, FLOAT, INTEGER, MONEY, SMALLFLOAT, and SMALLINT.

Unary plus (+) and minus (-) operators are recursive. Parentheses must separate the subtraction operator from any immediately following unary minus sign, as in "minuend -(-subtrahend)" unless you want 4GL to interpret the -- symbols as indicating the presence of a comment.

Binary Arithmetic Operators

There are six binary arithmetic operators which appear in number expressions. As the syntax diagrams at the beginning of this section indicate, four of these operators (*, /, +, and -) also can appear in time (DATE, DATETIME, and INTERVAL) expressions. The MOD and exponentiation (**) operators accept some DATE values as operands, but such expressions might return values that are difficult to interpret.

Symbol	Operator Name	Name of Result	Precedence
**	Exponentiation	Power	13
MOD	Modulus	Remainder	13
*	Multiplication	Product	12
/	Division	Quotient	12
+	Addition	Sum	11
-	Subtraction	Difference	11

4GL performs calculations with binary arithmetic operators of number data types after automatically converting both operands to DECIMAL values.

Time operands in arithmetic expressions cannot be quoted strings representing numeric date-and-time values or numeric time intervals. Instead DATETIME or INTERVAL literals should be used that include an appropriate qualifier.

For example, the following LET statement that attempts to use an arithmetic time expression as its right-hand term in fact assigns a null value to the INTERVAL variable **time_difference**, rather than an interval of 10 years.

```
LET totalsec = "2004-01-01 00:00:00.000" - "1994-01-01 00:00:00.000"
```

To achieve the desired result, which is not null, would need:

```
LET totalsec = DATETIME (2004-01-01 00:00:00.000) YEAR TO FRACTION  
DATETIME (1994-01-01 00:00:00.000) YEAR TO FRACTION
```

Sample Code: Demonstration of Binary Arithmetic Operators

```
MAIN
  DEFINE inp_char CHAR(1)

  # A date/datetime - date/datetime returns an interval, not a date/datetime.
  #DEFINE time_difference DATE
  DEFINE time_difference INTERVAL YEAR TO YEAR

  #Incorrect - value on the right hand side will assign NULL value
  LET time_difference = "2005-01-01 00:00:00.000" - "1995-01-01 00:00:00.000"

  DISPLAY "Incorrect method of time difference calculation 2005 - 1995" at 3,5
  DISPLAY "LET time_difference =" AT 4,5
  DISPLAY "\"2005-01-01 00:00:00.000\" - \"1995-01-01 00:00:00.000\" (NULL)" at
5,5
  DISPLAY "Time Difference = ", time_difference at 6,5

  #correct method
  LET time_difference = DATETIME (2005-01-01 01:01:01.001) YEAR TO FRACTION -
DATETIME (1995-01-01 01:01:01.001) YEAR TO FRACTION


  DISPLAY "Correct method of time difference calculation 2005 - 1995" at 9,5
  DISPLAY "LET time_difference =" AT 10,5
  DISPLAY "DATETIME (2005-01-01 01:01:01.001) YEAR TO FRACTION" at 11,5
  DISPLAY " - DATETIME (1995-01-01 01:01:01.001) YEAR TO FRACTION" at 12,5
  DISPLAY "Time Difference = ", time_difference at 13,5
  PROMPT "Press any key to close this demo application" FOR inp_char
END MAIN
```

If the first operand of an arithmetic expression includes the UNITS operator, you must enclose that operand in parentheses.

The following table shows the precedence (**P**) and data types of operands and of returned values for both unary and binary arithmetic operators. Time operands not listed here produce either errors or meaningless results.

P	Expression	Left (=x)	Right (=y)	Returned Value
13	+ y		Number or INTERVAL	Same as y
	- y		Number or INTERVAL	Same as y
12	x * * y	Number	INT or SMALLINT	Same as y
	x MOD y	INT or SMALLINT	INT or SMALLINT	Same as y
11	x * y	Number or INTERVAL	Number	Same as y
	x / y	Number or INTERVAL	Number	Same as x
10	x + y	Number	Number	Number
	x + y	INT or SMALLINT	DATE	DATE
	x + y	DATE	INT or SMALLINT	DATE
	x + y	DATE or DATETIME	INTERVAL	DATETIME
	x + y	INTERVAL	DATE or DATETIME	DATETIME
	x + y	INTERVAL	INTERVAL	INTERVAL
	x - y	Number	Number	Number
	x - y	INT or SMALLINT	DATE	DATE
	x - y	DATE	INT or SMALLINT	DATE
	x - y	DATE or DATETIME	INTERVAL	DATETIME
	x - y	DATE or DATETIME	DATETIME	INTERVAL
	x - y	DATETIME	DATE	INTERVAL
	x - y	INTERVAL	INTERVAL	INTERVAL
	x - y	DATE	DATE	INT

The precedence of all 4GL operators is listed in "Operators in 4GL Expressions" section within the Datatypes volume; the operators that are not listed there have a precedence of 1. These precedence (**P**) values are ordinal numbers to show relative ranks.

	<i>DATE</i> and <i>DATETIME</i> values have no true zero point. Such values can support addition, subtraction, and the relational operators, but division, multiplication, and exponentiation are logically undefined for these data types.
---	---

Exponentiation (**) Operator

The exponentiation (**) operator returns a value calculated by raising the left-hand operand to a power corresponding to the integer part of the right-hand operand. This right-hand operand cannot have a negative value.

An expression specifying the right-hand MOD operand cannot include the exponentiation operator. Before conversion to DECIMAL for evaluation, 4GL converts the right-hand operand of the exponentiation operator to an integer.

```
MAIN
  DEFINE int_r INTEGER
  DEFINE dec_r DECIMAL(8,2)

  LET int_r = 12
  LET dec_r = 123.12

  DISPLAY "Integer ** Integer: ", int_r, " ** -1 = ", int_r ** -1 USING "<<<<<&"
  DISPLAY "Integer ** Integer: ", int_r, " ** 0 = ", int_r ** 0 USING "<<<<<&"
  DISPLAY "Integer ** Integer: ", int_r, " ** 5 = ", int_r ** 5 USING "<<<<<&"
  DISPLAY "Integer ** Decimal: ", int_r, " ** 5.12 = ", int_r ** 5.12 USING
"<<<<<&"
  DISPLAY "Decimal ** Integer: ", dec_r, " ** -1 = ", dec_r ** -1 USING "<<<<<&.&&"
  DISPLAY "Decimal ** Integer: ", dec_r, " ** 0 = ", dec_r ** 0 USING "<<<<<&.&&"
  DISPLAY "Decimal ** Integer: ", dec_r, " ** 5 = ", dec_r ** 5
  DISPLAY "Decimal ** Decimal: ", dec_r, " ** 5.12 = ", dec_r ** 5.12

  SLEEP 4
END MAIN
```

mod

The modulus (MOD) operator returns the remainder from integer division when the integer part of the left-hand operator is divided by the integer part of the right-hand operator.

Usage

The demonstration code sample below shows the values of y and z, both of which are rounded down for the calculation. The resultant y value is then divided by the resultant value of z, to give a remainder value of two (2), which is the integer returned by the operator. The syntax is:

```
MAIN
  DEFINE
    y,z FLOAT,
    x INTEGER,
    str_disp CHAR(75)

  LET y = 11.83
  LET z = 3.97

  LET x = y MOD z
```

```
LET str_disp = "x = y MOD z = " || x

DISPLAY "The remainder of the integer division 11 / 3 is: ", x at 4,5
DISPLAY str_disp at 5,5

CALL fgl_winmessage("Exit","Press any key to close this demo application","info")
END MAIN
```

In 4GL programs, MOD is a reserved word. Do not use it as a 4GL identifier.

An expression specifying the right-hand MOD operand can neither include the exponentiation or modulus operator, nor can it be zero. Before conversion to DECIMAL for evaluation, 4GL converts any operand of MOD that is not an INTEGER or SMALLINT data type to an INTEGER value by truncation. Any fractional part is discarded.

Multiplication (*) and Division (/) Operators

The multiplication (*) operator returns the scalar product of the operands on either side of it. The division (/) operator returns the result of the calculation in which the left-hand operand is divided by the right-hand operand. An error is returned if the value on the right-hand side is zero.

If both operands of the division operator are of the INT or SMALLINT datatype, 4GL discards any fractional portion of the result.

For multiplication and division, if the left-hand operand has an INTERVAL value, the result is an INTERVAL value of the same precision. The right-hand operand must be an expression that returns a number data type.

When the results of division are assigned to a fixed-point DECIMAL variable, results are rounded, if the fractional part of the result contains more decimal places than the declared scale of the receiving DECIMAL data type.

Addition (+) and Subtraction (-) Operators

The addition (+) and subtraction (-) operators return the algebraic sum and difference, respectively, between their left- and right-hand operands.

You can use DATE operands in addition and subtraction, but not the sum of two DATE values. All the other binary arithmetic operators also accept DATE operands, equivalent to the count of days since December 31, 1899; but the values returned (except from a DATE expression as the left-hand MOD operand) are meaningless in most applications.

Do not write expressions that specify the sum of two DATE or DATETIME values, or a difference whose second operand is a DATE or DATETIME value, and whose first operand is an INTERVAL value.

The difference between two DATETIME values (or a DATETIME and a DATE value, but not two DATE values) is an INTERVAL value. If the operands have different qualifiers, the result has the qualifier of the first operand.

The difference between two DATE values is an INTEGER value, representing the positive or negative number of *days* between the two calendar dates. You must explicitly apply the UNITS DAY operator to a difference between DATE values to store the result as an INTERVAL DAY TO DAY value.

An arithmetic expression cannot combine an INTERVAL value of precision in the range YEAR to MONTH with another in the DAY to FRACTION range. Neither can an expression combine an INTERVAL value with a DATETIME or DATE value that has different qualifiers. You must explicitly use the EXTEND operator to change the DATE or DATETIME precision to match that of the INTERVAL operand.

Arithmetic with DATE or DATETIME values sometimes produces errors. For example, adding or subtracting a UNITS MONTH operand to a date near the end of a month can return an invalid date (such as January 31).

SIZE Operator

The SIZE operator is specific to Q4GL. This returns the size of an array, or the number of elements in a record.

Boolean Operators

A 4GL Boolean operator returns TRUE (= 1), FALSE (= 0) or NULL. These 4GL operators resemble the SQL Boolean operators, but some are not identical.

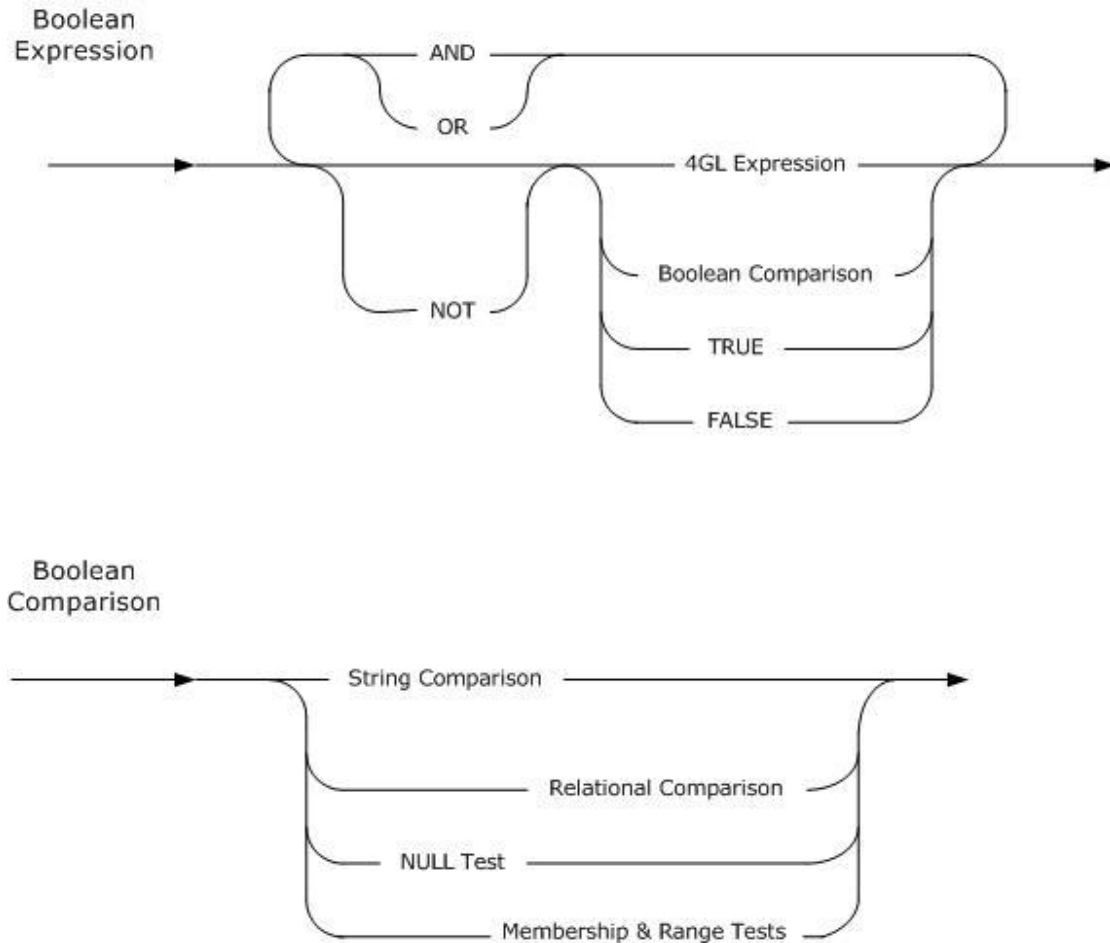


Figure 8 - Boolean Operators

The Boolean operators include the logical operators AND, OR, and NOT, and operators for Boolean comparisons, as described in the sections that follow.

Logical Operators

The logical operators AND, OR, and NOT combine Boolean values into a single 4GL Boolean expression. AND, OR, and NOT produce the following results (where T means TRUE, F means FALSE, and ? means NULL).

AND	T	F	?		OR	T	F	?		NOT	
T	T	F	?		T	T	T	T		T	F
F	F	F	F		F	T	F	?		F	T
?	?	F	?		?	T	?	?		?	?

Figure 9 - Logical Operators

The top row of each table lists the name of the operators and, for AND and OR, the possible values of their left-hand operands. The values of the right-hand operands are listed below the operator name. The values returned by the operators appear in the cells where the row and column intersect. Any non-zero operand that is not null is assumed by the logical operators to be TRUE.

When one or both arguments are null, the result can also be null in some cases. For example, if **var1** = 0 and **var2** = NULL, the following expression assigns a null value to variable **x**:

```

MAIN
  DEFINE
    bol1, bol2, x smallint

  LET bol1 = TRUE
  ## LET bol2 = NULL
  LET x = bol1 OR bol2

  IF x THEN
    CALL fgl_winmessage("bol1 OR bol2 = TRUE", "bol1=TRUE    bol2=NULL \nThis
boolean expression -      LET x = bol1 OR bol2 \n is evaluated as TRUE", "info")
  ELSE
    CALL fgl_winmessage("bol1 OR bol2 = FALSE", "bol1=TRUE    bol2=NULL \nThis
boolean expression -      LET x = bol1 OR bol2 \n is evaluated as FALSE", "info")
  END IF

  CALL fgl_winmessage("Exit", "Press any key to close this demo application", "info")
END MAIN

```

4GL attempts to evaluate both operands of AND and OR logical operators, even if the value of the first operand has already determined the returned value. The NOT operator is recursive.

Boolean Comparisons

Boolean comparisons can test any type of expression for equality or inequality, null values, or set membership, using the following Boolean operators:

- Relational operators to test for equality or inequality
- IS NULL (and IS NOT NULL) to test for null values
- LIKE or MATCHES to compare character strings
- IN to test for set membership
- BETWEEN...AND to test for range

The IN and BETWEEN...AND operators are valid only in SQL statements. They cause a compilation error if you include them in other 4GL statements. They are listed here, however, because they are valid in the WHERE clause of a form specification file that includes the COLOR attribute.

Boolean expressions in the CASE, IF, or WHILE statements (or in the WHERE clause of a COLOR attribute specification) return FALSE if any element of the comparison is null, unless it is the operand of the IS NULL or IS NOT NULL operator. Use a NULL test to detect and exclude null values in Boolean comparisons. In this CASE statement fragment, the value of the comparison is null if the value of **salary** or of **last_raise** is null:

```
WHEN x * y > 100
```

You can use any value that is not false (= 0) or null as a Boolean expression that returns TRUE. The constant TRUE, however, has the specific value of 1. Thus, the value FALSE is returned by a comparison like:

```
MAIN
  DEFINE
    str_input char(1)

  DISPLAY "The value of the constant TRUE is 1 - this means IF(100 = TRUE) is
false" at 1,1

  IF (100=TRUE) THEN
    CALL fgl_winmessage("IF (100=TRUE) is True","The boolean expression IF (100 =
TRUE) returns TRUE", "info") returning str_input
  ELSE
```

```
CALL fgl_winmessage("IF (100=TRUE) is FALSE","The boolean expression IF (100 =  
TRUE) returns FALSE", "info") returning str_input  
END IF  
  
IF (100) THEN  
    CALL fgl_winmessage("IF (100) is True","The boolean expression IF (100) returns  
TRUE", "info") returning str_input  
ELSE  
    CALL fgl_winmessage("IF (100) is FALSE","The boolean expression IF (100)  
returns FALSE", "info") returning str_input  
END IF  
  
CALL fgl_winmessage("Exit","Press any key to close this demo application","info")  
END MAIN
```

Relational Operators

These operators perform relational comparisons in Boolean expressions.

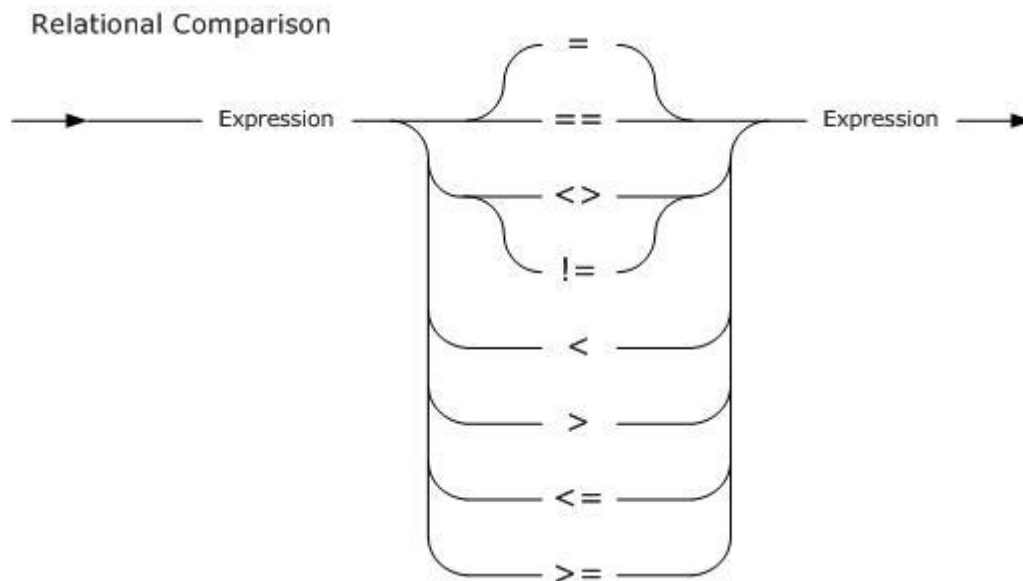


Figure 10 - Relational Operators

Boolean expressions in 4GL statements can use these relational operators (=, ==, <, >, <=, >=, <>, or !=) to compare operands. For example, each of the following comparisons returns TRUE or FALSE.

Expression	Value
$(22+8)/3 = 15$	FALSE
$5 \leq 10$	TRUE
"Tom" = "Dom"	FALSE

For character expressions, the result depends on the position of the initial character of each operand within the collation sequence. The collation sequence is the code-set order unless the locale files define a localized collation sequence in the COLLATION category, and **DBNLS** is set to 1. If the initial characters are identical in both strings, 4GL compares subsequent characters, until a non-identical character is found, or until the end of a string is found.

For number expressions, the result of a relational comparison reflects the relative positions of the calculated values of the two operands on the real line.

Relational comparisons of time expression operands follow these rules:

- Comparison $x < y$ is true when x is a briefer interval span than y , or when x is an earlier DATE or DATETIME value than y .
- Comparison $x > y$ is true when x is a longer interval span than y , or when x is a later DATE or DATETIME value than y .
- You cannot compare an interval with a DATE or DATETIME value, but you can compare DATE and DATETIME values with each other.

The value of the built-in constant TRUE is 1. A Boolean expression such as the following example returns FALSE unless **b** is exactly equal to 1:

```
IF (b = TRUE) THEN ...
```

To determine whether some value is not zero or null, avoid using TRUE in Boolean comparisons, and instead use expressions like:

```
IF (b) THEN ...  
IF (b != FALSE) THEN ...
```

Your code might be easier to read and might produce better results if you avoid using TRUE as an operand of the ==, =, !=, or <> operators.

MAIN

DEFINE

```
bol_value ARRAY[6] OF SMALLINT,  
str_display_expression ARRAY[6] OF CHAR(30),
```

```
str_display_v ARRAY[6] OF varchar(70),  
str_display_t, str_display_f char(70),  
counter SMALLINT
```

```
LET bol_value[1] = True  
LET str_display_expression[1] = "LET bol_value[1] = True"  
LET str_display_v[1] = "IF (TRUE)"  
LET bol_value[2] = 1  
LET str_display_expression[2] = "LET bol_value[2] = 1"  
LET str_display_v[2] = "IF (1)"  
LET bol_value[3] = 0  
LET str_display_expression[3] = "LET bol_value[3] = 0"  
LET str_display_v[3] = "IF (0)"  
LET bol_value[4] = FALSE  
LET str_display_expression[4] = "LET bol_value[4] = FALSE"  
LET str_display_v[4] = "IF (FALSE)"  
LET bol_value[5] = 100  
LET str_display_expression[5] = "LET bol_value[5] = 100"  
LET str_display_v[5] = "IF (100)"  
#LET bol_value[6] = <not assigned>  
LET str_display_expression[6] = "LET bol_value[6] is nothing"  
LET str_display_v[6] = "IF (NULL)"  
  
FOR counter = 1 to 6  
  
    LET str_display_t = str_display_expression[counter] || " - " || str_display_v[counter] || " = TRUE"  
    LET str_display_f = str_display_expression[counter] || " - " || str_display_v[counter] || " = FALSE"  
  
    IF bol_value[counter] THEN  
        CALL fgl_winmessage(str_display_expression[counter],str_display_t, "info")  
    ELSE  
        CALL fgl_winmessage(str_display_expression[counter],str_display_f, "info")  
    END IF  
END FOR  
  
CALL fgl_winmessage("Exit","Press any key to close this demo application","info")  
  
END MAIN
```

The NULL Test

If any operand of a 4GL Boolean comparison is NULL, the value of the comparison is FALSE (rather than NULL), unless the IS NULL keywords are also included in the expression. Applying the NOT operator to a null value does not change its FALSE evaluation.

To process expressions with null values in a different way from other values, you can use the IS NULL keywords to test for a null value.

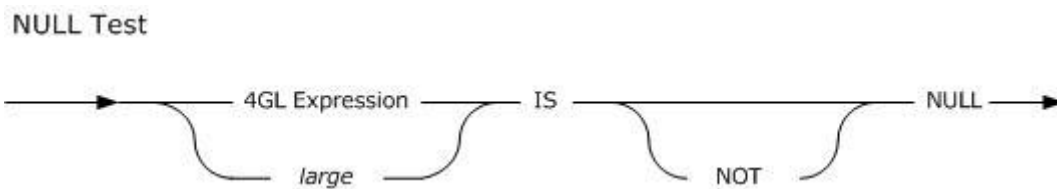


Figure 11 - NULL Test

In a null test, a 4GL expression can have a *large* element, which is a BYTE or TEXT data type declaring the name of a program variable.

Without the NOT keyword, the comparison returns TRUE if the operand has a null value. (If you include the NOT keyword, the comparison returns TRUE if the value of the operand is not null.) Otherwise, it returns FALSE.

The NULL test is one of the exceptions to the rule that BYTE or TEXT data type variables cannot be used in 4GL expressions.

like

The LIKE operator tests whether a character value matches a quoted string. That string can include wildcard characters. (The MATCHES operator works in a similar way, except that it supports different wildcards.)

If an operand has a null value, the entire string comparison returns NULL.

You can use the following syntax to compare character strings:



Figure 12 - LIKE Operator

The value for *mask* is a character expression, which can include the percent and underscore symbols. The % symbol can represent any number of characters; an underscore replaces only one character. If you want to use the percent or underscore symbols as literal characters you will need to put a backslash (\) in front of each of them.

In this example, ESC is a single character, between single or double speech marks, that specifies an escape symbol. The default escape key is the backslash (\).

The string that the operator will return can include literals, wildcards and other symbols.

The following WHERE clause tests the contents of character field field1 for the string: 'board'. Here the * wildcards specify that the comparison is true if 'board' is found alone or in a longer string, such as boarding or game board:

```
COLOR = RED WHERE field1 MATCHES "**board**"
```

If you use the keyword LIKE to compare strings, the wildcard symbols of MATCHES have no special significance, but you can use the following wildcard characters of LIKE within the right-hand quoted string.

Symbol	Effect in LIKE Expression
%	A percentage sign matches any number of characters, including none.
_	An underscore matches a single character.

The next example tests for the string 'board' in the character variable board, either alone or in a longer string:

```
IF string LIKE "%board%"
```

The next example tests whether a substring of a character variable (or else an element of a two-dimensional array) contains an underscore symbol. The backslash is necessary because underscore is a wildcard symbol with LIKE.

```
IF string_array[3,8] LIKE "%\__%" WHERE >> out.a
```

You can replace the backslash as the literal symbol. If you include an ESCAPE char clause in a LIKE or MATCHES specification, 4GL interprets the next character that follows char as a literal in the preceding character expression, even if that character corresponds to a special symbol of the LIKE or MATCHES keyword. The double speech mark (") symbol cannot be used as a char variable.

For example, if you specify `ESCAPE z`, the characters `z_` and `z?` in a string stand for the literal character `_` and `?`, rather than wildcards. Similarly, characters `z%` and `z*` stand for the characters `%` and `*`. Finally, the characters `zz` in the string stand for the single character `z`. The following expression is true if the variable `product` does not include the underscore character:

```
NOT product LIKE "%z_% " ESCAPE "z"
```

matches

The `MATCHES` operator tests whether a character value matches a quoted string. That string can include wildcard characters. (The `LIKE` operator works in a similar way, except that it supports different wildcards.)

If an operand has a null value, the entire string comparison returns `NULL`.

You can use the following syntax to compare character strings:

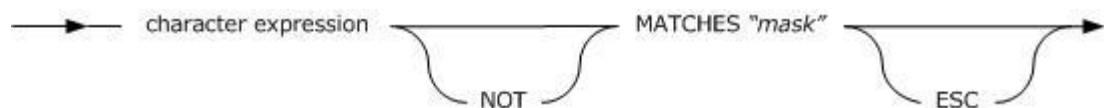


Figure 13 - MATCHES Operator

The value for *mask* is a character expression, which can include the asterisk (`*`) and question mark symbols. The `*` symbol can represent any number of characters, a question mark replaces only one character. If you want to use the asterisk or question mark as literal characters you will need to put a backslash (`\`) in front of each of them. Square brackets [] around a group of characters will cause any of those characters to be matched.

In this example, `ESC` is a single character, between single or double speech marks, that specifies an escape symbol. The default escape key is the backslash (`\`).

A hyphen between characters that are between brackets [f-k], causes any of that range of characters to be matched. This example is case sensitive.

A caret (or circumflex) at the beginning of a sequence of characters placed within brackets, such as [^xyz], causes any character to be matched, other than those listed between the brackets.

The following `WHERE` clause tests the contents of character field `field1` for the string: `'board'`. Here the `*` wildcards specify that the comparison is true if `'board'` is found alone or in a longer string, such as `boarding` or `game board`:

```
COLOR = RED WHERE field1 MATCHES "*board*"
```

If you use the keyword LIKE to compare strings, the wildcard symbols of MATCHES have no special significance, but you can use the following wildcard characters of LIKE within the right-hand quoted string.

The next example tests for the string 'board' in the character variable, with the word 'board' either alone or in a longer string:

```
IF string LIKE "%board%"
```

The next example tests whether a substring of a character variable (or else an element of a two-dimensional array) contains an underscore symbol. The backslash is necessary because underscore is a wildcard symbol with LIKE.

```
IF string_array[3,8] LIKE "%\_%" WHERE >> out.a
```

You can replace the backslash as the literal symbol. If you include an ESCAPE char clause in a LIKE or MATCHES specification, 4GL interprets the next character that follows char as a literal in the preceding character expression, even if that character corresponds to a special symbol of the LIKE or MATCHES keyword. The double speech mark (") symbol cannot be used as a char variable.

For example, if you specify ESCAPE z, the characters z_ and z? in a string stand for the literal character _ and ?, rather than wildcards. Similarly, characters z% and z* stand for the characters % and *. Finally, the characters zz in the string stand for the single character z. The following expression is true if the variable product does not include the underscore character:

```
NOT product LIKE "%z\_%" ESCAPE "z"
```

Set Membership and Range Tests

The BETWEEN...AND, and IN () operators that test for set membership or range are supported for 4GL programs in three contexts:

- In SQL statements
- In the WHERE clause of the COLOR attribute in 4GL form specifications
- In the condition column of the syscolatt table

They are not valid in 4GL statements that are not also SQL statements.

The following diagram shows the syntax for using the IN () operator to test for set membership.

Set Membership Test

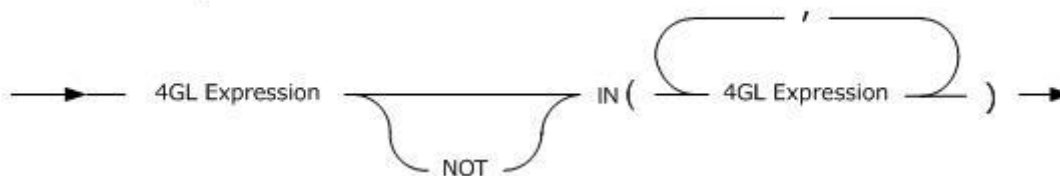


Figure 14 - Set Membership Test

If you omit the NOT keyword, this test returns TRUE if any expression in the list (within parentheses) at the right matches the expression on the left.

If you include the NOT keyword, the test equates to FALSE if no expression in the list matches the expression on the left.

The following diagram shows the syntax for the BETWEEN...AND operators to test whether a value is included within a specified range (or an inclusive interval on the real line).

Range Test



Figure 15 - Range Test

Operands must return compatible data types. Values returned by the second (O2) and third (O3) operands that define the range must follow these rules:

- For number or INTERVAL values, O2 must be less than or equal to O3.
- For DATE and DATETIME values, O2 must be no later than O3.
- For character strings, O2 must be earlier than O3 in the ASCII collation sequence.

Where non-default locales are being used, the code-set order (or whatever order of sorting the COLLATION category specifies in the locale files) replaces the ASCII collation order. The **DBNLS** environment variable must also be set to 1 for 4GL to support non-code-set sorting of string values.

If you omit the NOT keyword, this test evaluates as TRUE if the first operand has a value not less than the second operand or greater than the third. If you include the NOT keyword, the test evaluates as FALSE if the first operand has a value outside the specified range.

Data Type Compatibility

You might get unexpected results if you use relational operators with expressions of dissimilar data types. In general, you can compare numbers with numbers, character strings with strings, and time values with time values.

If a time expression operand of a Boolean expression is of the INTERVAL data type, any other time expression to which it is compared by a relational operator must also be an INTERVAL value. You cannot compare a span of time (an INTERVAL value) with a point in time (a DATE or DATETIME value).


Evaluating Boolean Expressions

In contexts where a Boolean expression is expected, 4GL applies the following rules after it evaluates the expression:

- The Boolean expression returns the value TRUE if the expression returns a nonzero real number *or* any of the following values:
 - A character string representing a nonzero number
 - A nonzero INTERVAL value
 - Any DATETIME or DATE value (except 31 December 1899)
 - A true value returned by any Boolean function or operator
 - The integer constant TRUE
- The Boolean expression returns the value TRUE if the value is null and the expression is the operand of the IS NULL operator.
- The Boolean expression returns null if the value of the expression is null *and* the expression appears in none of the following contexts:
 - An operand of the IS NULL or IS NOT NULL operator
 - An element in a Boolean comparison
 - An element in a conditional statement of 4GL (IF, CASE, WHILE)
- Otherwise, the Boolean expression returns the value FALSE.

Operator Precedence in Boolean Expressions

If a Boolean expression has several operators, they are processed according to their precedence. Operators that have the same precedence are processed from left to right.

	<p>The relative precedence values are ordinal numbers.</p> <p>The precedence of 4GL operators is described in "Operators in 4GL Expressions" in the Datatypes volume of this guide.</p>
---	---

The following table lists the precedence (**P**) of the Boolean operators of 4GL and summarizes the data types of their operands.

All of the expressions in this table return a Boolean value.

p	Description	Expression	Left (=x)	Right (=y)
9	String comparison	x LIKE y	Character	Character
	String comparison	x MATCHES y	Character	Character
8	Test for: less than	x < y	Any simple data type	Same as x
	Less than or equal to	x <= y	Any simple data type	Same as x
	Equal to	x = y or x == y	Any simple data type	Same as x
	Greater than or equal to	x >= y	Any simple data type	Same as x
	Greater than	x > y	Any simple data type	Same as x
	Not equal to	x != y or x <> y	Any simple data type	Same as x
7	Test for: set membership	x IN (y)	Any	Any
6	Test for: range	x BETWEEN y AND z	Any	Same as x
5	Test for: NULL	x is NULL	Any	
	Test for: NULL	x IS NOT NULL	Any	
4	Logical inverse	NOT y		Boolean
3	Logical intersection	x AND y	Boolean	Boolean
2	Logical union	x OR y	Boolean	Boolean
1	Test whether field is edited	FIELD_TOUCHED(y)		Field name
	Test for current field	INFIELD(y)		Field name

Besides the Boolean operators listed in this table, the built-in 4GL operators FIELD_TOUCHED() and INFIELD() also return Boolean values. Their precedence is lower (**P** = 1) than that of the OR operator. They can use the name of a field in the current form as their operand. Both the INFIELD() and FIELD_TOUCHED() operators are described elsewhere in this volume.

References

FIELD_TOUCHED(), INFIELD()

Bitwise Operators

These operators are used to compare and operate on values based on certain rules. The operators are as follows:

and (&)

This compares the bits of two numbers, equal in length, with the result only being a 1 if both of these bits are also 1. If any of the two numbers being compared are 0, then the result is a 0.

For example:

```
01100110 & 10011100 = 0000100
```

or (|)

The OR operator again compares the bits of two numbers, equal in length, and the result is a 1 if either of the two bits being compared is a 1, or both are a 1. If neither bit is a 1, then the result is 0.

For example:

```
00110011 | 11011001 = 11111011
```

xor (^)

XOR again takes two numbers of equal lengths and compares the bits of both. If the bits being compared are different, i.e., a 0 and a 1, then the result is a 1. Otherwise, the result is a 0.

For example:

```
010110011 ^ 10101010 = 11110001
```

not (~)

Finally, the NOT operator takes only one value and inverts its bits; a 1 becomes a 0, and a 0 becomes a 1.

For example:

$\sim 01100011 = 10011100$

This chapter describes all the built-in functions that can be used with Lycia to create and enhance 4GL applications. They are listed below in categorised order.

Database

fgl_column_info()

The `fgl_column_info()` function returns information about a specified database column.

Usage

The function takes two arguments, both of which are CHAR(64) datatype variables. The first should be the table name and the second the column name.

The information it returns are INTEGERS which report the column size and column type.

```
DATABASE cms

MAIN
    DEFINE sqltype INTEGER,
           sqllen INTEGER,
           temp_string char(100)

    CALL fgl_column_info("contact","cont_name")
           RETURNING sqltype, sqllen

    DISPLAY "The database column information for" at 5, 5
    DISPLAY "CMS - Table 'contact', Field 'cont_name' is" at 6, 5
    DISPLAY "SQL Type: ", sqltype AT 7, 5
    DISPLAY "Column Length: ", sqllen AT 8, 5

    CALL fgl_winmessage("Exit","Press any key to close this demo application","info")
END MAIN
```

fgl_driver_error()

The fgl_driver_error() function returns a CHAR(n) datatype value, which represents the last error received from the database interface.

Usage

```
DATABASE cms

MAIN
  DEFINE sql_stmt VARCHAR(255)
  DEFINE driver_error, native_error VARCHAR(255)

  LET sql_stmt = "Invalid SQL statement"

  WHENEVER ERROR CONTINUE
    PREPARE p1 FROM sql_stmt
    DECLARE c1 CURSOR FOR p1
    OPEN c1
  WHENEVER ERROR STOP

  LET driver_error = fgl_driver_error()
  LET native_error = fgl_native_error()

  DISPLAY "Driver error: "
  DISPLAY driver_error
  DISPLAY ""
  DISPLAY "Native error: "
  DISPLAY native_error

  CALL fgl_winmessage("Exit","Press any key to close this demo
application","info")
END MAIN
```

fgl_find_table()

The fgl_find_table() function will tell you if a specific table exists in the current database.

Usage

The function takes a CHAR(64) variable as its argument, which is the name of the table, and returns TRUE if the table exists within the current database.

```
DATABASE cms

MAIN

    IF fgl_find_table("contact") THEN
        DISPLAY "The table contact exists in CMS" at 5,5
    ELSE
        DISPLAY "The table contact DOES NOT exist in CMS" at 5,5
    END IF
    CALL fgl_winmessage("Exit","Press any key to close this demo application","info")
END MAIN
```

fgl_native_code()

When working with non-Informix RDBMS, Querix 4GL tries to provide the closest Informix error code on an error condition.

The `fgl_native_code()` function returns an INTEGER value, which represents the actual error code from the native RDBMS for the last SQL statement executed, as opposed to the Informix equivalent.

Sample Code: `fgl_native_code_function.4gl`

```
DATABASE cms

MAIN
  DEFINE sql_stmt VARCHAR(255)
  DEFINE driver_error, native_error VARCHAR(255)
  DEFINE native_code INTEGER

  LET sql_stmt = "Invalid SQL statement"

  WHENEVER ERROR CONTINUE
    PREPARE p1 FROM sql_stmt
    DECLARE c1 CURSOR FOR p1
    OPEN c1
  WHENEVER ERROR STOP

  LET driver_error = fgl_driver_error()
  LET native_error = fgl_native_error()
  LET native_code = fgl_native_code()

  DISPLAY "Driver error: " AT 5, 5
  DISPLAY driver_error AT 6, 5
  DISPLAY "" AT 7, 5
  DISPLAY "Native error: " AT 8, 5
  DISPLAY native_error AT 9, 5
  DISPLAY "Native error code: ", native_code AT 16,5

  CALL fgl_winmessage("Exit","Press any key to close this demo application","info")
END MAIN
```

fgl_native_error()

The fgl_native_error() function returns a CHAR(n) value, which represents the text of the error message of the native RDBMS for the last SQL statement executed.

Sample Code: fgl_native_error_function.4gl

```
DATABASE cms

MAIN
  DEFINE sql_stmt VARCHAR(255)
  DEFINE driver_error, native_error VARCHAR(255)
  DEFINE native_code INTEGER

  LET sql_stmt = "Invalid SQL statement"

  WHENEVER ERROR CONTINUE
    PREPARE p1 FROM sql_stmt
    DECLARE c1 CURSOR FOR p1
    OPEN c1
  WHENEVER ERROR STOP

  LET driver_error = fgl_driver_error()
  LET native_error = fgl_native_error()
  LET native_code = fgl_native_code()

  DISPLAY "Driver error: " AT 5, 5
  DISPLAY driver_error AT 6, 5
  DISPLAY "" AT 7, 5
  DISPLAY "Native error: " AT 8, 5
  DISPLAY native_error AT 9, 5
  DISPLAY "Native error code: ", native_code AT 16,5

  CALL fgl_winmessage("Exit","Press any key to close this demo application","info")
END MAIN
```

membership (.)

The membership operator is a period, or full stop (.). It specifies that the right-hand operand is a member of the set whose name is its left-hand operand. For example:

```
main_group.group_member
```

Where *group_member* is the name of a component within a structure, and *main_group* is the name of a RECORD variable, screen record, screen array of records, or a table, view or synonym. The *main_group* must have a component called *group_member*.

Usage

The *main_group* value can specify a screen record, screen array, RECORD variable, or database table, view, or synonym.

If *group_member* is the name of a database column, *main_group* can be qualified by a table qualifier.

The *main_group* value, *group_member* value, or both can be expressions that include the membership operator. For example:

```
l_contact.cont_name - member of a program or screen record
```

```
l_cont_arr[i].cont_id - field within a screen array
```

The LET statement syntax diagram (shown in the "Data Types" volume) illustrates the syntax of members that are RECORD variables. The sections on "THRU or THROUGH Keywords and .* Notation" in the DataTypes volume also contain a description of the members of a RECORD variable. In situations where more than one member is valid, you can substitute a wildcard asterisk (*) for *group_member*, to specify every member of *main_group*.

Cursor

This chapter contains the built-in function `cursor_name()` which returns the actual cursor name and the methods used to manipulate a cursor like an object rather than identifier. These methods are used with the variables of the `CURSOR` data type. These variables are declared like the variables of simple data types, thus the `CURSOR` variables allow you to use cursors of different scopes of reference and to pass them to functions. The way to use the methods is described [above](#).

cursor_name()

cursor_name("cursor_id")

The `CURSOR_NAME()` function takes as its argument the SQL identifier of an `UPDATE` or `DELETE` cursor, a `CURSOR` variable, or the identifier of a prepared statement, and returns the mingled name.

Usage

The Lycia compiler will *mingle* a cursor or prepared statement identifier, using a constant algorithm. This results in a new name, which will always be 18 characters long. The first half is derived from the user supplied name, and the second half, from the current system clock; because of this the names are guaranteed to be unique to the 4GL modules which reference them.

An example function call:

```
CALL cursor_name("curs") RETURNING c_name
```

close()

variable.Close()

This method is used with a CURSOR variable to close the cursor opened previously by the Declare() method used with the same variable. If you close an insert cursor using this method, it inserts all the records stored in the insert buffer into the corresponding table.

Usage

The Close() method accepts no arguments and closes the cursor associated with the variable. The variable should be initiated by the Declare() method prior to that and opened by the Open() method before you can use the Close() method with it. After the cursor is closed, you can reopen it with the Open() method again.

This method returns sqlca.sqlcode which is 0, if the cursor was closed properly and which has a negative value indicating the error number, if something went wrong while executing the method.

An example method call:

```
CALL cur_v.Close() RETURNING err_code
```

declare()

variable.Declare(statement [, with_scroll, with_hold])

This method is used to initialize a CURSOR variable and to assign a cursor to it together with the statement it is declared for and all other features. After the cursor is declared in this way, it can be referenced by other methods used with cursors. Typically it would then be used with the Open() method which will open the declared cursor and make it ready to fetch rows.

Usage

The Declare() method accepts three arguments.

The statement argument can be a character variable or a quoted character string containing the text of the SQL statement associated with the cursor; it can also be represented by a variable of the PREPARED data type which has previously been initialized using the Prepare() method. The SQL statement can be either a SELECT statement or an INSERT statement, depending on whether you want to declare a select or an insert cursor. When declaring a cursor for a SELECT statement, you should include the FOR UPDATE keywords to the statement text, if you want to use the cursor for updating data.

The *with_scroll* argument specifies whether the cursor should be declared WITH SCROLL. This arguments is of the BOOLEAN data type and is FALSE by default. If you want to declare a cursor WITH SCROLL (applicable only for the cursors declared for a SELECT statement), you should set this argument to 1, otherwise to 0)

The *with_hold* argument specifies whether the cursor will be declared WITH HOLD - whether it will be closed at the end of a transaction automatically. If you don't want the cursor to be closed automatically, set this argument to 1. The default value is 0 (FALSE).

This method returns sqlca.sqlcode which is 0, if the cursor was closed properly and which has a negative value indicating the error number, if something went wrong while executing the method.

The examples of method calls:

```
# Declaring a cursor for a normal SQL statement without hold and without scroll
DEFINE cur_v CURSOR

CALL cur_v.Declare("SELECT * FROM customer WHERE customer.cust_id > ? FOR UPDATE")
RETURNING err_code

# Declaring a cursor with hold for a prepared statement
DEFINE cur_v CURSOR,
      prep_v PREPARED

CALL prep_v.Prepare("SELECT * FROM customer WHERE customer.cust_id > ?")
CALL cur_v.Declare(prepare_v, 0, 1) RETURNING err_code
```

Fetching Rows Methods

There are a number of methods which are used for fetching rows from database tables using an initialized CURSOR variable. These methods are synonymous with the FETCH statement. The CURSOR variable need to be used with the Declare() and Open() methods before it can be used with the fetching methods. They can be used only with a cursor declared for a SELECT statement.

These methods return sqlca.sqlcode, which is 0 if a row was properly fetched, and 100 if the NOT FOUND condition occurred. If an error occurred while fetching a row, these methods will return a negative number with the error code.

FetchNext()

variable.FetchNext()

This method is used to fetch the rows consequently from the active set. It can be used with both cursors declared WITH SCROLL and with the sequential cursors. This method moves the cursor to the next row in the active set and retrieves it. It does not accept any arguments.

```
CALL cur_v.FetchNext()
```

You can also use this method to verify whether there are no rows left to fetch, because this method returns sqlca.sqlcode which is 0, if there are still rows to fetch and 100, if the NOT FOUND condition occurs, e.g.:

```
WHILE (cur_v.FetchNext()=0)
```

FetchPrevious()

variable.FetchPrevious()

This method is used to fetch the previous row from the active set. It can be used only with cursors declared WITH SCROLL. This method moves the cursor to the previous row in the active set and retrieves it. It does not accept any arguments.

```
CALL cur_v.FetchPrevious()
```

FetchFirst()

variable.FetchFirst()

This method is used to fetch the first row from the active set regardless of the current cursor position. It can be used only with cursors declared WITH SCROLL. This method moves the cursor to the first row in the active set and retrieves it. It does not accept any arguments.

```
CALL cur_v.FetchFirst()
```

FetchLast()

variable.FetchLast()

This method is used to fetch the last row from the active set regardless of the current cursor position. It can be used only with cursors declared WITH SCROLL. This method moves the cursor to the last row in the active set and retrieves it. It does not accept any arguments.

```
CALL cur_v.FetchLast()
```

FetchRelative()

variable.FetchRelative(offset)

This method is used to fetch the row from the active set which depends on the current cursor position and is either above or below the current cursor position by the specified offset. It can be used only with cursors declared WITH SCROLL.

It accepts one argument which is either a literal integer or an integer variable. It specifies the number of rows the cursor should be moved either upwards or downwards from the current position. This method moves the cursor to the row specified by the offset and retrieves it. The argument can be either positive or negative integer. If the offset is represented by a positive integer (the plus sign is optional), the cursor will be moved down by the specified number of rows. If the offset is represented by a negative integer (in this case the minus sign should be specified explicitly), the cursor will be moved up by the specified number of rows.

If you specify 0 as the argument, this method will function in the same way as FETCH CURRENT statement - it will not move the cursor from the current row and will fetch it again.

```
CALL cur_v.FetchRelative(-3)
CALL cur_v.FetchRelative(0)
```

FetchAbsolute()

variable.FetchAbsolute(row_number)

This method is used to fetch the specified row from the active set regardless of the current cursor position. It can be used only with cursors declared WITH SCROLL. This method moves the cursor to the specified row and retrieves it. The absolute row number depends on the position of the rows located in the active set, e.g. it can depend on the ordering you applied to the SELECT statement for which the cursor was declared.

It accepts one argument which is the row position and is represented by a literal positive integer or an integer expression returning a positive integer. You cannot use a negative integer as the row number. If you try to use a row number below the active set, the NOT FOUND condition will occur.

```
CALL cur_v.FetchAbsolute(10)
```

flush()

variable.Flush()

This method is used to insert the records previously buffered by the Put() method into database tables. It can be used only with the cursor declared for the INSERT statement.

Usage

The Put() method adds a row to the buffer, and when the buffer is filled, the stored information is written to the database. The Flush() method is used to force the data insertion from the buffer to the database before the buffer is filled. The buffer is freed after the method execution. The method does not need any arguments.

Before the Flush() method is used, the variable should be initialized by the Declare() method and opened by the Open() method before you can use the Close() method with it. For the method to insert at least one row you need to execute at least one Put() method previously.

This method returns sqlca.sqlcode which is 0, if the records were inserted properly and which has a negative value indicating the error number if something went wrong while executing the method.

An example method call:

```
CALL cur_v.Flush() RETURNING err_code
```

free()

variable.Free()

This method is used to free a cursor associated with a CURSOR variable. It can be used with cursors of all types. After the CURSOR variable is freed, you can no longer use it with other methods until you declare the cursor again using the Declare() method.

Usage

The Free() method accepts no arguments and frees the cursor associated with the variable. The variable should be used with the Close() method before you can Free() it.

This method returns sqlca.sqlcode which is 0, if the cursor was freed properly and which has a negative value indicating the error number, if something went wrong while executing the method.

An example method call:

```
CALL cur_v.Free() RETURNING err_code
```

getName()

variable.GetName()

This method is used with a CURSOR variable to get the cursor name which is used internally by the RDBMS. It is synonymous to cursor_name() built-in function.

Usage

This method accepts no parameters and returns one value which is the mingled name of the cursor.

An example method call:

```
CALL cur_v.GetName() RETURNING cur_name
```


getStatement()

variable.GetStatement()

This method is used with a CURSOR variable. It returns the text of the SELECT or INSERT statement for which the cursor was declared. It can be used only if the CURSOR variable has been initialized using the Declare() method first.

Usage

This method accepts no parameters and returns one value which is the text of the SQL statement associated with the cursor assigned to the specified CURSOR variable.

An example method call:

```
CALL cur_v.GetStatement() RETURNING sql_stmt
```

open()

variable.Open()

This method is used with the variables of the CURSOR data type to open the cursor previously declared by the Declare() method.

Usage

This method accepts no parameters. If it is a select cursor and the SELECT statement contains any placeholders, you should call SetParameters() method before using the Open() method. For an insert cursor the SetParameters() method can be called either before the Open() method or after it.

This method returns sqlca.sqlcode which is 0, if the cursor was opened successfully.

An example method call:

```
DEFINE cur_v CURSOR,  
       par_v INT  
...  
CALL cur_v.Declare(SELECT * FROM customer WHERE customer.cust_id > ?)  
LET par_v = 105  
CALL cur_v.SetParameters(par_v) -- setting the parameter for the placeholder  
CALL cur_v.Open() RETURNING err_code
```

put()

variable.Put()

This method is used with a CURSOR variable only if it is an insert cursor. The Put() method is analogous to the PUT statement. Each Put() method execution passes a row to the insert buffer. If the buffer is filled and there is no room for the new row, the buffered rows are automatically written to the database and the buffer is freed. Therefore, in some cases the Put() method makes some rows to be written to the database, and in some it does not.

The records from this buffer are inserted into the database when either Flush() or Close() method is executed for the given cursor.

Usage

This function accepts no parameters. If the INSERT statement for which the cursor is declared contains placeholders, you should call the SetParameters() method before calling the Put() method. This method returns sqlca.sqlcode.

An example method call:

```
CALL cur_v.Put()
```

setParameters()

variable.SetParameters(value [, value...])

This method is used with a CURSOR variable for both the insert and the select cursors to set the values for the placeholders, if they are present in the SELECT or INSERT statements associated with the cursor.

Usage

This method accepts a number of parameters which data types, number and order depend on the placeholders for which they are to supply values. This method can be called either before or after the Open() method for an insert cursor and only before the Open() method for a select cursor. This method returns sqlca.sqlcode.

An example method call:

```
CALL cur_v.Declare("INSERT INTO customer VALUES (?, ?, ?, ?)")  
CALL cur_v.SetParameters("Thomas", "Jones", "35", "m")
```

setResults()

variable.SetResults(variable [, variable...])

This method is used with a variable of the CURSOR data type. It sets the output values for the cursor associated with the variable - it is typically used with a select cursor which SELECT statement has no INTO clause. This method can be called at any stage in the cursor lifecycle. Calling the SetResults() method will overwrite any output values previously specified.

Usage

This method accepts a number of parameters which data types, number and order depend on the output cursor values you want to set. Its function is analogous to the INTO clause of the FETCH or FOREACH statements - it specifies the variables into which the output from a SELECT statement associated with the cursor should be performed. It is not required if the statement associated with the cursor has the INTO clause, but if it is used, it will override the variables specified in the SELECT statement. The SetResults() method returns sqlca.sqlcode.

An example method call:

```
CALL cur_v.SetResults(ret_rec.*)
```

Prepared Statements

The methods described in this chapter are applied to variables of the PREPARED data type which are declared in the same way as variables of simple data types. These methods treat the cursor as an object associated with a variable. This makes a prepared statement more flexible. Unlike a prepared statement declared implicitly with the help of the PREPARE statement which has only the module scope of reference, a PREPARED variable can be declared with any scope of reference. Variables of the PREPARED data types can be passed to functions and returned by them. The usage of the methods is described [above](#).

execute()

variable.Execute()

This method is used with a variable of the PREPARED data type to execute the prepared statement or statements associated with this variable. It also returns output values, if the prepared statement produces any.

Usage

This method accepts no parameters. To be able to use this method you must first initialize the PREPARED variable using the Prepare() method. This method returns sqlca.sqlcode.

An example method call:

```
CALL prep_v.Execute()
```

free()

variable.Free()

This method is used with a variable of the PREPARED data type. It frees the memory allocated to the prepared statement and makes it possible to use this PREPARED variable to prepare another SQL statement.

Usage

This method accepts no parameters and returns sqlca.sqlcode. To be able to free a variable, it should be previously initialized using the Prepare() method.

An example method call:

```
CALL prep_v.Free()
```

getName()

variable.GetName()

This method is used with a PREPARED variable to get the name of the prepared statement associated with it which is used internally by the RDBMS.

Usage

This method accepts no parameters and returns one value, which is the name of the prepared statement.

An example method call:

```
CALL prep_v.GetName() RETURNING prep_name
```


getStatement()

variable.GetStatement()

This method is used with a PREPARED variable. It returns the text of the SQL statement or statements which have been prepared using the Prepare() method with the same variable. It can be used only if the PREPARED variable has been initialized using the Prepare() method first.

Usage

This method accepts no parameters and returns one value which is the text of the prepared SQL statement.

An example method call:

```
CALL prep_v.GetStatement() RETURNING cur_name
```

isNative()

variable.IsNative()

This method is used with a variable of the PREPARED data type. It verifies whether the SQL statement has been prepared natively. You need to initialize the variable using the Prepare() statement first which defines whether the statement should be prepared natively or not.

Usage

This method accepts no parameters and returns TRUE (or 1), if the statement was prepared natively, bypassing the SQL translator.

An example method call:

```
CALL prep_v.IsNative() RETURNING native
```

prepare()

variable.Prepare(statement [, isNative])

This method is used to initialize a variable of the PREPARED data type. It prepares an SQL statement for execution in the RDBMS, and allocates the necessary resources to the prepared statement.

After a statement is prepared, it can be executed using the Execute() method. An initialized PREPARED variable can also be used in the Declare() method to declare a cursor for the prepared statement

Usage

This method accepts two parameters and returns sqlca.sqlcode, if the statement was prepared without errors.

The *statement* argument is either a quoted character string or a variable of a character data type which contains the text of the SQL statement to be prepared. The *isNative* argument specifies whether the statement should be prepared natively or not, and is optional. It is a BOOLEAN argument and its default value is 0 (FALSE) which indicates that the statement is not prepared natively. If you set it to 1 (TRUE), the statement will be prepared natively bypassing the SQL translator. The value of this argument for an initialized PREPARED variable can be checked using the IsNative() method.

An example method call:

```
# Preparing and executing an SQL statement
DEFINE prep_v PREPARED

CALL prep_v.Prepare("EXECUTE PROCEDURE procl", 1)
CALL prep_v.Execute()
...
#Using a prepared statement in a cursor
DEFINE prep_v PREPARED,
      cur_v CURSOR
CALL prep_v.Prepare("SELECT * FROM customers WHERE cust_id>100")
CALL cur_v.Declare(prepare_v)
```

setParameters()

variable.SetParameters(value [, value...])

This method is used with a variable of the PREPARED data type. The variable must first be initialized using the Prepare() method. The SetParameters() method is used if the prepared statement contains any placeholders - it supplies values to substitute these placeholders.

Usage

This method accepts a number of arguments, their number, order and data types depending on the placeholders. The values are bound by reference, so any changes in the bound values will be reflected in the prepared statement.

You can call the SetParameters() method for the PREPARED variable, if this variable is then used independently of any cursor. If you use this variable in the Declare() method to associate the prepared statement with a cursor, you need to execute the SetParameters() method for the CURSOR variable declared by this method and not for the PREPARED variable.

An example method call:

```
# Using it for the PREPARED variable
DEFINE prep_v PREPARED

CALL prep_v.Prepare("SELECT * FROM customers WHERE cust_id>?")
CALL prep_v.SetParameters(100)
...
# Using it for a CURSOR variable
DEFINE prep_v PREPARED,
      cur_v CURSOR
CALL prep_v.Prepare("SELECT * FROM customers WHERE cust_id>?")
CALL cur_v.Declare(prepare_v)
CALL cur_v.SetParameters(100)
```

setResults()

variable.SetResults(variable [, variable...])

This method is used with a variable of the PREPARED data type to change the output values produced by the prepared statement. This method can be called either for the PREPARED variable, or for the CURSOR variable, if the PREPARED variable was used for this cursor. It is typically used for a prepared SELECT statement which has no INTO clause to specify where the retrieved values should be placed.

Usage

This method accepts a number of arguments. Their number, order and data types depend on the values returned by the prepared statement and must match.

You can call the SetResults() method for the PREPARED variable, if this variable is then used independently of any cursor. If you use this variable in the Declare() method to associate the prepared statement with a cursor, you need to execute the SetResults() method for the CURSOR variable declared by this method and not for the PREPARED variable.

An example method call:

```
CALL prep_v.Prepare("SELECT fname, lname, age FROM customers") -- selects 3 values
CALL prep_v.SetResults(r_fname, r_lname, r_age) -- provides the location for them
```

File I/O

channel

The Channel functions read in from or write to a file or a pipe. A pipe acts as an intermediary between one program's output and another's input. They allow other actions to be performed on the output rather than simply dumping it all at once. An example use would be like the pipe function in Linux with the 'ls' and 'less' functions to see the contents of a directory screen by screen. For example:

```
ls z:\docs
ls z:\docs | less
```

The 'ls' function will send the contents of 'docs' to the pipe which then sends the data to the screen, depending on the function that has been specified after the pipe symbol. In this case, the 'less' function will send the data to fill the screen and wait until the user prompts the function to send the next screen's worth of data.

The following functions are used to achieve the Channel functionality:

fgl_channel_open_file(name, filename, mode)

This function takes as its parameters the name of the file to be used within the 4GL code, the actual name of the file and the mode in which the file will be used (for example, read, write, etc.). With all these parameters provided, the function will then open the named file ready for use.

fgl_channel_open_pipe(name, command, mode)

This function requires the pipe name, the command to be performed on the data from the opened file and, again, the mode. It opens a pipe that is to be used to store the data from the file. This data can then be read from the pipe by other programs.

fgl_channel_set_delimiter(name, symbol)

The set_delimiter function takes as its parameters the file name (as created in the fgl_channel_open_file function), and the symbol that is used as a separator in the file. The function tells the program how the data in the file is split up.

fgl_channel_read(name, values)

This function is used to read the values from the named file or pipe into the 'values' parameter passed to it. If no delimiter has been set, then all the information in the file is read into the 'values' parameter. If, however, there is a delimiter defined then each item of information between these delimiters is entered into the 'values' parameter in order. In this instance, a loop is required to define different values to read to.

fgl_channel_write(name, values)

This is similar in functionality to the read function, except that instead of reading the values from file or pipe, those values are written to them.

fgl_channel_cat(name, values)

This function is the same as the write function, except that the write function writes data with a new line, and the cat function writes without a new line.

fgl_channel_close(name)

This file simply closes the file that has been used in the previous functions. It takes as its parameter the name of the file.

For the fgl_open_channel() and fgl_open_pipe functions there are several modes that can be used – the following table describes the modes available:

Mode	Description
R	Read mode. This mode opens a file or pipe in preparation for being read. When opened in read mode, a file can only be read using the fgl_channel_read() function.
W	Write mode. This mode opens a file or pipe in preparation for being written to. When using write mode only the functions fgl_channel_write() and fgl_channel_cat() can be used. Any information in the named file or pipe will be overwritten.
A	Append mode. This mode opens a file or pipe in preparation for being written to. When using append mode only the functions fgl_channel_write() and fgl_channel_cat() can be used. Append mode will keep the information in the named file or pipe and append the new values onto the end.
1 (one)	Raw mode. This mode, like Write and Append, opens a file or pipe for writing, except that when the data is written all control characters will be enacted and all delimiters will be ignored. When using this mode, only the functions fgl_channel_write() and fgl_channel_cat() can be used. Raw mode will overwrite any data held in the named file or pipe.

An example of using all of these functions follows:

```
fgl_channel_open_file("stream", "unl/contact.unl", "r")
```

Opens the file 'contact.unl' located on the working directory '/unl/contact.unl' and refers to the file in the 4GL as 'stream'. The 'r' tells the program that the file is to be opened for reading.

```
fgl_channel_open_pipe("pipe", input_pipe, "r")
```

This code will create a pipe called 'pipe' and depending on the input_pipe variable will either pass all the data, or pass it element at a time.

```
fgl_channel_set_delimiter("stream", "|")
```

The above will tell the program that in the file 'stream', values are separated by a pipe symbol.

```
fgl_channel_read("stream", buffer)
```

This will read the contents of 'stream' into the variable 'buffer'. If the values in stream are split up by a delimiter then only the first value will be put into 'buffer'. In order to read all values in a file into 'buffer' you would need to define 'buffer' as an array and set up a loop to populate it. For example:

```
DEFINE
    buffer ARRAY[20] OF CHAR[30]

LET i = 1
FOR i = 1 TO 20
    fgl_channel_read("stream", buffer[i])
END FOR
```

```
fgl_channel_write("stream", buffer)
```

The fgl_channel_write() function will write the contents of 'buffer' to 'stream'. This is writing to the actual file that was opened earlier.

```
fgl_channel_cat("stream", buffer)
```

As with the write function, the cat function will write the contents of 'buffer' to 'stream', except that data written to 'stream' will be written without new lines.

```
fgl_channel_close("stream")
```

This function will close the file 'stream' so that any further attempts to access it will fail.



4J's requires you to add the 'USE CHANNEL' construct at the start of your 4GL module if you want to use any of the channel functions. This construct is not needed with Lycia.

4J's also requires the use of square brackets when using a variant amount of arguments in a function call – Lycia does not require these square brackets, and does not support them.

Sample Code: channel_function.4gl

```
MAIN
  DEFINE buffer CHAR(128)
  DEFINE input_pipe CHAR(64)

  IF fgl_arch() = "nt" THEN
    LET input_pipe = "dir"
  ELSE
    LET input_pipe = "ls -l"
  END IF

  CALL fgl_channel_open_file("stream", "myfile.txt", "w")
  CALL fgl_channel_open_pipe("pipe", input_pipe, "r")
  CALL fgl_channel_set_delimiter("pipe", ",", ",")
  CALL fgl_channel_read("pipe", buffer)
  RETURNING ret
  CALL fgl_channel_write("stream", buffer)
  CALL fgl_channel_close("stream")

  DISPLAY "Data read from pipe: " AT 3, 4
  DISPLAY buffer AT 4, 4

  CALL fgl_winmessage("Dynamic 4gl Channel Functions","Press any key to exit this
application", "info")
END MAIN
```

DDE functions

DDE is a data exchange mechanism which allows programs to talk to each other. This can be used for one-off data exchanges, or continual transfers, that occur when data has been updated.

DDE only works on a Windows client, so for Querix this is currently supported by LyciaDesktop. Lycia supports data exchange with any application that supports DDE.

Using DDE

The DDE process follows these four stages:

Your 4GL application sends a DDE order to the Windows client (LyciaDesktop) using TCP/IP.

LyciaDesktop executes the DDE order and sends the data to the Windows application through the DDE.

The Windows application executes the command and sends the data (or an error code) to LyciaDesktop.

LyciaDesktop sends the result to your 4GL application, using TCP/IP.

There are several DDE functions that can be used for the control of data exchange:

DDEConnect

DDEConnect opens a connection to an application which supports DDE. It takes two arguments which are the name of the program that you want to exchange data with, and the document name, which is the name of the file in which the data is held.

```
DDEConnect (progrname, docname)
```

Both the program and file name are CHAR(128) data strings. The return value will be true if the connection has been opened successfully or false if an error occurred.

A DDE connection is represented by a unique identifier, which comprises the program name and the document name.

DDEExecute

The DDEExecute function executes a command in the specified document, using the open program.

```
DDEExecute (progrname, docname, command)
```

Both the program and file name are CHAR(128) data strings. The return value will be true if the connection has been opened successfully or false if an error occurred.

The third argument is optional and is a command to be carried out within the file that has been opened. This is a CHAR(2048) data string, the syntax of which is dependent on the program that is used to open the file. The program must be able to recognize the command.

DDEFinish

This is the same as DDEClose.

```
CALL DDEFinish (progrname, docname)
```

DDEFinish closes the connection channel to the program and document specified as arguments in the syntax of the call.

Both arguments are of the CHAR(128) datatype. The returned value is true if the channel was successfully closed.

DDEFinishAll

DDEFinishAll is used to close all DDE connections, and the program that is being communicated with via DDE.

```
DDEFinishAll () RETURNING ret
```

DDEFinishAll takes no arguments, it will close all open channels and related programs. It will return True if all channels are closed, or False if an error occurs. Any error details can be viewed using DDEGeterror.

DDEGetError

The DDEGeterror function retrieves the last error recorded for the DDE channel.

```
DDEGeterror ()
```

The function takes no arguments, returning only the last recorded error. If there are no recorded errors in the current session the return will be NULL.

DDEPeek

The DDEPeek function will get values from a specified place within a specific file, and store those values in a variable.

```
CALL DDEPeek (progrname, docname, cells)
```

All three of the arguments are of the CHAR(128) datatype. *Progrname* is the name of the operating program and *docname* is the name of the document opened by the program. *Cells* is the description of the place from where the data is taken, such as cell names in a spreadsheet.

If successful, the function will return the values from the specified locations; if an error occurs the return will be NULL. Any error details can be seen using the DDEGeterror function.

If more than one value is returned, they are each separated by a TAB character. Any new line characters are returned as ASCII 13 characters.

DDEPoke

The DDEPoke function sends data to the open document, and places it in the specified part of the document.

```
CALL DDEPoke (progrname, docname, cells, values)
```

All four of the arguments are of the CHAR(128) datatype. *Progrname* is the name of the operating program and *docname* is the name of the document opened by the program. *Cells* is the description of the place where the data is to be placed, such as cell names in a spreadsheet. *Values* is the data to be inserted in the place defined by *cells*.

The function will return TRUE if the data is transferred successfully, or FALSE if an error occurs. Any error details can be seen using the DDEGeterror function.

fgl_basename()

The file management functions allow for the management of files and directories. You are able to find the file name from a given path and similarly you can find just the directory from a given file path. You are also able to create and remove directories and paths.

Usage

`fgl_basename(path, [suffix])` returns `char(n)` path.

The `fgl_basename()` function returns only the file name of a given path, and optionally strips the suffix from the file. For example:

```
fgl_basename("/tmp/docs/productX/xxx")
```

will return 'xxx', as will the call:

```
fgl_basename("/tmp/docs/productX/xxx.4gl", ".4gl")
```

where the suffix, `'.4gl'` has been specified in the function call, thereby removing the `'.4gl'` extension. If an incorrect suffix is specified but the file does exist, then the function will return the file name with that suffix. The function has been designed so that the file suffix is not necessarily the file extension; it simply returns the last section of the path. It is assumed that this is a file, but it may be a folder, in which case the folder is returned.

fgl_dirname()

The file management functions allow for the management of files and directories. You are able to find the file name from a given path and similarly you can find just the directory from a given file path. You are also able to create and remove directories and paths.

Usage

`fgl_dirname(path)` returns `char(n)` path

The `fgl_dirname()` function returns the directory part of a path provided to it, removing the file name. For example:

```
fgl_dirname("/tmp/docs/productX/xxx.4gl")
```

will return `"/tmp/docs/productX"`. As with `fgl_basename()`, the path is assumed to be a directory and file name so the function simply returns the path minus the last section. If this last section is a folder instead of a file, then that folder is removed anyway.

fgl_download()

Lycia supports transferring files between client and server, which is supported by corresponding built-in functions. The `fgl_download()` function is used to pass files from server to client.

Usage

The `fgl_download()` function allows to retrieve a file stored on the server and pass it to the client. The function needs two arguments to be specified:

```
fgl_download(server_file, client_file)
```

Server_file - a string value specifying the path, the name and the extension of the file to be downloaded.

Client_file - a string value specifying the file target location, name and extension after downloading.

Both arguments can specify the full path as well as the relative one.

For example, the following function call may be used to get a copy of a file located on another machine at `C:\Docs\Howto.doc` and move it to a different location on our own machine:

```
fgl_download("C:\\Docs\\Howto.doc", "Z:\\MyFiles\\Guides\\Howto.doc")
```

If the specified file does not exist, file transfer is not performed, and the function returns 0/FALSE. If the operation is performed successfully, the function returns 1/TRUE.



The function `fgl_download()` replaces the function `fgl_file_to_client()`, retaining the same functionality. There is also the `fgl_upload()` function available, and it is used to pass files from client to server.

fgl_grid_export()

The `fgl_grid_export()` function allows an application developer to export a grid's contents to a clipboard or file, in either a text or html format. This function passes five parameters; as well as the Record Name, Export Type and Format, the Start and End Index values can also be specified which are applicable to the grid. Specifying an Index range will allow the developer to define which rows of the grid are going to be exported. Please see below for a syntax example (where we have expressed rows 1 to 5 as the ones that are to be exported):

```
fgl_grid_export(scr_contacts,1,5,"clipboard","html")
```

Please note that the filename parameter is only required when specifying 'file' as the export type.

Two export formats are available: HTML and CSV. To express CSV as a parameter you can use the following script:

```
fgl_grid_export(scr_contacts,1,5,"file","csv",[my_file_name])
```

It is important to note that for the `fgl_grid_export()` function that where a Type of 'file' has been entered and a filename is not specified, the client will display a Save File dialog box allowing users to enter a filename and file location.

fgl_grid_viewexport()

This works in the same way as `fgl_grid_export()` except that `start_index` and `end_index` are omitted, as in effect they are automatically filled with the current active grid's visible view.

```
fgl_grid_viewexport("scr_contacts", "file", "csv", [my_file_name])
```

For the two grid export functions, Lycia allows you to specify the divider when using 'csv' as the export format. This is useful for documents that are separated by other characters other than commas. For example, to specify that the document in the above example is a pipe separated document, use the following line of code prior to any export function call:

```
CALL fgl_setproperty("gui", "array.grid.CSVSeperator", "|")
```

fgl_mk()

The file management functions allow for the management of files and directories on the application server. You are able to find the file name from a given path and similarly you can find just the directory from a given file path. You are also able to create and remove directories and paths on the server side.

Usage

`fgl_mk(file_name)`

This function creates a file at the path passed to it. It returns TRUE on success and FALSE on failure. Again, it will create the file at the current working directory, unless told otherwise. For example:

```
fgl_mk("/docs/Help")
```

would create the Help file at /docs/.

fgl_mkdir()

The file management functions allow for the management of files and directories on the application server. You are able to find the file name from a given path and similarly you can find just the directory from a given file path. You are also able to create and remove directories and paths on the server side.

Usage

`fgl_mkdir(directory)`

This function will attempt to create the directory passed to it. The function returns TRUE on success, and FALSE on failure. For example:

```
fgl_mkdir("/docs/productX/help")
```

would create the directory "/docs/productX/help". The default location for creating this directory is the current working directory. If it is required that the directory is created on another drive, then you are able to specify this, though care should be taken as specifying drives is not standard across different operating systems.



If you use a CHAR variable as the function argument, remember, that the trailing white spaces are added to the CHAR variable, if the value assigned to it has fewer characters than the declared variable size. It is highly possible that the directory name will contain all these trailing white spaces. Use VARCHAR variables to avoid this effect.

fgl_rm()

The file management functions allow for the management of files and directories on the application server. You are able to find the file name from a given path and similarly you can find just the directory from a given file path. You are also able to create and remove directories and paths on the server side.

Usage

`fgl_rm(file_name)`

This function removes the named file if it exists, and returns TRUE on success and FALSE if it fails. For example, to remove the file 'HowTo' use the following line of code:

```
fgl_rm("HowTo")
```

If you only specify the filename, the function will search in the current working directory for that file. It is also possible to pass an entire path too.

fgl_rmdir()

The file management functions allow for the management of files and directories on the application server. You are able to find the file name from a given path and similarly you can find just the directory from a given file path. You are also able to create and remove directories and paths on the server side.

Usage

`fgl_rmdir(directory)`

`fgl_rmdir()` has a directory passed to it and deletes it if it exists. The return values for it are TRUE on success and FALSE on failure. For example:

```
fgl_rmdir("/docs/productX/help")
```

would remove the directory `"/docs/productX/help"`. If the directory specified contains subfolders or files then the function will not remove it.

fgl_test()

The `fgl_test()` function can be used to test the application server's file system for the presence of files and for some of their attributes.

The function takes two arguments, the first being a `CHAR(1)`, which defines the test type (shown in the list below), and the second being a `CHAR(255)`, which is the name of the file on which the test is to be done.

The function returns a `SMALLINT`, which will be `TRUE` or `FALSE`, depending on the result of the test.

Test types for the `fgl_test()` function:

- `'e'`: named file exists
- `'l'`, `'h'`: named file is a symbolic link
- `'x'`: named file is executable by user
- `'r'`: named file is readable by user
- `'w'`: named file is writeable by user
- `'s'`: named file is non-zero size
- `'d'`: named file is a directory

Code Sample: fgl_test_function.4gl

```
MAIN

DEFINE
    file_path VARCHAR(20),
    env_variable CHAR(60),
    output_string1, output_string2 CHAR(80)

CALL fgl_winprompt(5, 2, "Enter the name of a file", "", 20, 0)
    RETURNING file_path
#LET env_path = fgl_getenv(env_variable)

LET output_string1 = "The file " || file_path || " does not exist"
LET output_string2 = "The file " || file_path || " exists"

IF NOT fgl_test("e",file_path) THEN -- check if the file exists (not exists
returns NULL)
    DISPLAY output_string1 CLIPPED AT 5,5
ELSE
    DISPLAY output_string2 CLIPPED at 5,5
END IF

sleep 5
END MAIN
```

fgl_upload()

Lycia supports transferring files between client and server, which is supported by corresponding built-in functions. The fgl_upload() function is used to pass files from client to server.

Usage

The fgl_upload() function allows to retrieve a file stored on the client and pass it to the server. The function needs two arguments to be specified:

```
fgl_upload(client_file, server_file)
```

Client_file - a string value specifying the path, the name and the extension of the file to be downloaded.

Server_file - a string value specifying the file target location, name and extension after downloading.

Both arguments can specify the full path as well as the relative one.

For example, to send a file located at C:\Docs\Help.doc on your machine to the application server use the following function call:

```
fgl_upload("C:\\Docs\\Help.doc", "Help.doc")
```

If the specified file does not exist, file transfer is not performed, and the function returns 0/FALSE. If the operation is performed successfully, the function returns 1/TRUE.

base.CHANNEL Class

The Channel class is the class introduced to provide read and write access to files as well as some functionality for communicating with the application sub-processes.

There are some restrictions that you should keep in mind when using the Channel class:

- The escape character in reading or writing actions should be "\"
- For both input and output, the code-set used in the target file should correspond to your system locale settings. The system does not perform any character set conversion.

The base.CLASS objects need to be declared with the help of the DEFINE statement:

```
DEFINE chan_obj base.CHANNEL
```

The Create() method is then used to initialize the variable. It is a class method called with the "base.Channel." prefix before it.

Create()

Channel class needs instantiation. This means, that you have to define variables that belong to the class and then apply the methods to these variables.

After a Channel variable is declared, one can activate it by means of the Create() method. To create a variable you should use the LET statement together with the. The method returns an object of base.Channel type and accepts no arguments. Create statement following the class name and namespace prefix:

```
DEFINE chan_obj base.Channel  
LET chan_obj = base.Channel.Create()
```

SetDelimiter()

SetDelimiter() is an optional method used to specify a delimiter other than the default one. The default delimiter is specified in DBDELIMITER or, if it is not set, "|" is used. This method accepts a single parameter that is the delimiter sign surrounded by quotes.

An example of the method invocation is given below:

```
CALL chan_obj.setDelimiter("^")
```

If no argument is specified, the delimiter is set to NULL. In this case, the "\" symbol will not be used by Read() and Write() methods as an escape character.

It is possible to make the application read and write in Comma Separated Value format. To switch to this mode, pass "CSV" as the delimiter to the SetDelimiter() method.

```
CALL chan_obj.setDelimiter("CVS")
```

There are several differences between this format and the standard Channel format, they are listed below:

- The file values may contain the double quotes (" ").
- The values in the file must be surrounded by quotes. The coma or the newline symbols does not make the method escape the value processing.
- The output value must be taken in quotes. If values contain double quotes, these symbols are doubled when sent to the output file.
- If a read/write operation encounters a backslash symbol, it is read and the value is not escaped. The value should be taken in quotes.
- Leading or trailing spaces are not truncated.
- The record lines don't have to end with an ending delimiter.

Opening a Channel

Before a channel object can be used, a channel associated with it must be opened. It can be either a file, a socket or a pipe. If a channel is not opened, any other methods associated with it will have no effect and will return a runtime error.

OpenFile()

The OpenFile() method is used to open a file for writing, reading, or performing both these operations. The method requires two parameters of a character data type, the file name with optional path and the opening flag:

```
CALL channel_name.openFile(filename, opening_flags)
```

The *opening flag* argument can be represented by one of the following characters:

Mode	Flag	Effect
Read mode	r	Read from the file. If <i>Path</i> to the file is not specified, the input is performed to the standard folder.
Write Mode	w	Write to an empty file. If <i>Path</i> to the file is not specified, the output is performed to the directory where the executable file is located. If the file does not exist, it will be created.
Append Mode	a	Write to the end of the file (append). If <i>Path</i> to the file is not specified, the output is performed to the directory where the executable file is located. If the file does not exist, it will be created.

Read And Write	u	Read and Write from and to the standard output and input.
Binary mode	b	This flag can follow any of the listed above (e.g., "wb"). Used to avoid carriage return/line feed (CR/LF) translation.

Here is an example of OpenFile() method usage:

```
CALL chl.openFile("mytext.txt", "r")
```

This command will open the file named "mytext.txt" in the folder on the application server where the program executable is located. If the program executable is located in a sub-folder, the file will be opened or created in this sub-folder.

OpenPipe()

The OpenPipe() method provides you with the ability to write to the standard output, to read from an sub-process standard output, or to perform both these operations.

The method requires two parameters of a character data type to be specified, the command name and the opening flag:

```
CALL channel_name.openPipe(command, opening_flag)
```

The opening_flag is a quoted symbol or a variable containing one of the following flags:

Mode	Flag	Effect
Read Only	r	Perform a reading operation from the command standard output.
Write Only	w	Perform a writing operation to the command standard input.
Write Only	a	Perform a writing operation to the command standard input by appending the existing content.
Read And Write	u	Perform both reading and writing operations with the command standard input.

OpenClientSocket()

The OpenClientSocket() method is used to set up a TCP connection with a server. The method invocation needs four arguments to be passed, the host name, the port number, the opening flags and the timeout value:

```
CALL chl.openClientSocket(host, port_number, opening_flag, timeout)
```

The host argument is the quoted string or a variable containing the name or the IP address of the host machine to connect to. The port number should be an integer value specifying the service port number.

The timeout argument should contain an integer value specifying the timeout in seconds after which the program will no longer wait for the response from the socket. If it is set to -1 the timeout is disabled and the program will wait for the response forever.

The opening_flags is a quoted string or a variable containing one or two opening flags, which are listed in the following table:

Mode	Flag	Effect
Read Mode	r	Specifies that the program only reads from the socket and does not write to it.
Write Mode	w	Specifies that the program only writes to the socket and does not read from it.
Read and Write Mode	u	The program reads from and writes to the socket.
Binary Mode	b	The socket will be opened in a binary mode to avoid the translation of the carriage return and line feed symbols.

Below is given an example of the method usage:

```
CALL chl.openClientSocket("localhost", 9090, "ub", 50)
```

Closing a Channel

The Close() method is used to close the opened channel after you finished working with it.

The method does not require parameters:

```
CALL chl.Close()
```

A channel can be also closed automatically as soon as all the references to it are deleted.

readLine()

The ReadLine() method is used to ignore delimiters specified by SetDelimiter() method and to read a complete line from the opened Channel:

```
LET gotline = chl.ReadLine()
```

If the reading operation (for example, activated in a loop) reaches the end of the file, the ReadLine() method returns NULL. It returns an empty string, if the line fetched is empty. If the value is returned into a CHAR or VARCHAR variable, the empty lines and the end of line will be both treated as empty strings. Therefore, it is necessary to distinguish empty lines from the end of the file. To do this, use the STRING data type.

However, the most accurate way to find the end of the file is `IsEOF()` method.

If this method is used with a file, pipe or socket opened with 'u' flag, the program waits for the input from the standard input source which is the keyboard by default. The input of the line in this case finishes with the pressing of the Enter key or with the symbol of a new line.

writeLine()

The `WriteLine()` method is used to ignore delimiters specified by `SetDelimiter()` method and to write a complete line to the opened Channel:

```
CALL ch1.WriteLine("The line passed to the Cahannel")
```

If this method is used with a file opened with 'u' flag, the program outputs the line to the console window which is the standard output channel.

isEOF()

The `IsEOF()` method is used to indicate the end of the file being read from the a Channel. The method needs no arguments. It returns `TRUE`, if the read operation has reached the end of the file and `FALSE` if not. It is convenient to use the method in a conditional statement:

```
WHILE TRUE
    LET gotarray[i] = ch1.ReadLine()
    LET i = i+1
    IF ch1.IsEOF() THEN EXIT WHILE END IF
END WHILE
```

read()

The `Read()` method is used to read data records in which delimiters are used to separate fields. When using this method, you must specify the list of the variables which will receive the read values. The list should be given in square braces:

```
CALL ch1.Read([val1,val2,val3])
```

Here, you can also use a variable of `RECORD` data type:

```
CALL ch1.Read([myrec.*])
```

If the `Read()` method successfully reads the data, it return `TRUE`. This means that the values specified in the method arguments are searched through the file and, if they are found, `TRUE` is returned, and the found values are not modified in any way. Otherwise, the returned value is `FALSE`, which means that the method has reached the end of the stream or the file and was not able to find the specified values.

For example, if the method can be used for the following file:

```
1|22|a|bb|
12|4111|xxxaa|bbbzzz|
```

```
333||x||
```

In this case the code looking for the values in the last line of this file will be as follows:

```
MAIN
DEFINE ch base.Channel,
    rec RECORD
        a,b INTEGER,
        c,d CHAR(20)
    END RECORD

LET ch = base.Channel.create()
CALL ch.openFile("C:/datafiles/file1.txt", "r")

LET rec.a = 33
LET rec.b = null
LET rec.c="x"
LET rec.d=null

DISPLAY ch.read([rec.*]) -- will return 1 (TRUE), because the line can be read.

END MAIN
```



Note: the channel must be open for reading, otherwise this method will throw a runtime error. Make sure that the method for opening the channel includes u or r flag as a parameter.

write()

The Write() method is used to write data records to a channel and add delimiters to separate fields. When using this method, you must specify the list of the variables from which the values will be taken. The list of variables must be given in square braces:

```
CALL chl.write([val1,val2,val3])
```

As with the Read() method, variables of RECORD data type are allowed:

```
CALL chl.Write([myrec.*])
```

Note that the write mode will depend on the way you opened the channel. Thus if a channel was opened for writing (w flag) and you write to a file, the contents of the file will be replaced by the written values. If you want to add more lines to a file without erasing the existing content, use a flag for appending.



Note: the channel must be open for writing, otherwise this method will throw a runtime error. Make sure that the method for opening the channel includes a, u or w flag as a parameter.

Errors handling

It is possible to handle errors occurring during read or write operations by means of WHENEVER ERROR statement, since the errors will change the value of the status built-in variable. For example:

```
WHENEVER ERROR CONTINUE
CALL ch1.Read([num,label])
IF STATUS THEN
  ERROR "A reading error occurred"
CALL ch1.close()
RETURN -1
END IF
WHENEVER ERROR STOP
```

Input/Output formatting

Input and output operations are mostly performed with formatted pieces of text, each line of which represents one record. A delimiter specified with the SetDelimiter() method indicates the end of each field value. The default delimiter is the pipe (|) symbol. The fields that do not have any values should also be bordered by delimiters. Here is an example of a formatted text file ready for reading:

```
1|John Smith|123445|London|
2|Samanta Black||Huston|
3||34567|New York|
|Tim Copper|67889|Dublin|
|Dolores Crew|||
6|Joseph Lee|9993338|Chikago|
```

Read() or Write() method will treat empty fields as NULL.

Generally, the Read() and Write() methods process the data according to the same formatting rules as the LOAD and UNLOAD SQL statements. The only difference is that here it is impossible to specify a NULL delimiter, because an empty DBDELIMITER environment variable is treated as the default pipe symbol.

The backslash symbol (\) can be used as an escape character, but only in case a delimiter is used. The Write() method will escape any backslash, delimiter and other special characters. The Read() method will convert any escaped character (\char combination) into char.

The next extract of the code will write one field value, containing a backslash, a delimiter and a line-feed character. In Lycia 4GL, backslash is treated as an escape character; therefore, it must be doubled to be added to a character string. The line-feed character in 4GL is "\n":

```
CALL chl.SetDelimiter("^")
CALL chl.Write("a\\bc^cd\\nEEE")
```

When the following code is executed, the text file will get the following string:

```
a\\bc\\^cd\\
EEE^
```

Using line terminators in Read() and Write() methods

The Read() and Write() methods transfer the escaped line-feed characters BS+LF. This is how they are written to the output. BS+LF are detected and transferred back when a reading operation is performed.

If you want to add a line-feed character to a value, the backslash and the line-feed should be given as two independent characters. When the value is written to the string constant, the backslash should be escaped.

In the following example, we use an empty delimiter in order to make the explanation simpler.

```
CALL chl.setDelimiter("")
CALL chl.write("aaa\\\\nbbb")  -- [aaa<bs><lf>bbb]
CALL chl.write("ccc\\nddd")    -- [aaa<lf>bbb]
```

The code above will result in the following output:

```
aaa\
bbb
ccc
ddd
```

The first and the second lines of this file will get to the same line in a Channel record.

The Read() method will interpret these lines into the following strings:

```
{Read 1} aaa<bs><lf>bbb
{Read 2} ccc
{Read 3} ddd
```

These lines are equal to the following assignments in string constants:

```
LET str = "aaa\\\\nbbb"
LET str = "ccc"
LET str = "ddd"
```

Using line terminators in ReadLine() and WriteLine() methods

The ReadLine() and WriteLine() methods treat the line-feed character as the end of the line. However, if a line-feed character is preceded by a backslash, the ReadLine() method will read it as a part of the line.

The `WriteLine()` method writes any linefeed character as is. The linefeed character escaped by a backslash is not read as a part of the line.

The example below is given to demonstrate these rules.

This code:

```
CALL ch1.WriteLine("aaa\\nbbb")  -- [aaa<bs><lf>bbb]
CALL ch1.WriteLine("ccc\\nddd")  -- [aaa<lf>bbb]
```

when executes, will result in following output:

```
aaa\  
bbb  
ccc  
ddd
```

If you use the `ReadLine()` method to read the output above, the method will get four lines, the first ending with a backslash:

```
aaa<bs>  
bbb  
ccc  
ddd
```

Line terminators on different platforms

On Windows platforms, the line terminators for DOS formatted text are CR/LF. The Channel class can be used to work with documents of this type.

On Windows and Unix platforms, the CR/LF line terminators are removed when a DOS file is read with the Channel class.

When writing on Windows, the CR/LF are used as line terminators. On UNIX, only LF is used.

To avoid the translation of CR/LF on Windows, add the *b* flag to the `OpenFile()` and `OpenPipe()` methods. The *b* flag is typically used together with *r* or *w* flags:

```
CALL ch1.OpenFile("myfile.txt", "wb")
```

On Windows, the *b* flag removes only LF from CR/LF terminator when reading from a file. CR will be added to the last field as a character value. Contrary, when the lines are written, the *b* flag removes CR and leaves LF.

On Unix, it does not matter whether the binary mode is used or not.

Input

arr_count()

The ARR_COUNT() function returns a positive whole number, typically representing the number of records entered in a program array during or after execution of the INPUT ARRAY statement.

Usage

You can use ARR_COUNT() to determine the number of program records that are currently stored in a program array. In typical 4GL applications, these records correspond to values from the active set of retrieved database rows from the most recent query. By first calling the SET_COUNT() function, 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 screen cursor 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). If SET_COUNT() was explicitly called, ARR_COUNT() returns the greater of these two values: the argument of SET_COUNT() or the highest value attained by the array index.

The sample code in the following example uses the value returned by ARR_COUNT() to count the size of the screen record. This can be found in the demonstration file "arr_count_function.4gl".

```
MAIN
  DEFINE
    x ARRAY[10] OF RECORD
    a CHAR(10),
    b CHAR(10)
    END RECORD,

    i INTEGER

  FOR i = 1 TO 5
    LET x[i].a = "Line: ", i
    LET x[i].b = "Data: ", i
  END FOR

  OPEN WINDOW w_test
    AT 2, 2
    WITH FORM "function_arr_count"

  CALL set_count(5)

  INPUT ARRAY x WITHOUT DEFAULTS FROM sc_rec.*
    ON KEY (F5)
```

```
        LET i = arr_count()
        ERROR "There are ", i USING "<&", " lines of data"

END INPUT

CALL fgl_winmessage("Exit","Press any key to close this demo application","info")
CLOSE WINDOW w_test

END MAIN
```

Form File: arr_count_function.per

```
DATABASE formonly

SCREEN
{
[f1          ][f2          ]
[f1          ][f2          ]
[f1          ][f2          ]
[f1          ][f2          ]
[f1          ][f2          ]
[f1          ][f2          ]
[f1          ][f2          ]
[f1          ][f2          ]
[f1          ][f2          ]
[f1          ][f2          ]

Press (F5) to check the array count.
Press (Escape) to exit
}

ATTRIBUTES
f1=formonly.a;
f2=formonly.b;

INSTRUCTIONS
SCREEN RECORD sc_rec [10] (
    formonly.a,
    formonly.b
)

DELIMITERS "[ ]"
```

The following example makes use of `ARR_COUNT()` and the related built-in functions `ARR_CURR()` and `SCR_LINE()` to assign values to variables within the `BEFORE ROW` clause of a `DISPLAY ARRAY` statement.

By calling these functions in BEFORE ROW, the respective variables are evaluated each time the cursor moves to a new line and are available within other clauses of the DISPLAY ARRAY statement.

```
MAIN
  DEFINE arr1 ARRAY[20] OF RECORD
    f1 CHAR(10),
    f2 CHAR(10),
    f3 CHAR(10)
  END RECORD
  DEFINE i, j INTEGER

  OPEN WINDOW w_test
    AT 2, 2
    WITH FORM "arr_curr_function"

  FOR i = 1 TO 20
    LET arr1[i].f1 = "Row ", i USING "<&"
    LET arr1[i].f2 = "Column 2"
    LET arr1[i].f3 = "Column 3"
  END FOR

  CALL set_count(20)

  DISPLAY ARRAY arr1 TO sc_rec.*
  BEFORE ROW
    LET i = arr_curr()
    LET j = scr_line()
    DISPLAY i TO f_arr_curr
    DISPLAY j TO f_scr_line
  END DISPLAY

  CLOSE WINDOW w_test
END MAIN
```

References

ARR_CURR(), SCR_LINE(), SET_COUNT()

arr_curr()

During or immediately after the INPUT ARRAY or DISPLAY ARRAY statement the ARR_CURR() function returns the number of the program record within the program array that is displayed in the current line of a screen array, or the last active screen array if there is no current array.

Usage

The current line of a screen array is the line that displays the screen cursor at the beginning of a BEFORE ROW or AFTER ROW clause.

The ARR_CURR() function returns an integer value. The first row of the program array and the first line (that is, top-most) of the screen array are both numbered 1. The built-in functions ARR_CURR() and SCR_LINE() can return different values if the program array is larger than the screen array.

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 following program segment tests the user input for duplication of what should be a unique column. If the field duplicates an existing item, the program instructs the user to try again.

```
MAIN
  DEFINE arr1 ARRAY[20] OF RECORD
    f1 CHAR(10),
    f2 CHAR(10),
    f3 CHAR(10)
  END RECORD
  DEFINE i, j INTEGER

  OPEN WINDOW w_test
    AT 2, 2
    WITH FORM "arr_curr_function"

  FOR i = 1 TO 20
    LET arr1[i].f1 = "Row ", i USING "<&"
    LET arr1[i].f2 = "Column 2"
    LET arr1[i].f3 = "Column 3"
  END FOR

  CALL set_count(20)

  DISPLAY ARRAY arr1 TO sc_rec.*
    BEFORE ROW
      LET i = arr_curr()
```

```
        LET j = scr_line()
        DISPLAY i TO f_arr_curr
        DISPLAY j TO f_scr_line
    END DISPLAY

    CLOSE WINDOW w_test
END MAIN
```

The ARR_CURR() function is frequently used with a DISPLAY ARRAY statement in popup windows to return the user's selection.

Form File: arr_curr_function.per

```
DATABASE formonly

SCREEN
{
    [f1      ] [f2      ] [f3      ]
    [f1      ] [f2      ] [f3      ]
    [f1      ] [f2      ] [f3      ]
    [f1      ] [f2      ] [f3      ]
    [f1      ] [f2      ] [f3      ]

    Program\g \gArray\g \gLine   [f16 ]
    Form\g \gArray\g \gLine      [f17 ]

}

ATTRIBUTES
f1=formonly.f1;
f2=formonly.f2;
f3=formonly.f3;
f16=formonly.f_arr_curr;
f17=formonly.f_scr_line;

INSTRUCTIONS
SCREEN RECORD sc_rec[5] (
    formonly.f1,
    formonly.f2,
    formonly.f3
)

DELIMITERS "[" "]"
```

References

ARR_COUNT(), SCR_LINE()

fgl_bell()

The `fgl_bell()` function requires no arguments to be passed to it and simply invokes a system bell in the client.

This system bell is a standard sound that can be used, for example, in conjunction with a warning dialog to inform the user of errors.

Each function call plays one short bell sound. For multiple instances, further function calls are required.

In case the `fgl_bell()` function is called for several times immediately one after another, only one bell sound is produced. To separate the sounds the `SLEEP` statement can be used.

Usage

Querix would advise using this function in place of the `'\b'` bell meta-character.

```
MAIN

    DISPLAY "Let's play a BELL sound" AT 5,5
    CALL fgl_bell()

    CALL fgl_winmessage("Exit","Press any key to close this demo application","info")
END MAIN
```

fgl_dialog_fieldorder()

The function `fgl_dialog_fieldorder()` is used to toggle between two different sets of events that occur when moving from one field to another.

The normal behaviour is that traversing from one field to another will apply some before and some after logic for all other fields between them. This behaviour can be altered where only the after logic of the current field and the before logic of the destination field is triggered.

Usage

The field order logic refers to the before/after field logic that applies when navigating from one control to another. In simplistic terms this logic dictates an application event (e.g., a message prompt) that occurs before and after a user traverses from one field to another.

Therefore, where five fields exist, e.g., f1-f5, and field f1 is active, clicking f5 will execute the *after* field logic for f1 and then the *before* and *after* logic for f2, f3, f4, and then the *before* logic for f5. If the function is called with the Boolean value of FALSE, the application will only execute the *after* field logic for f1 and the *before* logic for f5.

This function passes a single argument, TRUE or FALSE, expressed as a SMALLINT. The default behaviour for this function is TRUE which will apply all field order logic for all fields. To only apply the field order logic for the current and destination fields, pass a parameter of '0' with the function:

```
fgl_dialog_fieldorder(0)
```


fgl_dialog_setcurrline()

The fgl_dialog_setcurrline() function displays a specified row of the program array at a specified line of the screen array.

Usage

The fgl_dialog_setcurrline function takes two comma-separated arguments (*screen_line*, *prog line*), where *screen line* is the screen line number in the array, and *prog line* is the row number in the program array.

The code sample below creates a display using two ON KEY options. When you press the F4 key, the 4th row of the program record is displayed.

```
MAIN
  DEFINE x1 ARRAY[10] OF RECORD
    a CHAR(10),
    b CHAR(10),
    c CHAR(10)
  END RECORD

  DEFINE i SMALLINT

  OPEN WINDOW w_test
    AT 2, 2
    WITH FORM "fgl_dialog_setcurrline_function"

  FOR i = 1 TO 10
    LET x1[i].a = "Line", i USING "<&"
    LET x1[i].b = "AAAAAAAAAAAA"
    LET x1[i].c = "BBBBBBBBBBBB"
  END FOR

  CALL set_count(10)

  INPUT ARRAY x1 WITHOUT DEFAULTS FROM sc_rec.*
  ON KEY (F1)
    CALL fgl_dialog_setcurrline(1,1)
  ON KEY (F2)
    CALL fgl_dialog_setcurrline(2,2)
  ON KEY (F3)
    CALL fgl_dialog_setcurrline(3,3)
  ON KEY (F4)
    CALL fgl_dialog_setcurrline(4,4)
  ON KEY (F5)
    CALL fgl_dialog_setcurrline(5,5)
  ON KEY (F6)
    CALL fgl_dialog_setcurrline(6,6)
```

```
ON KEY (F7)
  CALL fgl_dialog_setcurrline(7,7)
END INPUT

CLOSE WINDOW w_test
END MAIN
```

Form File: fgl_dialog_setcurrline_function.per

DATABASE formonly

```
SCREEN
{
[f001  ] [f002      ] [f003        ]
[f001  ] [f002      ] [f003        ]
[f001  ] [f002      ] [f003        ]
[f001  ] [f002      ] [f003        ]
[f001  ] [f002      ] [f003        ]
[f001  ] [f002      ] [f003        ]
[f001  ] [f002      ] [f003        ]
}
```

```
ATTRIBUTES
f001=formonly.f1;
f002=formonly.f2;
f003=formonly.f3;
```

```
INSTRUCTIONS
SCREEN RECORD sc_rec[7] (
  formonly.f1,
  formonly.f2,
  formonly.f3
)
```

```
DELIMITERS "[ ]"
```

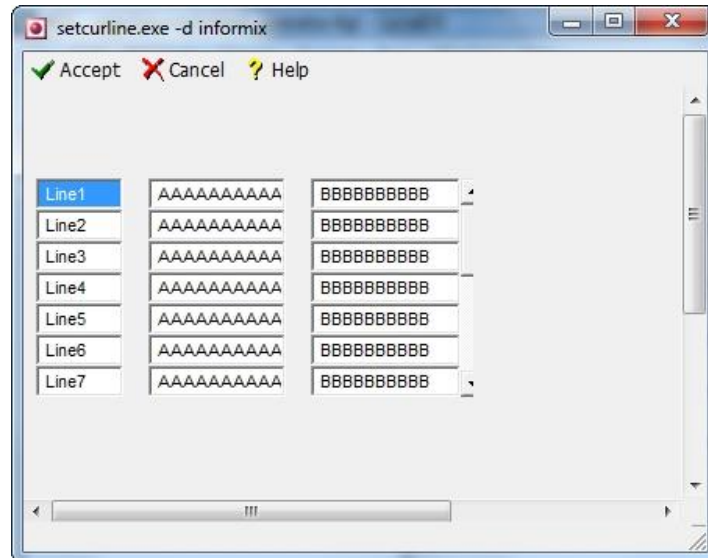


Figure 16 – Set Current Line Screenshot Example

Displaying a Row at a Given Line in a Screen Array

After executing the program, an array appears with three columns and a scroll bar, as shown in Figure 16 above,

If you press F4, the screen array is displayed with the fourth row being the current line in the display.

fgl_getkey()

The function FGL_GETKEY() waits for a key to be pressed and returns the integer code of the keystroke that the user makes. The function can take an optional argument, which is an INTEGER value for a timeout period, measured in seconds.

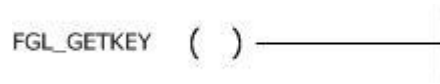


Figure 17 - fgl_getkey()

If the timeout is specified, then the function will return the INTEGER value of the keypress made within the timeout period. If no keypress is made within the timeout period, the function will return 431.

Usage

Unlike fgl_lastkey(), which can return a value indicating the logical effect of whatever key the user pressed, FGL_GETKEY() returns an integer representing the raw value of the keystroke that the user makes.

The FGL_GETKEY() function recognizes the same codes for keys that the fgl_keyval() function returns. Unlike fgl_keyval(), which can only return keystrokes that are entered in 4GL forms, FGL_GETKEY() can be invoked in any context where the user is providing keyboard entry.

Here is an example of a program fragment that calls both functions, so that fgl_keyval() evaluates what FGL_GETKEY() returns.

Sample Code: fgl_getkey_function.4gl

```
MAIN
  DEFINE key_press int
  LET key_press = 0

  DISPLAY "*** FGL_GETKEY() Example ***" at 1,5
  DISPLAY "FGL_GETKEY() waits for a key press and returns its key-code-number" AT
3,5
  DISPLAY "fgl_keyval() returns the value of defined key-constants" AT 4,5

  DISPLAY "Press any key to read its value or "
  || "press Escape to quit." AT 6,5

  WHILE key_press != fgl_keyval("escape")
    LET key_press = fgl_getkey()
    DISPLAY "You pressed " , key_press at 8,5
  END WHILE

END MAIN
```



In the code example shown above "key" refers to a key on the keyboard or the effect that keystroke has, rather than the SQL construct called KEY.

References

ASCII, fgl_keyval(), fgl_lastkey(), ORD()

fgl_key_queue()

The FGL_KEY_QUEUE() function pushes the keyvalue specified as its argument into a key queue.

If the key queue exists, then the key at the top of the queue is automatically returned on a call to fgl_getkey().

The argument for this function is an INTEGER which represents the key value.

Code Sample: fgl_key_queue_function.4gl

```
MAIN
  DEFINE my_key int

  #Push the key 'a' into the queue
  CALL fgl_key_queue(50)

  CALL fgl_getkey()

  #retrieve the value of the last key buffer
  LET my_key = fgl_lastkey()

  #Display the key value
  DISPLAY "You pressed the key: ", my_key at 5,5

  CALL fgl_winmessage("Exit","Press any key to close this demo application","info")
END MAIN
```

Reference

fgl_getkey(), get_keyval()

fgl_keyname()

The fgl_keyname() function has been provided as a reverse to the fgl_keyval() function in that it will provide the keyname corresponding to the key value provided.

Usage

This function takes an integer parameter representing a key value, and returns a VARCHAR(n) which is the key name associated with the given value.

For example:

```
DEFINE i INTEGER
DEFINE c CHAR(20)

LET i = fgl_keyval("Escape")
LET c = fgl_keyname(i)
```

Would return 'escape'.

fgl_keyval()

Function fgl_keyval() returns the integer code of a logical or physical key.

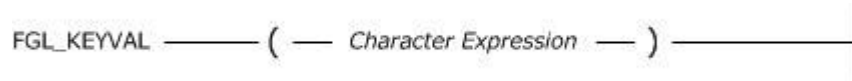


Figure 18 - fgl_keyval()

Usage

The fgl_keyval() function returns NULL unless its argument specifies one of the following physical or logical keys:

- A single letter or a digit
- A non-alphanumeric symbol from the keyboard (such as !, @, and #)
- Any of the keywords shown below, whether typed in upper or lowercase

ACCEPT	HELP	NEXT or	RETURN
DELETE	INSERT	NEXTPAGE	RIGHT
DOWN	INTERRUPT	PREVIOUS	TAB
ESC or ESCAPE	LEFT	PREVIOUSPAGE	UP
F1 to F64			
CONTROL-char (except A, D, H, I, J, M, L, R or X)			

The argument should always be placed between speech marks. If a single letter is specified,

fgl_keyval() becomes case-sensitive. When more than one character is used as the argument the case is not considered.

If the argument is invalid, fgl_keyval() returns NULL.

Code Sample: fgl_keyval1_function

This code sample presents a message console that displays the integer value of the Escape key.

```
MAIN

    DISPLAY "fgl_keyval(\"escape\") - Escape Key has the following value: ",
fgl_keyval("escape")

    sleep 10
END MAIN
```

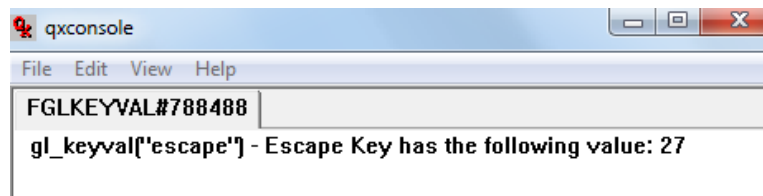


Figure 19 - fgl_keyval Example Output

Code Sample: fgl_keyval2_function

This code sample presents a message console that displays the integer value of any keyboard key that the user presses.

```
MAIN

    DEFINE key_press int
    LET key_press = 0

    DISPLAY "*** fgl_keyval() Example ***" at 1,5
    DISPLAY "FGL_GETKEY() waits for a key press and returns its key-code-number" AT
3,5
    DISPLAY "fgl_keyval() returns the value of defined key-constants" AT 4,5

    DISPLAY "Press any key to read its value or "
    || "press Escape to quit." AT 6,5

    WHILE key_press != fgl_keyval("escape")
        LET key_press = fgl_getkey()
        DISPLAY "You pressed " , key_press at 8,5
    END WHILE

END MAIN
```

The screen shot taken from the fgl_keyval2_function program shows the result of the user pressing the 'f' key on the keyboard.

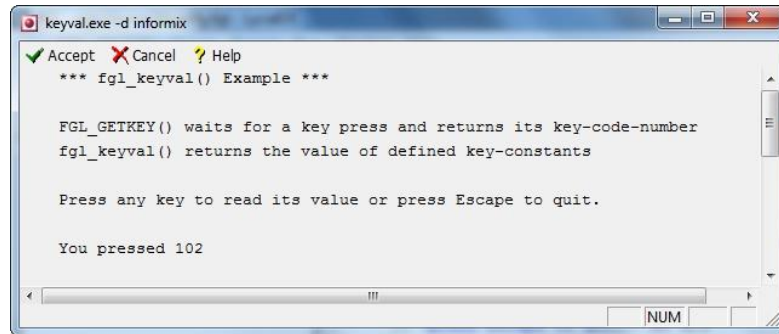


Figure 20 - fgl_keyval Return Value Example

Using fgl_keyval() with FGL_GETKEY() or fgl_lastkey()

fgl_keyval() can be used in form-related statements to examine a value returned by the FGL_GETKEY() or fgl_lastkey() function. By comparing the values returned by fgl_keyval() with what FGL_GETKEY() or fgl_lastkey() returns, you can determine whether the last key that the user pressed was a specified logical or physical key. Typically, you are most likely to use the fgl_keyval() function in conditional statements and Boolean comparisons:

```
MAIN
  DEFINE x CHAR
  DEFINE msg CHAR(100)

  DISPLAY "*** fgl_lastkey() Example 1 ***"   at 1,5

  PROMPT "Press any key" FOR CHAR x

  LET msg = "You pressed a key with value: ", fgl_lastkey()

  CALL fgl_message_box(msg)

END MAIN
```

To determine whether the user performed some 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.

References

ASCII, FGL_GETKEY(), fgl_lastkey(), ORD()

fgl_lastkey()

The fgl_lastkey() function returns an INTEGER code for the last key that was received by the runtime.

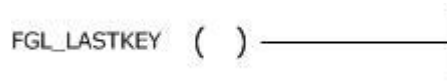


Figure 21 - fgl_lastkey Function

Usage

The code that the fgl_lastkey() function returns is the ASCII numeric code for the last keystroke before fgl_lastkey() was called. For example, if the last key that was pressed was the lowercase d, the fgl_lastkey() function returns 100. The value returned by fgl_lastkey() is not defined in a MENU statement.

Using fgl_lastkey() with fgl_keyval()

You do not need to know the specific key codes to use fgl_lastkey(). The built-in fgl_keyval() function can return a code to compare with the value returned by fgl_lastkey(). The fgl_keyval() function lets you compare the last key that the user pressed with a logical or physical key. For example, to check if the user pressed the Accept key, compare fgl_lastkey() with the fgl_keyval("accept") value.

Code Sample: fgl_lastkey1_function

This code sample produces a message box with a request for a key press.

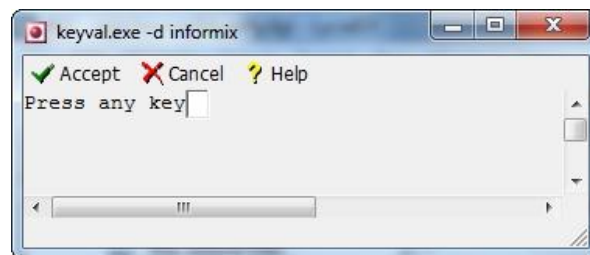


Figure 22 - fgl_lastkey Message Box

When you press a key, the program returns a dialog box with a text string giving the value of the key you just pressed.

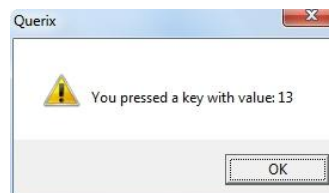


Figure 23 - fgl_lastkey Return Value Example

```
MAIN
  DEFINE x CHAR
  DEFINE msg CHAR(100)

  DISPLAY "*** fgl_lastkey() Example 1 ***"  at 1,5

  PROMPT "Press any key" FOR CHAR x

  LET msg = "You pressed a key with value: ", fgl_lastkey()

  CALL fgl_message_box(msg)

END MAIN
```

Code Sample: fgl_lastkey2_function

This code sample produces a message box with a request for a key press. When you press a key, the program returns a dialog box with a text string giving the value of the key you just pressed. The screen shot below shows the message box as it would be after you pressed the 'f' key.

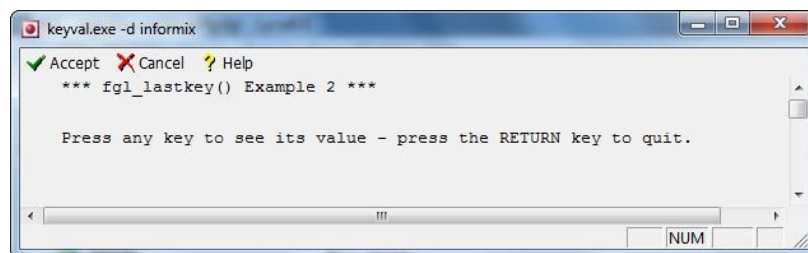


Figure 24 - fgl_lastkey Message Box Output

When you press the Return key, a dialog box is displayed, requesting an action to close the demonstration.

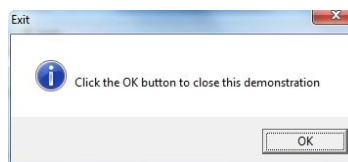


Figure 25 - fgl_lastkey Confirmation Box

```
GLOBALS
  DEFINE value CHAR
  DEFINE key INTEGER
END GLOBALS
```

```
MAIN
  DISPLAY "*** fgl_lastkey() Example 2 ***"  at 1,5
  DISPLAY "Press any key to see its value - press the RETURN key to quit."  at 3,5

  LET key = 0 -- fgl_lastkey()

  WHILE key != fgl_keyval("return")
    CALL my_getkey()
  END WHILE
  -- exit - -CALL quit()
  CALL fgl_winmessage("Exit","Click the OK button to close this
demonstration","info")
END MAIN

FUNCTION my_getkey()
  LET key = fgl_getkey()

  DISPLAY "Key pressed: ", key AT 5, 5
END FUNCTION
```

AUTONEXT Fields

If you type something into a field that has an AUTONEXT attribute, and then invoke the fgl_lastkey function, 4GL returns the code value of the last key that you pressed, regardless of any processing done in the AFTER FIELD or BEFORE FIELD clause. AUTONEXT is covered in more detail in the 4GL Statements volume.

References

ASCII, FGL_GETKEY(), fgl_keyval(), ORD()

scr_line()

The SCR_LINE() function returns a positive integer that corresponds to the number of the current screen record in its screen array during a DISPLAY ARRAY or INPUT ARRAY statement.

If this function is used to get the current line of a screen grid the number of the current screen record may be greater than the number of lines visible on the screen, because screen grid creates as many screen records as many records there are in the program array. In the case with a screen array the program records are moved up and down the visible screen records thus the number of the current program record and screen record may not coincide. If a screen greed each program record is assigned to an individual screen record and when you scroll the screen records are also moved.

Usage

The current screen record is the line of a screen array that contains the screen cursor at the beginning of a BEFORE ROW or AFTER ROW clause.

The first record of the program array and of the screen array are both numbered 1. The built-in 4GL functions SCR_LINE() and ARR_CURR() can return different values if the program array is larger than the screen array.

The following program dynamically updates the screen line to a form field as the user moves between rows of a DISPLAY ARRAY.

Code Sample: scr_line_function.4gl

```
MAIN
  DEFINE arr1 ARRAY[20] OF RECORD
    f1 CHAR(10),
    f2 CHAR(10),
    f3 CHAR(10)
  END RECORD
  DEFINE i, j INTEGER

  OPEN WINDOW w_test
    AT 2, 2
    WITH FORM "scr_line_function"

  FOR i = 1 TO 20
    LET arr1[i].f1 = "Row ", i USING "<&"
    LET arr1[i].f2 = "Column 2"
    LET arr1[i].f3 = "Column 3"
  END FOR

  CALL set_count(20)

  DISPLAY ARRAY arr1 TO sc_rec.*
  BEFORE ROW
    LET i = arr_curr()
```

```

        LET j = scr_line()
        DISPLAY i TO f_arr_curr
        DISPLAY j TO f_scr_line
    END DISPLAY

    CLOSE WINDOW w_test
END MAIN

```

This example also makes use of the related `ARR_CURR()` built-in function to assign values to variables within the `BEFORE ROW` clause of a `DISPLAY ARRAY` statement. Because these functions are invoked in the `BEFORE ROW` control block, the respective **i** and **j** variables are evaluated each time that the cursor moves to a new line and are available within other clauses of the `DISPLAY ARRAY` statement.

```

BEFORE ROW
    LET i = arr_curr()
    LET j = scr_line()
    DISPLAY i TO f_arr_curr
    DISPLAY j TO f_scr_line

```

Form File: `scr_line_function.per`

```

DATABASE formonly
SCREEN
{

    [f1      ] [f2      ] [f3      ]
    [f1      ] [f2      ] [f3      ]
    [f1      ] [f2      ] [f3      ]
    [f1      ] [f2      ] [f3      ]
    [f1      ] [f2      ] [f3      ]

    Program\g \gArray\g \gLine   [f16 ]
    Form\g \gArray\g \gLine      [f17 ]

}

ATTRIBUTES
f1=formonly.f1;
f2=formonly.f2;
f3=formonly.f3;
f16=formonly.f_arr_curr;
f17=formonly.f_scr_line;

INSTRUCTIONS
SCREEN RECORD sc_rec[5] (
    formonly.f1,
    formonly.f2,

```



```
formonly.f3  
)
```

```
DELIMITERS "[]"
```

References

ARR_COUNT(), ARR_CURR()

set_count()

The SET_COUNT() function takes an integer expression as its argument and specifies the number of records that contain data in a program array.

Usage

Before you use an INPUT ARRAY WITHOUT DEFAULTS statement or a DISPLAY ARRAY statement, you must call the SET_COUNT() function with an integer argument to specify the total number of records in the program array. Most often, these records contain the values in the retrieved rows that a SELECT statement returned from a database and are associated with a database cursor.

The SET_COUNT() built-in function sets an initial value from which the ARR_COUNT() function determines the total number of members in an array. If you do not explicitly call ARR_COUNT(), a default value of zero is assigned.

Sample Code: set_count_function.per

[Field sizes for 'f1' are reduced on the page to show layout.]

```
DATABASE formonly

SCREEN
{

\g-----\g
[f0 ][f1 ] Press\g \g(F1)\g \gfor\g \gless\g \grows
[f0 ][f1 ] Press\g \g(F2)\g \gfor\g \gmore\g \grows
[f0 ][f1 ]
[f0 ][f1 ]
[f0 ][f1 ]
[f0 ][f1 ]
[f0 ][f1 ]
[f0 ][f1 ]
[f0 ][f1 ]
[f0 ][f1 ]
[f0 ][f1 ]
[f0 ][f1 ]
[f0 ][f1 ]
[f0 ][f1 ]
[f0 ][f1 ]
[f0 ][f1 ]
[f0 ][f1 ]
[f0 ][f1 ] Press\g \g(F9)\g \gto\g \gexit

}

ATTRIBUTES
f0=formonly.idx;
f1=formonly.data;
```

```
INSTRUCTIONS
SCREEN RECORD sc_rec [15]  (
    formonly.idx,
    formonly.data
)

DELIMITERS "[" "]"
```

Sample Code: set_count_function.4gl

```
MAIN
    DEFINE arr ARRAY[15] OF RECORD
        idx INTEGER,
        data CHAR(20)
    END RECORD
    DEFINE i INTEGER

    OPEN WINDOW w_test
        AT 2, 2
        WITH FORM "function_set_count"
        ATTRIBUTE(BORDER)

    FOR i = 1 TO 15
        LET arr[i].idx = i
        LET arr[i].data = "Array line ", i
    END FOR

    LET i = 5

    WHILE TRUE
        CALL set_count(i)

        DISPLAY ARRAY arr TO sc_rec.*
        ON KEY (F1)
            LET i = i - 1
            EXIT DISPLAY

        ON KEY (F2)
            LET i = i + 1
            EXIT DISPLAY

        ON KEY (F9)
            EXIT WHILE
    END DISPLAY

    IF i > 15 THEN
        LET i = 15
```

```
END IF

IF i < 1 THEN
  LET i = 1
END IF
END WHILE
END MAIN
```

In the following program fragment, the variable **i** is an array index that received its value in an earlier FOREACH loop. The index was initialized with a value of 1, so the expression (n_rows -1) represents the number of rows that were fetched from a database table in the FOREACH loop. The expression SET_COUNT (i - 1) tells DISPLAY ARRAY how many program records containing row values from the database are in the program array, so it can determine how to control the screen array.

```
FOREACH c_contact INTO cont_arr[i]
  LET i = i + 1
END FOREACH
CALL SET_COUNT(i - 1)
DISPLAY ARRAY cont_arr
  TO sc_cont.*
```

If no INPUT ARRAY statement has been executed, and you do not call the SET_COUNT () function, the DISPLAY ARRAY or INPUT ARRAY WITHOUT DEFAULTS statement displays no records.

References

ARR_COUNT(), ARR_CURR()

DIALOG Methods

This section covers the methods used during multiple input and display dialogs combined in the DIALOG statements. They cannot be used outside the user interaction statements and reference a predefined DIALOG class object.

The DIALOG class is a class which belongs to ui namespace. The class is designed to provide additional control to the following user interaction statements, used within a DIALOG ... END DIALOG block:

- INPUT
- INPUT ARRAY
- DISPLAY ARRAY
- CONSTRUCT

The Dialog class has a pre-defined object variable which is referenced by the DIALOG keyword. When you want to reference the DIALOG object, use it with a corresponding method within a control block of a user interaction statement:

```
INPUT ARRAY myarr FROM scrarr.*  
  ON ACTION ("action")  
    CALL DIALOG.setFieldActive("custid",0)  
END INPUT
```

A DIALOG keyword used outside an interaction statement causes a compile-time error, because the reference to the pre-defined Dialog object can be performed only during some dialog operation. However, the object to a function in order to make a dialog manipulation code, common for several user interaction operations.

In this case, you'll have to declare an instance for the Dialog object using the DEFINE statement:

```
INPUT ARRAY myarr FROM scrarr.*  
  BEFORE INPUT  
    CALL dialog_setup(DIALOG)  
END INPUT  
  
FUNCTION dialog_setup(dlg)  
  DEFINE dlg ui.Dialog  
  IF user.group = "admin" THEN  
    CALL dlg.setActionActive("delete",1)  
    CALL dlg.setActionActive("modify",1)  
    CALL dlg.setActionActive("link",1)  
  ELSE  
    CALL dlg.setActionActive("delete",0)  
    CALL dlg.setActionActive("modify",0)
```

```
CALL dlg.setActionActive("link",0)
END IF
END FUNCTION
```

Thus you can use either a predefined DIALOG object within the interactive statements, or define a custom ui.Dialog object.

General Methods

ui.Dialog.GetCurrent()

The ui.Dialog.GetCurrent() method is used to bind the Dialog object with the currently active dialog. The method returns the current dialog object. If no dialog is active at the moment of the method invocation, it will return NULL. It is useful, if you use a custom dialog object and not the default one.

```
DEFINE dial ui.Dialog
LET dial = ui.Dialog.GetCurrent()
```

accept()

The Accept() method is used to accept the value inputted to the current field and terminate the current dialog. The method is equal to Accept key press. The method can prove useful when you want to terminate input from a function, where it is impossible to manipulate the calling routine dialog.

getArrayLength()

The GetArrayLength() method is used to return an integer number representing the total number of rows that compose the DISPLAY ARRAY or INPUT ARRAY lists. The method needs the screen array name as an argument:

```
DIALOG
INPUT ARRAY custlist FROM sa_custlist.*
BEFORE ROW
MESSAGE "Row count: " || DIALOG.getArrayLength("sa_custlist")
...
END INPUT
```

nextField()

The NextField() method is used to specify the name of the field to which the cursor must go when the program control returns to the dialog. The method performance is similar to that of the NEXT FIELD statement, but it does not break the program workflow.

The NextField() method needs an argument which specifies the target field name. Therefore, the target field can be changed at runtime depending on the situation (key press, values, privileges etc.)

```
DEFINE str STRING,
```

```
        field1, field2 STRING
    ...
    ON ACTION ("switch")
        IF str = check_val THEN
            CALL DIALOG.nextField(field1)
            CONTINUE DIALOG
        ELSE
            CALL DIALOG.nextField(field2)
            CONTINUE DIALOG
        END IF
    CALL save_changes()
```

getCurrentRow()

The GetCurrentRow() method retrieves the current row of the DISPLAY ARRAY or INPUT ARRAY list. The method needs the name of the screen array as an argument. The method has the same effect as the scr_line() built-in function. Below is given an example of the method usage.

```
DIALOG
DISPLAY ARRAY orders TO ordes_scrarr.*
BEFORE ROW
    MESSAGE "You're in row #: " || DIALOG.getCurrentRow("orders_scrarr")
    ...
END DISPLAY
INPUT ARRAY clients TO client_scrarr.*
BEFORE ROW
    MESSAGE "You're in row #" || DIALOG.getCurrentRow("client_scrarr")
    ...
```

setCurrentRow()

The SetCurrentRow() method is used to move the cursor to the new row during INPUT ARRAY or DISPLAY ARRAY statements execution. This method effect is identical to that of the fgl_dialog_setcurrline() function. The method needs two arguments to be passed:

```
CALL DIALOG.SetCurrentRow(screen_array, row_num)
```

The screen_array argument is the name of the screen array. The row_num is the index of the row in a program array where the cursor must be placed.

Note, that control blocks (like AFTER ROW) are ignored when you use the SetCurrentRow() method to move the cursor. In case when multi-selection is enabled, all the selected rows become inactive and the newly specified row is selected.

getCurrentItem()

The GetCurrentItem() method is used to retrieve the name of the currently active form item and accepts no parameters. It can return the following values depending on the focus location:

- the name of the corresponding action, if the program focus is on a button or another item having the ACTION attribute;
- The name of the current field represented in [tab_name.]field_name format. The tab_name is added to the retrieved value if a FROM clause contains an explicit list of fields. If the dialog uses FROM screen_record.* or BY NAME structures, the prefix is omitted;
- The screen_array name, if the program focus is in an active DISPLAY ARRAY list;
- Screen_array.screen_record identifier if the program focus is in INPUT ARRAY list. In some cases the field name can stay unidentified (e.g., in the BEFORE INPUT block of an INPUT ARRAY statement)

getFieldBuffer()

The GetFieldBuffer() method is used to return the input buffer of the field specified as the method argument. The field specification can optionally include a prefix:

```
LET cur_buff = DIALOG.GetFieldBuffer("mytable.mycolumn")
```

The input buffer is usually specified by fgl_dialog_setbuffer() function and DISPLAY TO/DISPLAY BY NAME statements.

getFieldTouched()

The GetFieldTouched() method is used to check whether the value in the specified field or set of fields was changed during the input. The method returns TRUE if at least one of the specified fields is marked as touched and FALSE - if all the fields of the fields remain untouched.

Note, that the value in the field is considered to be changed even if the changes were made and then undone (for example, a letter inputted and then erased). The result of this method is the same as the result of the field_touched() built-in function. However, whereas the function does not allow variables as the field specification, the method does, which gives it the necessary flexibility.

```
DIALOG.GetFieldTouched("field_list")
```

The field_list argument can be represented in several ways:

- as a field name with an optional prefix

```
DIALOG.GetFieldTouched("mytable.field1")  
DIALOG.GetFieldTouched("screenrecord.field2")
```

- as a screen_record identifier with an asterisk, which means that the method will check all the fields included to the specified screen record.

```
DIALOG.GetFieldTouched("my_screc.*")
```

- as an asterisk, identifying that the method will check all the fields used in the current dialog:


```
DIALOG.GetFieldTouched("*")
```

setFieldTouched()

The SetFieldTouched() method is used to set the value for the touched flag of the specified fields. The method needs two arguments: the fields list and the flag value:

```
DIALOG.SetFieldTouched("field_list", value)
```

The rules of the fields reference are the same as with the GetFieldTouched() method.

The value can be set as TRUE - to set the fields as touched and FALSE to set them as untouched:

```
DIALOG.SetFieldTouched("scrrec.*", TRUE)
```

getForm()

The GetForm() method returns a ui.Form object which will be used to manipulate the form currently used in the dialog. The resulting form object can be used to manipulate current form elements.

setActionActive()

The SetActionActive() method is used to enable/disable actions specified in ACTION or COMMAND clauses. The method has the following reference syntax:

```
DIALOG.SetActionActive(action, boolean)
```

The action argument is a string (lower case is a must) which defines the action influenced by the method. The action can be identified fully or partially. If the last is the case, the missing information is automatically filled depending on the context:

- action_name
- dialog_name.action_name
- dialog_name.field_name.action_name
- field_name.action_name (for singular dialogs only)

The dialog_name is a string containing the name of a singular dialog or a sub-dialog. The field_name is a string identifying the field used in an INFIELD clause of an ON ACTION block.

If the method does not find the action specified in the action argument a run-time error will occur. If the SetActionActive() method is used within a sub-dialog, the identifier of this sub-dialog can be omitted.

In a singular interaction statements like INPUT, the field-specific actions can be referenced by field_name.action_name structure if the dialog definition did not contain the NAME attribute.

The Boolean attribute specifies whether the action can be invoked. The attribute should be represented as 1 (TRUE, the action is enabled) or 0 (FALSE, the action is disabled). In GUI mode, the form elements, such as buttons, become inactive when the actions bound to them are disabled.

You can use the `SetActionActive()` method in an `INFIELD` block in order to disable or enable an action when a cursor goes to a specified field. In order to simplify the actions management, you can create a separate function which analyses the context and activates/deactivates the actions available on the current stage.

setActionHidden()

The `SetActionHidden()` method is used to modify the Default View of an action so that it can be visible or hidden. The method invocation needs two arguments:

```
DIALOG.SetActionHidden("action", boolean)
```

The action argument is a string (lower case is a must) which defines the action influenced by the method. The action can be identified fully or partially. If the last is the case, the missing information is automatically filled depending on the context:

- action_name
- dialog_name.action_name
- dialog_name.field_name.action_name
- field_name.action_name (for singular dialogs only)

The `dialog_name` is a string containing the name of a singular dialog or a sub-dialog. The `field_name` is a string identifying the field used in an `INFIELD` clause of an `ON ACTION` block. If the `SetActionActive()` method is used within a sub-dialog, the identifier of this sub-dialog can be omitted.

If the method does not find the action specified in the action argument a run-time error will occur.

In a singular interaction statements like `INPUT`, the field-specific actions can be referenced by `field_name.action_name` structure if the dialog definition did not contain the `NAME` attribute.

The Boolean attribute specifies whether the action is hidden. The attribute should be represented as 1 (TRUE, the action is hidden) or 0 (FALSE, the action is visible).

Field manipulation methods

There is a number of field manipulation methods supported by the Dialog class. These methods need the field reference as arguments. The main variants of syntax of the fields reference are given in the list below:

- field_name

- table_name.field_name
- screen_record_name.field_name
- FORMONLY.field_name

If the prefix is omitted, the program will take the first form field matching. A runtime error -1373 occurs when the specified field cannot be found.

Note, that the prefix given in the field name specification should match the one given by the dialog according to the field binding operation performed at the beginning of an interactive statement.

If the screen record was specified implicitly, for example in an INPUT BY NAME statement, the field prefix should be represented by name of any valid screen-record in which the field is engaged or by the FORMONLY keyword or a database table name used in the form file.

If a screen record is specified explicitly in the dialog FROM clause (e.g., INPUT .. FROM screen_record.*), you should specify the prefix matching the name of the screen record given in the FROM clause.

Some of the methods, e.g., Validate(), SetFieldTouched(), GetFieldTouched(), SetFieldActive() can take several fields as an argument ("customer_rec.*")

setFieldActive()


The SetFieldActive() method is used to enable/disable form fields used in the current dialog. The disabled field remains visible, but the user cannot edit the value in it.

The syntax of the method invocation is:

```
DIALOG.SetFieldActive("field_list", boolean)
```

The field_list argument is represented by a string which contain the field qualifier and, optionally, a prefix (e.g., "[table_name.]column"). The field list can also be represented as a table prefix followed by an asterisk, which means that all the columns present in the table will be taken into account.

The Boolean argument is represented by 1(TRUE) or 0(FALSE) and indicates whether the specified fields are active or not, respectively.

	Note: If you disable all the fields in the present dialog, the dialog execution will be skipped altogether.
---	--

insertRow()

The InsertRow() method is used to insert a row to the given position of the list. The method execution does not influence the cursor position nor initiates AFTER ROW/BEFORE ROW AFTER INSERT/BEFORE INSERT control blocks. The method performance is similar to inserting a new row to the program array.

The difference is that the internal registers are updated automatically. If multi-row selection is enabled in the current dialog, the selected rows will stay selected after the method execution.

The syntax of the method reference is:

```
DIALOG.InsertRow("screen_array", index)
```

The screen_array argument is the name of the screen array in the current dialog. The index argument is the index of the newly created row.

The InsertRow() method is used only to insert a new row to the current screen array list and does not fill it with any values. To focus the new row, use the SetCurrentRow() method.

The InsertRow() method must not be used in BEFORE ROW/AFTER ROW, BEFORE INSERT/AFTER INSERT, BEFORE DELETE/AFTER DELETE blocks. However, you can use it in an ON ACTION block.

If you want to take control of the treeview, use InsertNode() method instead of the InsertRow().

insertNode()

The InsertNode() method is similar in its performance to the InsertRow() method, but it should be used in a dialog controlling a treeview. The syntax of the method reference is:

```
DIALOG.InsertNode("screen_array", index)
```

Here, index is an integer value that specifies the sibling node in the program array before which the current one is to be inserted.

When the InsertNode() method passes the index to the next sibling node. In program array, the parent-id member of the new node gets the parent-id of the next sibling node. After that the internal tree structure is automatically rebuilt.

appendRow()

The AppendRow() method is used to append a row to the end of the current list. The method, when executed, does not initiate any BEFORE ROW/AFTER ROW or BEFORE INSERT/AFTER INSERT actions or set a new row selection. The method performance is similar to appending a new row to the program array. The difference is that the internal registers are updated automatically. If multi-row selection is enabled in the current dialog, the selected rows will stay selected after the method execution. The new row appears unselected at the end of the list. The AppendRow() method does not move cursor to the new row, nor the focus is switched to the list.

The AppendRow() method must not be used in BEFORE ROW/AFTER ROW, BEFORE INSERT/AFTER INSERT, BEFORE DELETE/AFTER DELETE blocks. However, you can use it in an ON ACTION block.

Note, that the AppendRow() method is used only to add a new row to the current screen array list and does not fill it with any values.

The method needs one argument:

```
DIALOG.AppendRow("screen_array")
```

The screen_array argument specifies the name of the screen array identifying the list to which the new row is to be appended.

The following snippet of the source code adds a number of new rows:

```
ON ACTION add_rows
LET num = fgl_winprompt (5, 5, "Enter the number of rows to be added", "1", 5, 2)
FOR i = 1 TO num
    CALL DIALOG.AppendRow("scr_arr")
END FOR
```

appendNode()

The AppendNode() method is used to add a new node within a specified parent. One should use the method to modify a tree table widget array during a dialog execution. It is possible to fill the program array from BEFORE DISPLAY and BEFORE DIALOG control blocks before the dialog execution starts.

The method needs two arguments:

```
CALL DIALOG.AppendNode("screen_array", index)
```

The screen_array argument passes the name of a screen array which identifies the tree. The index attribute passes the index of the parent node.

When new rows are added to a tree table widget, the new node and parent node id is used for creating the structure of the internal tree. The AppendNode() method passes the index of the parent node, and the new node will be appended under it.

In a program array, the parent-id member which belongs to the new node is initialized automatically and gets the id value of the parent node identifier using the index argument. After that, the whole tree structure is re-built.

If zero is passed as a parent index, the method will create a new root node and append it to the tree.

In the program array, the parent node id member specified by the index will be automatically set to the parent-id member of the new node.

deleteRow()

The DeleteRow() method is used to delete a row from the specified screen array list. The row selection is not influenced by the method. The method performance is similar to deleting rows from the program array. The difference is that the internal registers are updated automatically.

The method execution does not initialize the actions specified in BEFORE ROW/AFTER ROW or BEFORE DELETE/AFTER DELETE control blocks, except for some cases. The BEFORE ROW/BEFORE FIELD control blocks are executed if the DeleteRow() method deletes the currently selected row during execution of an INPUT ARRAY or DISPLAY ARRAY statements which have the focus.

One shouldn't include the DeleteRow() method to BEFORE ROW, AFTER ROW, BEFORE DELETE, AFTER DELETE, BEFORE INSERT, AFTER INSEET control blocks. However, the method is allowed in an ON ACTION block.

The method reference syntax is:

```
CALL DIALOG.DeleteRow("screen_array", index)
```

- The screen_array argument passes the name of the screen array which identifies the list.
- The index attribute is used to specify the index of the row to be deleted. The method performs no actions if the index attribute is specified as zero.

If the DeleteRow() method deletes the last row in an INPUT ARRAY list, a temporary row is automatically appended so that the user can enter new data. No temporary row is added if there is AUTO APPEND = FALSE attribute. In this case, the current row register will be changed in order to prevent it from being greater than the total number of rows.

deleteNode()

The DeleteNode() method is used to delete nodes when the dialog works with a tree table. You can fill the program array before the dialog execution using BEFORE DISPLAY and BEFORE DIALOG control blocks.

The method needs two arguments:

```
CALL DIALOG.DeleteNode("screen_array", index)
```

- The screen_array argument passes the name of the screen array identifying the tree.
- The index argument specifies the index of the node that should be deleted from the program array.

Unlike the DeleteRow() method, the DeleteNode() method deletes all child nodes and only after that removes the node identified by the index argument.

deleteAllRows()

The DeleteAllRows() method is used to remove all the rows of a DISPLAY ARRAY or INPUT ARRAY list.

One shouldn't include the DeleteAllRows() method to BEFORE ROW, AFTER ROW, BEFORE DELETE, AFTER DELETE, BEFORE INSERT, AFTER INSEET control blocks. However, the method is allowed in an ON ACTION block.

The method needs only one argument which specifies the name of the screen array the list of which should be deleted.

During the INPUT ARRAY or DISPLAY ARRAY statement execution, the dialog automatically appends a temporary row to the list after the DeleteAllRows() deletes all the rows. The temporary row won't be

added if the AUTO APPEND = FALSE attribute is used. In this case, the current row register will be changed in order to prevent it from being greater than the total number of rows.

The BEFORE ROW control block is executed if the DeleteAllRows() method deletes the currently selected row during execution of an INPUT ARRAY or DISPLAY ARRAY statements which have the focus.

validate()

The Validate() method is used to check the values inputted to the form fields. The syntax of the method reference is:

```
DIALOG.Validate("field_list")
```

The method checks the values according to the parameters specified in INCLUDE, NOT NULL and REQUIRED field attributes. The method returns 0 if the values satisfy all the requirements, otherwise, it returns an error code of the first field which does not.

Note, that to avoid inputting wrong data, the current field is always checked, even if it is not mentioned in the Validate() method fields list.

If an error occurs during the Validate() method execution, the program performance does not stop. Instead, a corresponding error message occurs and the program finds the next field to where the cursor will move after the program control returns to the interactive process.

setDefaultUnbuffered()

The SetDefaultUnbuffered() method is used to change the value of the UNBUFFERED attribute. The method gets two arguments - TRUE and FALSE:

```
CALL ui.Dialog.SetDefaultUnbuffered(TRUE)
```

setArrayAttributes()

The SetArrayAttributes() method is used to set display attributes for specific cells of the screen array. The method needs two arguments to be specified :

```
CALL Dialog.SetArrayAttributes("screen_array", program_array)
```

The screen and program array should have the same number of records. The elements of the screen and program array must have the same names. Moreover, the screen array elements should be defined as a character data type, which is typically STRING.

The program_array attribute specifies the program array containing the screen array colour and video attributes. The cell attributes can be represented as combinations of following:

- REVERSE attribute.
- UNDERLINE attribute.
- BLINK attribute.

- colours (only one can be specified at once): BLACK, MAGENTA, BLUE, CYAN, GREEN, YELLOW, RED, WHITE.

When the attribute values are stored to the program array, they can be passed to the dialog with the `SetArrayAttributes()` method used in a BEFORE DISPLAY or BEFORE INPUT blocks.

The attributes set this way can be changed dynamically. To cancel the applied attributes, pass NULL to the `SetArrayAttributes()` as the `program_array` argument.

setCellAttributes()

The `SetCellAttributes()` method is used to specify display attributes for singular dialogs with one screen array. The method needs only one attribute:

```
CALL Dialog.SetCellAttributes(program_array)
```

The `program_array` attribute is a program array containing the list of attributes.

setSelectionMode()

The `SetSelectionMode()` method is used to enable or disable multi-row selection. The method needs two arguments:

```
CALL Dialog.SetSelectionMode ("screen_array", mode)
```

The `screen_array` argument specifies the screen array to which the method will be applied.

The `mode` argument takes an integer value specifying the selection mode. 0 stands for single row selection, 1 stands for multi-row selection. To select a continuous set of rows, select the first row of the set, hold SHIFT and select the last row of the set.

To select separate rows, hold CTRL while selecting each new row.

You can switch between the selection modes dynamically throughout the dialog execution. When the multi-row selection mode is off, the currently selected rows, if any, are marked as unselected.

isRowSelected()

The `IsRowSelected()` method is used in a multi-row selection mode to check whether a row is selected. The method returns a Boolean value, which is TRUE if the specified row is selected, and FALSE - if not. The method is typically used in a conditional statement and needs two arguments:

```
IF Dialog.IsRowSelected ("screen_array", index) THEN...
```

The `index` attribute specifies the index of the row to be checked.

setSelectionRange()

The SetSelectionRange() method is used to select or unselect a range of rows in a multi-row selection mode.

The method needs four arguments:

```
Dialog.SetSelectionRange ("screen_array", start_index, end_index, selection)
```

The start_index and the end_index arguments specify the first and the last row of the range to be selected. The indices values should be within the possible range of the array indices.

-1 specified as the end_index value indicates that the rows should be selected from the start_index to the end of the list.

The selection argument specifies whether the defined range is selected or deselected. Value 1 means that the rows will be selected, 0 will make them deselected.

selectionToString()

The SelectionToString() method is used to pass the values of the selected rows into a tab-separated list. In a single-row selection mode, the method returns the row which is currently selected.

The method needs one argument which specifies the name of the screen array from which the selected rows will be taken.

The method retrieves the values according to the following rules:

- The order of the rows is defined by the user-specified settings. If the user did not specify any settings, the default order is used.
- If a column was moved, its values will be placed to the same position as in the table.
- If a column is hidden, its values will be ignored.
- Phantom columns are ignored.

If the passed value contains special characters (double-quotes, controls characters having ASCII less than 0x20, new-line characters), it will be taken into double quotes.

Dynamic and Static Arrays Methods

This section covers the methods which apply DYNAMIC ARRAY variables. These methods serve for utilitarian purpose and are used for resigning the array, adding and removing its elements.

Most of the methods described below are applicable only to one-dimensional dynamic arrays, but some of them can also be used with multi-dimensional ones.

Querix 4GL dynamic arrays are resized automatically when elements are assigned values and their size corresponds to the largest element index.

Some of the methods have two variations, differing in method name and/or the set of the used parameters.

The general usage of methods is described [above](#).

append()/appendElement()

The Append() and AppendElement() methods add a new array element after the last array element. The size of the array will increase by 1. The Append() method accepts one parameter which is the value to be inserted into the added array element:

```
array_variable.append(value)
```

The AppendElement() method does not need any parameters and adds a new element with the NULL value. The syntax of the method is as follows:

```
CALL ar.AppendElement()
```

The methods are applicable only to dynamic arrays and do not have any effect on the static ones.

Usage

```
DEFINE ar DYNAMIC ARRAY OF CHAR(15)
...
FOR i = 1 TO 5                                #the array has 5 elements
    LET ar[i]="element "||i
END FOR

CALL ar.append("element 6")                    #adds 6th element to the array
CALL ar.appendelement()                        #adds 7th element with the NULL value
DISPLAY ar.GetLength()                        #displays "7"
```

clear()

The clear() method is used to set all the elements to NULL (in a static array) or remove all the elements (in a dynamic array). This method accepts no parameters.

The syntax of the method is as follows:

```
CALL ar.Clear()
```

or

```
array_variable.clear()
```

It is not identical to the usage of the INITIALIZE TO NULL statement, because the statement removes only the values leaving the array elements intact.

The clear() method can also be used with multi-dimensional dynamic arrays.

Usage

```
DEFINE ar DYNAMIC ARRAY OF CHAR(15)
...
FOR i = 1 TO 5                                #the array has 5 elements
    LET ar[i]="element "||i
END FOR

CALL ar.clear()                                #the array has no elements
```

delete()/deleteElement()

The delete() method removes all the specified array elements within the given range or at the given position. This method accepts from 1 to 2 parameters:

```
array_variable.delete(first [,last])
```

Both arguments are integer expressions which indicate the index of the array elements. If only one argument is given, the array element with the index corresponding to this argument will be deleted.

The deleteElement() method removes only one array element and needs the index of the element as the argument:

```
array_variable.deleteElement(element_index)
```

When elements are deleted from a dynamic array, the array size is reduced by the number of the deleted elements.

When elements are deleted from a static array, the remaining elements that go below the deleted ones are moved upwards, and the emptied elements get the NULL value.

Usage

```
DEFINE ar DYNAMIC ARRAY OF CHAR(15)
...
FOR i = 1 TO 100                #the array has 100 elements
    LET ar[i]="element "||i
END FOR

CALL ar.delete(50)              #the 50th element is deleted and the array size is
                                #99
CALL ar.delete(10,30)           #20 elements are deleted and the array size is 79
```

getSize()/getLength()

The `getSize()` and `getLength()` methods return the size or the length of a single-dimensional array. It accepts no parameters:

```
array_variable.getSize()  
array_variable.getLength()
```

Usage

```
DEFINE ar DYNAMIC ARRAY OF CHAR(15)  
...  
    LET ar[3000]="element "||3000  
  
CALL ar.getSize()           #returns 3000
```

insert()/insertElement()

The insert() method inserts one array element into the specified position. It accepts one or two arguments:

```
array_variable.insert(index , [value])
```

The index argument is an integer expression. It specifies the position where the element is to be added. The value parameter specifies the value to be stored in this element, it should be compatible with the data type of the array. If only one parameter is provided, it is treated as the index and an element with the default value corresponding to the array data type is added at the specified position.

The InsertElement() method inserts one empty element to a specified position. The method needs an argument specifying the position to which the element is to be inserted.

```
array_variable.insertElement(index)
```

When an element is inserted, all the subsequent array elements are moved downwards. The dynamic array size increases by 1, and a static array loses its last element.

Usage

```
DEFINE ar DYNAMIC ARRAY OF INT
...
FOR i = 1 TO 5                                #the array has 5 elements
    LET ar[i]=i
END FOR

CALL ar.insert(2)                             #inserts 0 as the second element, now array has 6
                                              #elements
CALL ar.insert(3, 1000)                       #inserts 1000 as the third element, the array now
                                              #has 7 elements
```

resize()

The `resize()` method resizes the given single-dimensional array to the specified size. If the new size is bigger than the previous array size, the elements will be appended to the end of the array and will contain default values. If the new size is smaller, the excessive elements will be deleted from the end of the array and all the data in them will be lost.

This method accepts a single parameter which is an expression returning a positive integer.

```
array_variable.resize(size)
```

Usage

```
DEFINE ar DYNAMIC ARRAY OF INT
...
FOR i = 1 TO 10           #the array has 10 elements
    LET ar[i]=i
END FOR

CALL ar.resize(100)      #the array is resized to 100 elements, 90 of which contain 0
CALL ar.resize(5)        #the array is then resized to 5 elements, so values 6, 7, 8, 9,
                          #and 10 are discarded together with 0 values
```

fgl_dialog_getbufferstart()

The `fgl_getbufferstart()` function is used to return the offset of the rows that comprise the page to be filled by `DISPLAY ARRAY` statement.

The syntax of the function invocation is:

```
CALL fgl_dialog_getbufferstart()
```

The function returns an integer number and is used within the `ON FILL BUFFER` clause.

fgl_dialog_getbufferlength()

The `fgl_dialog_getbufferlength()` function is used within the `ON FILLED BUFFER` clause of the `DISPLAY ARRAY` statement to retrieve the number of rows needed to fill a page of the paged display array.

The syntax of the function invocation is as follows:

```
CALL fgl_dialog_getbufferlength()
```

The function returns an integer value.

GUI Client

fgl_fglgui()

The `fgl_fglgui()` function identifies whether the application is to display in graphical or character mode by returning the current value of the GUI environment variable. This function exists for the purposes of compatibility.

Usage

The function has no parameters.

```
MAIN

    IF fgl_fglgui() THEN

        CALL fgl_winmessage("fgl_fglgui()", "You are running a graphical client",
"info")
    ELSE
        CALL fgl_winmessage("fgl_fglgui()", "You are running a text mode client",
"info")
    END IF

END MAIN
```


fgl_getproperty()

This function takes three arguments and is used as below:

```
fgl_getproperty(["gui"], type.name, value)
```

In the above syntax, "gui" refers to the subsystem that the property belongs to. This optional parameter defaults to "gui", so it can be omitted. The type is the element of the subsystem that the property affects; the name is the actual name of the property.

GUI properties are script-like properties that can be used to determine the overall look and functionality of a 4GL program. Unlike script options, the GUI properties are dynamic and so can be used to alter a program's appearance during its execution, allowing for greater customization.

Other kinds of properties that you can get are:

- Read registry settings
- Env variables
- System parameters, IP address, system name, host name
- Script options currently set
- Any arbitrary user properties

Property Type	Property Name	Example
System.registry	Registry key to be read	<code>fgl_getproperty("system.registry", "HKEY_LOCAL_MACHINE\\SOFTWARE\\QUERIX\\Q4GL\\QUERIXDIR")</code>
System.environment	Environment variable name	<code>fgl_getproperty("system.environment", "TEMP")</code>
System.network	"IpAddress"	
System.network	"hostname"	
system	"username"	
user	User property name	<code>fgl_getproperty("user.hello", "myoption")</code>

Read Only Properties

Some properties are read only, and therefore can only be used with the `fgl_getproperty()` function. The following are read only:

gui.array.grid.enabled

This property will return a TRUE or FALSE value depending on whether the GUI client supports the Grid functionality or not.

Example function call:

```
CALL fgl_getproperty("gui", "gui.array.grid.enabled", "")
```

gui.array.grid.datamode

This will return the datamode that the Grid is operating in. For example, if the Grid is sending the records to the Grid a set amount at a time, this function call would return "paged".

Example function call:

```
CALL fgl_getproperty("gui", "gui.array.grid.datamode", "")
```

gui.array.serverhighlight

This property shows whether the server controls the array line highlight. The array highlight is by default controlled by the client, but if traditional 4GL behaviour is required then this option should be set to TRUE to allow the server to control the array highlighting.

Example function call:

```
CALL fgl_getproperty("gui", "gui.array.serverhighlight", "")
```

gui.array.grid.topvisiblerow

gui.array.grid.bottomvisiblerow

These properties will return the positions of the top and bottom visible rows in a grid. This is used to determine the section of data when exporting the current grid's view.

Example function calls:

```
CALL fgl_getproperty("gui", "gui.array.grid.topvisiblerow", "")
```

```
CALL fgl_getproperty("gui", "gui.array.grid.bottomvisiblerow", "")
```

browser.cookie

Used in Chimera only, this property will return a list of all the cookies on the system accessible to the applet in the standard form:

```
<cookie_1_name>=<cookie_1_data>; <cookie_2_name>=<cookie_2_data>; ...
```

An example function call:

```
CALL fgl_getproperty("gui", "browser.cookie", "")
```

browser.cookie.name

This Chimera only property will return the contents of a named cookie that is accessible to the applet. It will return an empty string if the named cookie cannot be found.

Example function call:

```
CALL fgl_getproperty("gui", "browser.cookie.name", "")
```

system.file.exists

This property is used to test whether a file, passed as part of the `fgl_getproperty()` function, exists on the client's file system. The file name passed in the function must either be relative to the directory in which the program is being run, or a complete file path if elsewhere. The function call will return a TRUE value if the file does exist and a FALSE value otherwise.

Example function call:

```
CALL fgl_getproperty("gui", "system.file.exists", "images/GO.jpg")
```

system.file.cachename

When external files are used in a 4GL program by the LyciaDesktop client these files are renamed and stored in a local cache. Sometime it may be necessary to refer to these files during the execution of a program. In order to do so, you will need to use this property to retrieve the cached file name.

Default files, such as toolbar items, do not get stored in the cache and so can be referred to by their original name.

This property requires the original name of the file to be passed to it and returns the full path of the cached file name.

Example function call:

```
CALL fgl_getproperty("gui", "system.file.cachename", "image/GO.jpg")
```

system.file.shortname

system.file.longname

Using these properties will retrieve the full path of the given file name. The long name is the original name of the file, including its full directory and the short name refers to the name assigned to it by Windows. Windows uses the 8.3 file naming system for files longer than 8 characters for backwards compatibility with older applications and operating systems.

Example function calls:

```
CALL fgl_getproperty("gui", "system.file.shortname", "docs/my_other_file.txt")
```

```
CALL fgl_getproperty("gui", "system.file.longname", "docs/my_other_file.txt.")
```

system.file.client_temp

This property will return the client directory that can be used to accommodate user files. This is usually a client specific sub-directory found in the currently logged in user's temporary system directory. If for whatever reason this specific sub-directory cannot be created, it will return the user's temporary system directory.

Example function call:

```
CALL fgl_getproperty("gui", "system.file.client_temp", "")
```

fgl_getuftype()

The following function returns a CHAR() string defining the current user interface type. The syntax is: fgl_getuftype(), and it will return one of the following values:

- CHAR A character-based terminal
- WTK A Windows front end
- JAVA A Java front end
- HTML An HTML front end
- WEB A web client

For some GUI types, the function performs a network round-trip. To improve performance, it is NOT recommended that this function is performed inside a loop.

Code Sample Using fgl_getuftype() function

```
DEFINE uitype CHAR(5)
DEFINE i INTEGER
LET uitype = fgl_getuftype()
FOR i=1 TO 100
    IF uitype = "CHAR" THEN
        ...
    ELSE
        ...
    END IF
END FOR
```

fgl_init4js()

The function `fgl_int4js()` is a function which is required for 4gl users using 4js/dynamic 4gl compiler. This function is not required in Querix but is recognized (and ignored) by the compiler for compatibility reasons.

```
MAIN
```

```
CALL fgl_init4js()
```

```
DISPLAY "FGL_INIT4JS() is not required for Querix compiled applications" at 5,5  
DISPLAY "But it does not generate a compile error - it's simply ignored" at 6,5  
sleep 5
```

```
END MAIN
```

fgl_report_type()

The fgl_report_type() function specifies the type of report pipe that will automatically be redirected to the client from the database server. Optionally it is also possible to specify the file type in which that report is sent.

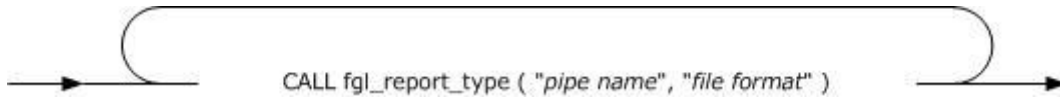


Figure 26 - fgl_report_type() Function

The function takes two arguments, the first of which is a CHAR(n) of the pipe name. The second argument, which is optional, is also a CHAR(n), and specifies the type of file format for the report to be sent.

The report is then viewed using the client-specified viewer for the user supplied file type.

Code Sample: fgl_report_type_function.4gl

```
MAIN
  DEFINE outpipe CHAR(10)

  CALL fgl_report_type("notepad","text")
  CALL fgl_report_type("printer","print")
  CALL fgl_report_type("word","word")

MENU "Print"
  COMMAND "Notepad" "Send report to notepad"
    LET outpipe = "notepad"
    START REPORT rep1 TO PIPE outpipe
    OUTPUT TO REPORT rep1 ()
    FINISH REPORT rep1

  COMMAND "Printer" "Send report to printer"
    LET outpipe = "printer"
    START REPORT rep1 TO PIPE outpipe
    OUTPUT TO REPORT rep1 ()
    FINISH REPORT rep1

  COMMAND "Word" "Send report to word"
    LET outpipe = "word"
    START REPORT rep1 TO PIPE outpipe
    OUTPUT TO REPORT rep1 ()
    FINISH REPORT rep1
```

```
        COMMAND "Exit" "Exit program"
        EXIT PROGRAM
    END MENU
END MAIN

REPORT repl()
    FORMAT ON EVERY ROW
    PRINT "This report is being sent to the program associated with a file type"
    PRINT ""
    PRINT "The program is defined within the .qxs file using statements of the
form:"
    PRINT "  default.viewer.<file_type>: program"
    PRINT ""
    PRINT "Querix 4GL uses a number of default viewers for reports:"
    PRINT ""
    PRINT COLUMN 1, "Report Type",
        COLUMN 20, "Phoenix viewer",
        COLUMN 40, "Chimera viewer"
    PRINT COLUMN 1, "-----",
        COLUMN 20, "-----",
        COLUMN 40, "-----"
    PRINT COLUMN 1, "text",
        COLUMN 20, "textview.exe",
        COLUMN 40, "built-in"
    PRINT COLUMN 1, "print",
        COLUMN 20, "printfile.exe",
        COLUMN 40, "built-in"
    PRINT COLUMN 1, "html",
        COLUMN 20, "iexplore.exe",
        COLUMN 40, "built-in"
END REPORT
```


fgl_setproperty()

fgl_setproperty is the same as fgl_getproperty, except that it *sets* rather than *gets* values.

The function is used in the following way:

```
fgl_setproperty(["gui"], type.name, value)
```

In the above syntax, "gui" refers to the subsystem that the property belongs to. This optional parameter defaults to "gui", so it can be omitted. The type is the element of the subsystem that the property affects; the name is the actual name of the property. Value, which is primarily used when editing properties, can either be the new desired setting when used with fgl_setproperty(), or a specific location/file when used with fgl_getproperty(). If there is no value required in a fgl_getproperty() call, then simply enter two sets of quotation marks.

This function overrides any matching settings specified in a script file.

Other types of properties that you can set are:

- Registry settings
- Environment variables
- System parameters, IP addresses, system name, host name
- Script options
- Any arbitrary user properties

Property Type	Property Name	Example
System.registry	Registry key to be read	fgl_setproperty("gui", "system.registry", "HKEY_LOCAL_MACHINE\\SOFTWARE\\QUERIX\\Q4GL\\QUERIXDIR")
System.environment	Environment variable name	fgl_setproperty("gui", "system.environment", "TEMP")
System.network	"Ipaddress"	
System.network	"hostname"	
system	"username"	
user	User property name	fgl_setproperty("gui", "user.hello", "myoption")

Editable Properties

The following GUI properties are used with `fgl_setproperty()`. They are used to edit properties rather than simply returning the non-specifiable properties.

window.background.image

Use this to set the background image used in the 4GL program. The path containing the image must be either the full path or relative to the 4GL program.

Example function call:

```
CALL fgl_setproperty("gui", "window.background.image", "image/GO.jpg")
```

window.background.style

Similar to the above, use this in the function call to set the style of the background used in the program. The name of the style must be provided with the function call, i.e., normal, stretched, tiled and centered.

Example function call:

```
CALL fgl_setproperty("gui", "window.background.style", "tiled")
```

window.background.x

window.background.y

These properties are used to set the position of the image used in the background on the x and y axes respectively. An integer position must be included in the function call that is within the boundaries of the screen.

Example function calls:

```
CALL fgl_setproperty("gui", "window.background.x", "5")  
CALL fgl_setproperty("gui", "window.background.y", "20")
```

window.background.width

window.background.height

Again, used in conjunction with the background image property, these will set the size of the image used in the background. As with the x and y positioning, an integer must be sent with the function call.

Example function calls:

```
CALL fgl_setproperty("gui", "window.background.width", "50")  
CALL fgl_setproperty("gui", "window.background.height", "100")
```

window.background.color

This property sets the background colour for the current window. The colour must be defined as one of the following – "default", "black", "white", "blue", "red", "green", "yellow", "magenta", or "cyan".

Example function call:

```
CALL fgl_setproperty("gui", "window.background.color", "blue")
```

window.doubleclick

This property is used to set the double-click event for the current window. The default event is "accept", but can be changed to any key press.

Example function call:

```
CALL fgl_setproperty("gui", "window.doubleclick", "F2")
```

window.icon

This property is used to set the icon to be used for the current window. If the image is stored in a different directory to the 4GL application the function call must include the full path of the image, otherwise the relative path is required.

Example function call:

```
CALL fgl_setproperty("gui", "window.icon", "images/OK.ico")
```

window.errorline

window.commentline

window.menuhelptextline

These properties are all used to determine the positions of any messages. As their names suggest, they set the placement of error messages, comments and menu help messages respectively. In the function call it is possible to specify a form, a line number, the statusbar, or a messagebox to send the messages to.

Example function calls:

```
CALL fgl_setproperty("gui", "window.errorline", "2")  
CALL fgl_setproperty("gui", "window.commentline", "statusbar")  
CALL fgl_setproperty("gui", "window.menuhelptextline", "myForm")
```

window.textfont

Use this window property to set the font used in the current window. This will only affect the text after this property has been set. In the function call, a font must be defined in the following format – "Font Name-Weight-Style--Size". In this definition, the Font Name is the name of the required font, the Weight is how heavy the font should be, i.e., light, medium or bold, the Style refers to whether the font should be regular(r), underlined(u), or italic(i), and the size is the point size of the font.

Example function call:

```
CALL fgl_setproperty("gui", "window.textfont", "Tahoma-medium-r--10")
```

window.fieldfont

This window property will set the font for use in any subsequent fields. The method of font declaration is the same as with gui.window.textfont.

Example function call:

```
CALL fgl_setproperty("gui", "window.fieldfont", "Tahoma-bold-u--14")
```

array.grid.cellfont

Use this property to define the font used in the cells of the grid. This will only affect the cells declared after this property has been set. Again, the font is declared in the same way as gui.window.textfont.

Example function call:

```
CALL fgl_setproperty("gui", "array.grid.cellfont", "Arial-light-i--8")
```

array.grid.headerfont

This property is used to change the font used for a grid header. Again, it will only affect subsequent header declarations and the font is declared in the same way as gui.window.textfont.

Example function call:

```
CALL fgl_setproperty("gui", "array.grid.headerfont", "Times New Roman-bold-r--  
24")
```

window.winshellexec.verb

This window property is used to set the action performed on a win[shell]exec() style call. This can be one of either "open" (default), "edit", "explore", "print", or "properties".

Example function call:

```
CALL fgl_setproperty("gui", "window.winshellexec.verb", "print")
```

window.winshellexec.show

This window property is used to define the style of window used when performing a win[shell]exec() style call. This can be any of the usual window size options – maximized, minimized, and also invisible.

Example function call:

```
CALL fgl_setproperty("gui", "window.winshellexec.show", "minimized")
```

array.grid.csvseparator

This grid property is used to set the separator when exporting data from a grid. The default separator is a comma, but this can be changed to be any character or group of characters by passing them to the function call.

Example function call:

```
CALL fgl_setproperty("gui", "array.grid.csvseparator", ";")
```

default.font

This property will set the default font to be used for window grid size calculation. This will affect any window opened after the function call.

Example function call:

```
CALL fgl_setproperty("gui", "default.font", "Verdana-medium-r--12")
```

browser.cookie

This Chimera only property will write data to the cookie named 'chimera' on the system accessible to the applet.

Example function call:

```
CALL fgl_setproperty("gui", "browser.cookie", "data")
```

browser.cookie.name

This Chimera only property will write data to the cookie specified by name on the system accessible to the applet.

Example function call:

```
CALL fgl_setproperty("gui", "browser.cookie.name", "data")
```

ui.INTERFACE Class

The Interface class is a built-in class which belongs to the ui namespace. The class is designed to manipulate the user interface.

The Interface class supports a set of class methods. Since it supports only class methods, you do not need to use the DEFINE statement to initialize an object. The general syntax of the methods invocation is:

```
CALL ui.Interface.Method()
```

GetFrontEndName()

The GetFrontEndName() method is used to retrieve the type of the front-end which the application uses. The method is typically used in debugging. It returns different values depending on the client used to run the application. The returned value can be one of the following character strings with the corresponding meaning:

- *lyciadesktop* an application is running with LyciaDesktop
- *lyciadjax* an application is running with LyciaAjax
- *lyciamobile* an application is running on a mobile device
- *console* an application is running in text mode

GetFrontEndVersion()

The GetFrontEndVersion() method returns the version of the client which the application uses. The method is typically used in debugging.

FrontCall()

This method is used to call non-4gl applications at the client side.

The syntax of the method invocation is as follows:

```
CALL ui.Interface.FrontCall(module, function, parameters_list, returning_list)
```

Where:

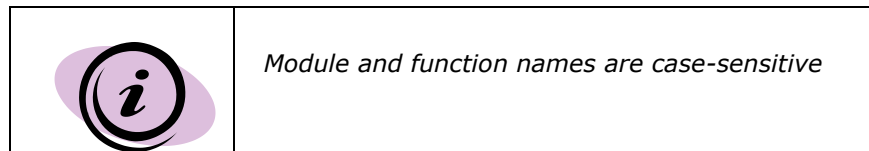
- *module* stands for a string value identifying the name of the shared library in which the function to be executed is specified;
- *function* stands for a string value identifying the name of the function to be called;
- *parameters_list* stands for the list of the necessary input parameters;

- *returning_list* stands for the list of the output parameters.

You should specify the input and output parameters as a list, within square braces [parameter_1, parameter_2, parameter_3,...]. The input parameters can be represented as expressions, but the output parameters must be variables only, so that they could get the returned values. The output parameters specification is optional; the system will ignore the values returned by the front-end. An empty parameters list should be identified as empty square braces.

If there are problems with input or output parameter specification, an error can arise. The number of the input and output variables is controlled by the front-end. If the program receives the execution status zero (Ok) from the front end and the number of the program and returned variables does not match, all the unmatched program variables will be initialized to NULL. Generally, it is advised that the program provides the expected input and output parameters according to the documentation.

When the module name is specified as "standard", the application will reference the front-end default built-in functions. Other module names specify .DLL and .so libraries; they must be installed in the front-end installation directory. You can see the front-end documentation for more details. You can write your own modules to customize the front-end calls.



For some of the front calls (e.g., *shellexec*), it is necessary to pass a parameter containing the file path. This path should correspond to the syntax required by the front-end file system. Below is given an example of passing a file path to the front-end on a Microsoft Windows system:

MAIN

```
DEFINE filepath STRING, res INTEGER
```

```
LET filepath = "\"c:\\dir\\my report.doc\""
```

```
-- This is: "c:\\work dir\\my report.doc"
```

```
CALL ui.Interface.frontCall( "standard", "shellexec", [path], [res] )
```

END MAIN

Refresh()

The Refresh () method is used to synchronize the client with the server. Normally the client refreshes the screen only when there is a user interaction. If there is not user interaction, but you still need the client to refresh the displayed data, use this method. You can change the refresh mode using the OPTIONS AUTOREFRESH statement. For more information about the refreshment modes see the OPTIONS statement description in Querix 4GL Reference.

Here is an example of the refresh() method used in the first refreshment mode when the displayed values become visible only once a screen interaction statement was executed:

```
MAIN
  DEFINE f001 INTEGER
  OPEN WINDOW w WITH FORM "various"
  FOR f001=1 TO 10
    DISPLAY BY NAME f001
    # Without this line the program will beexecuted silently and finish
    # without the user being able to see the output at all.
    CALL ui.Interface.refresh()
    SLEEP 1
  END FOR
END MAIN
```


Graphical Theme Files

Lycia II allows creating and applying graphical theme files which specify graphical and behavioural settings. Each application has a default theme file that can be changed by means of Theme Designer.

However, it is possible to create and manipulate non-default theme files by using the corresponding built-in functions.

The details of the theme files manipulation are given in Theme Designer Guide.

apply_theme()

The function *apply_theme()* is used to apply themes to your program. Its syntax is as follows:

```
CALL apply_theme(theme_file)
```

where *theme_file* stands for a string value identifying the name of the theme to be applied.
Example function call:

```
CALL apply_theme("my_styles")
```



Some of the properties are applied to the application right after its starting. These properties won't have effect if they are added or changed at application runtime.

Window

This section covers the 4GL built-in functions which apply to windows declared implicitly by means of the OPEN WINDOW statement or by means of the built-in functions. It also contains the descriptions of the methods used with the variables of the WINDOW data type. The usage of methods is described [above](#).

fgl_drawbox()

The fgl_drawbox() function displays a rectangle of a specified size.

It takes comma-separated arguments for the *height*, *width*, *line*, *left-offset* and may also take *color*.

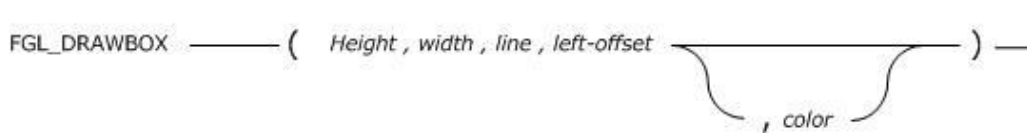


Figure 27 - fgl_drawbox() Function

Usage

The fgl_drawbox() function draws a rectangle with the upper-left corner at (*line*, *left-offset*) with the dimensions specified by the *height* and *width* values. These dimensions must have positive integer values, in units of lines and character positions, where (1,1) is the upper-left corner of the current 4GL window.

Element	Description
<i>color</i>	<i>color</i> is an integer expression that returns a positive whole number, specifying a foreground colour code. In practice 'foreground colour' means the colour of the border.
<i>height</i>	<i>height</i> is an integer expression; it defines the unit height of the rectangle measured in screen lines.
<i>left-offset</i>	<i>left-offset</i> is an integer expression; it defines the horizontal distance of the top left-hand corner of the rectangle, measured in characters, from the left-hand edge of the screen, where the first character space is represented by the figure 1.
<i>line</i>	<i>line</i> is an integer expression; it defines the vertical distance of the top left-hand corner of the rectangle, measured in screen lines, from the upper edge of the screen, where the first screen line is represented by the figure 1.

<i>width</i>	<i>height</i> is an integer expression; it defines the width of the outline of the rectangle, measured in characters.
--------------	---

The optional colour number must correspond to one of the following foreground colours.

Colour Number	Foreground Colour	Colour Number	Foreground Colour
0	WHITE	4	CYAN
1	YELLOW	5	GREEN
2	MAGENTA	6	BLUE
3	RED	7	BLACK

The **upscol** utility can specify these same colour options in the **syscolatt** table. The default colour is used when the colour number is omitted. The colour argument is optional.

As is the case with borders, the width of the line that draws the rectangle is fixed. This fixed width cannot be specified or modified when you invoke `fgl_drawbox()`. As with borders, 4GL draws the box with the characters defined in the **termcap** or **terminfo** files. You can specify alternative characters in these files. Otherwise, 4GL uses hyphens to create horizontal lines, pipe symbols (|) for vertical lines, and plus signs at the corners. To assign the box a colour, you must use **termcap** because **terminfo** does not support colour. The **termcap** and **terminfo** files are discussed in detail in the Appendix: "Modifying termcap and terminfo."

For example:

```

MAIN
  DEFINE inp_char CHAR(1)
  OPTIONS PROMPT LINE 24
  DISPLAY "***** FGL_DRAWBOX() Function example *****" AT 1,5

  CALL fgl_drawbox(10, 60, 3, 2)
  DISPLAY "This is a fgl_drawbox with the arguments" at 8,5
  DISPLAY "fgl_drawbox(10,60,3,2)" at 9,5
  DISPLAY "Height = 10 Lines, Width = 60 Columns" at 10,5
  DISPLAY "on Line 3, Column 2" at 11,5

  CALL fgl_drawbox(10,60,14,2,3)
  DISPLAY "This is a fgl_drawbox with the arguments" at 16,5
  DISPLAY "fgl_drawbox(10,60,14,2,3)" at 17,5
  DISPLAY "Height = 10 Lines, Width = 60 Columns" at 18,5
  DISPLAY "on Line 14, Column 2, with the color 3, which is red" at 19,5


  PROMPT "Press any key to close this demo application" FOR inp_char

END MAIN

```

Rectangles drawn by `fgl_drawbox()` are part of a displayed form. Each time that you execute the corresponding `DISPLAY FORM` or `OPEN WINDOW ... WITH FORM` statement, you must also redraw the rectangle.

If you invoke `fgl_drawbox()` several times to create a display in which rectangles intersect, output from the most recent function call overlies any previously drawn rectangles. Screen fields and reserved lines, however, have a higher display priority than `fgl_drawbox()` rectangles, regardless of the order in which the fields, lines, and rectangles are drawn.

	<p>In most applications, avoid drawing rectangles that intersect or overlap any field or reserved line. Reserved lines might be redrawn frequently during user interaction statements, partially erasing any rectangles at the intersections where they overlap the reserved lines. To avoid this problem, position the rectangles so they do not overlap any reserved lines or screen fields.</p>
---	--

fgl_drawline ()

The `fgl_drawline()` function displays a horizontal line of a specified size. It takes comma-separated arguments for the *column*, *row*, *width* and *color* of the line.

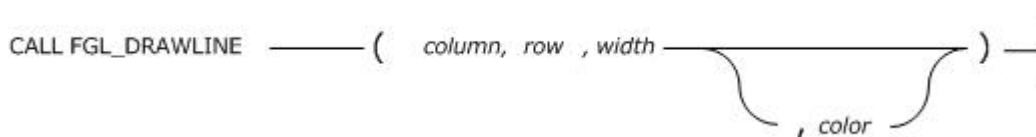


Figure 28 - fgl_drawline()

Usage

The *column* and *row* values are INTEGER values for the start point, on the screen, of the line. The *width* value is the number of screen columns across which the line will be displayed.

The *color* argument is optional. The number must correspond to one of the following foreground colours:

Colour Number	Foreground Colour	Colour Number	Foreground Colour
0	WHITE	4	CYAN
1	YELLOW	5	GREEN
2	MAGENTA	6	BLUE
3	RED	7	BLACK

These numbers are the same as those used by the `fgl_drawbox()` function.

```

MAIN
  DEFINE i INTEGER
  DEFINE x, y, length INTEGER

  LET x = 3
  LET length = 50

  OPEN WINDOW w_test
    AT 2, 2
    WITH 20 ROWS, 76 COLUMNS

  FOR i = 1 TO 7
    LET y = i + 2
    CALL fgl_drawline(30+x,y,length,i)

```

```
#      DISPLAY "fgl_drawline(", 30+x CLIPPED, ",", y CLIPPED, ",", length CLIPPED, ",",  
i CLIPPED, ")", i USING "<&", ":" AT y, x  
      DISPLAY "fgl_drawline(" || 30+x CLIPPED || ",", y CLIPPED || ",", length ||  
", " || i CLIPPED || ")", i USING "<&", ":" AT y, x  
      END FOR  
  
      CALL fgl_message_box("Exit this application")  
END MAIN
```

fgl_getwin_height

The fgl_getwin_height function returns the height in rows of the currently active program window.

The fgl_getwin_height demonstration produces a window that displays the height of the program window. It also shows the results of the fgl_getwin_width, fgl_getwin_x, and fgl_getwin_y functions. The sample demonstration applications for each of these functions are identical.

Sample Code: fgl_getwin_height_function.4gl

```
MAIN
  DEFINE x INTEGER
  DEFINE y INTEGER
  DEFINE wid INTEGER
  DEFINE hgt INTEGER

  OPEN WINDOW w_test
    AT 4, 4
    WITH 15 ROWS, 50 COLUMNS
    ATTRIBUTE (BORDER)

    LET x = fgl_getwin_x()
    LET y = fgl_getwin_y()
    LET wid = fgl_getwin_width()
    LET hgt = fgl_getwin_height()

    DISPLAY "Location: ", x USING "<&", ", ", y USING "<&" AT 2, 3
    DISPLAY "Width: ", wid USING "<&", " columns" AT 3, 3
    DISPLAY "Height: ", hgt USING "<&", " rows" AT 4, 3

    CALL fgl_message_box("Press any key to close this application")
  END MAIN
```

fgl_getwin_width

The fgl_getwin_width function returns the width, in columns, of the currently active 4GL program window. See details of the fgl_getwin_height function for the code sample.

fgl_getwin_x

The `fgl_getwin_x` function returns the column at which the currently active 4GL window was drawn. For example, in the following fragment:

```
OPEN WINDOW w_test
  AT 3, 2
...
```

You have opened the window at column 2. For this window, the `fgl_getwin_x` function would return the value 2.

See details of the `fgl_getwin_height` function for the code sample.

fgl_getwin_y

The `fgl_getwin_y` function returns the row at which the currently active program window was opened. For example:

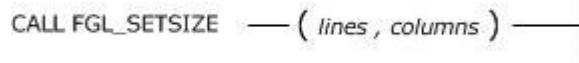
```
OPEN WINDOW w_test
  AT 4, 5
...
```

The window is opened at row 4 of the 4GL screen. In this case, the `fgl_getwin_y` function will return 4.

See details of the `fgl_getwin_height` function for the code sample.

fgl_setsize() §

The `fgl_setsize()` function allows you to change the default size of the program window. The function takes two INTEGER arguments, *lines* and *columns*, which specify the size at which the program window is displayed.



```
CALL FGL_SETSIZE ( lines , columns )
```

Figure 29 - fgl_setsize() Function

Code Sample: fgl_setsize_function.4gl

```
MAIN

    OPEN WINDOW win1 AT 1,1 WITH 24 rows, 80 columns ATTRIBUTES (BORDER)
    CALL report_winsize()

    CALL fgl_winmessage("fgl_setsize()", "Setting window size to 15 rows, 75
columns", "info")

    CALL fgl_setsize(75,15)
    CALL report_winsize()

    CALL fgl_winmessage("fgl_setsize()", "Setting window size to 10 rows, 70
columns", "info")

    CALL fgl_setsize(70,10)
    CALL report_winsize()

END MAIN

FUNCTION report_winsize()
    DEFINE y, x INTEGER
    DEFINE msg CHAR(60)
    CALL fgl_winsize() RETURNING y, x

    LET msg = "Window w_size has ", x USING "<&", " columns, and ", y USING "<&", "
rows"
    CALL fgl_winmessage("fgl_winsize()",msg,"info")
END FUNCTION
```

fgl_settitle() §

The fgl_settitle() function allows you to set the title of a program window. This will override the default action, which is to display the name of the current program.

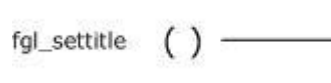


Figure 30 - fgl_settitle() Function

Usage

The function takes one argument, which is a string or variable that contains the new title of the window that is to be displayed.

Code Sample: fgl_settitle_function.4gl

```
MAIN

    DISPLAY "fgl_settitle() demo application" at 1,1

    CALL fgl_winmessage("fgl_settitle()", "Specify a program title for the titlebar
(GUI only) at runtime using fgl_settitle()", "info")

    CALL fgl_settitle("This is my new title specified using fgl_settitle()")

    DISPLAY "END of fgl_settitle() demo application" at 2,1

    CALL fgl_winmessage("fgl_settitle()", "Press OK to close the demo application",
"info")
END MAIN
```

fgl_window_clear

The `fgl_window_clear` function clears the contents of a window even if it is not the current window. The function takes a window name as its argument, which can be any window that is open.

Usage

For example, if two windows are opened as follows:

```
OPEN WINDOW win1 AT 3,3 WITH 20 ROWS, 70 COLUMNS ATTRIBUTES(BORDER)
OPEN WINDOW win2 AT 6,10 WITH 10 ROWS, 60 COLUMNS ATTRIBUTES(BORDER)
```

Assuming both windows have contents, make win2 the current window as follows:

```
CURRENT WINDOW IS win2
```

Then clear the contents of win1:

```
fgl_window_clear("win1")
```

This would keep the window open and win2 would still be current, but the contents of win1 would have been erased.

fgl_window_close()

The fgl_window_close() function closes a window. It takes as its argument the string which is the name of the window to be closed.

Sample Code: fgl_window_close.4gl

```
MAIN

OPEN WINDOW win1 AT 3,3 WITH 20 ROWS, 70 COLUMNS ATTRIBUTES(BORDER)

CALL fgl_winmessage("fgl_window_close()", "WIN1: This window was opened as
'win1'", "info")

OPEN WINDOW win2 AT 6,10 WITH 10 ROWS, 60 COLUMNS ATTRIBUTES(BORDER)

CALL fgl_winmessage("fgl_window_close()", "WIN2: This window was opened as
'win2'", "info")

CALL fgl_winmessage("fgl_window_close()", "Now, we close the window 'win1' using
fgl_window_close(\"win1\")", "info")

CALL fgl_window_close("win1")

CALL fgl_winmessage("fgl_window_close()", "WIN1: Window 'Win1' is now closed",
"info")

CALL fgl_winmessage("fgl_window_close()", "Now, we close the window 'win2' using
fgl_window_close(\"win2\")", "info")

CALL fgl_window_close("win2")

CALL fgl_winmessage("fgl_window_close()", "WIN2: Window 'Win2' is now closed -
Press OK to close application", "info")

END MAIN
```

fgl_window_current()

The `fgl_window_current()` function makes the specified window, named *window1* in this example, the active window.

```
call fgl_window_current("window1")
```

Sample Code: fgl_window_current_function.4gl

```
MAIN
```

```
OPEN WINDOW win1 AT 3,3 WITH 20 ROWS, 70 COLUMNS ATTRIBUTES(BORDER)
```

```
CALL fgl_winmessage("fgl_window_current()", "WIN1: This window was opened as  
'win1' and now has focus", "info")
```

```
OPEN WINDOW win2 AT 6,10 WITH 10 ROWS, 60 COLUMNS ATTRIBUTES(BORDER)
```

```
CALL fgl_winmessage("fgl_window_current()", "WIN2: This window was opened as  
'win2' and now has focus", "info")
```

```
CALL fgl_winmessage("fgl_window_current()", "Now, we change focus to 'win1' using  
fgl_window_current()", "info")
```

```
CALL fgl_window_current("win1")
```

```
CALL fgl_winmessage("fgl_window_current()", "WIN1: Window 'Win1' has now got  
focus again", "info")
```

```
CALL fgl_winmessage("fgl_window_current()", "Now, we change focus to 'win2' using  
fgl_window_current()", "info")
```

```
CALL fgl_window_current("win2")
```

```
CALL fgl_winmessage("fgl_window_current()", "WIN2: Window 'Win2' has now got  
focus again - Press OK to close application", "info")
```

```
END MAIN
```

fgl_window_getoption()

The `fgl_window_getoption()` function returns information about the current application window. It returns a string value in Lycia II instead of integer value it used to return in earlier versions, since the new functionality requires it.

Usage

This function uses the syntax: `fgl_window_getoption (option)`, with any of the *options* shown in the following table.

option	description
name	The name of the window.
x	The column position.
y	The line position.
width	The width.
height	The height.
border	TRUE if window has a border.
formline	The form line.
menuline	The menu line.
commentline	The comment line.
messageline	The message line.
errorline	The error line.
insertkey	The value of insertkey .
deletekey	The value of deletekey .
nextkey	The value of nextkey .
previouskey	The value of previouskey .
acceptkey	The value of acceptkey .
helpkey	The value of helpkey .
abortkey	The value of abortkey .
inputwrap	TRUE if input wrap option is used.
fieldorder	0 if the field order is UNCONSTRAINED.

	1 if field order option is CONSTRAINED. 2 if the field order is FORM.
--	--

Values returned for keys are the same as the values returned by the `fgl_getkey()` function.

Code Samples

The code samples for the `fgl_window_getoption()` function are long and have not been reproduced here.

For complete details please see the executable programs in the 'functions' demonstration application package supplied with Lycia.

fgl_window_open()

"Querix4GL" has the ability to open windows with a dynamic name by using the fgl_window_open() function. This opens the window in the specified position, makes it the required size and gives it the name as provided in the function call. Also in the function call is the optional parameter of a form to be opened in that window.

Usage

The function fgl_window_open() is synonymous with the OPEN WINDOW statement, and functions in the same manner. It takes as its arguments the window's name, its position and size and defines whether the window has a border or not.

For example:

```
CALL fgl_window_open("w_test", 2, 2, 20, 60, TRUE)
```

is functionally identical to the statement:

```
OPEN WINDOW w_test  
AT 2, 2  
WITH 20 ROWS, 60 COLUMNS  
ATTRIBUTE (BORDER)
```

This function also has the optional parameter of a form; including this parameter will open the window at the specified location with the dimensions as specified in the form definition. This means there is no need to include the ROWS or COLUMNS when opening a form.

For example

```
CALL fgl_window_open("w_test", 2, 2, "form1", TRUE)
```

is functionally equivalent to the statement:

```
OPEN WINDOW w_test  
AT 2, 2  
WITH FORM "form1"  
ATTRIBUTE (BORDER)
```

It is to be noted that the traditional method of closing windows, using the 4GL code:

```
CLOSE WINDOW(window_name)
```

will not work when opening windows that were opened with fgl_window_open().

fgl_winmessage()

The fgl_winmessage() function displays an interactive message box in a separate window with some text:

```
CALL fgl_winmessage ( title , text , icon )
```

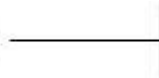


Figure 31 - fgl_winmessage()

The variables *title*, *text*, and *icon* are all CHAR(n) datatype variables which return the label for the button selected. *title* is the title for the message box, *text* is the text to be displayed in the message box, and *icon* is the name of the icon or graphic to be used in the message box.

The following example would produce a message dialog such as the one shown below the code sample.

Code Sample

```
MAIN

CALL fgl_winmessage( "Title of the message",
    "Text or variable", "info" )

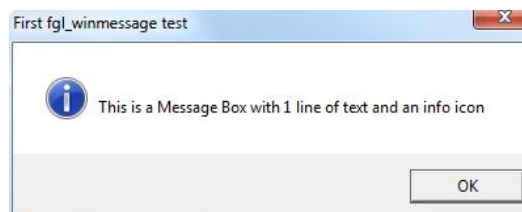
END MAIN
```

Simple message dialogs

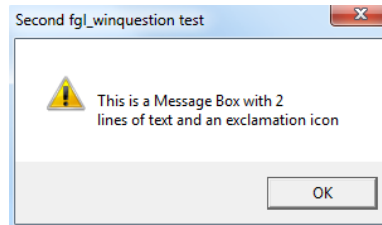
The message dialog boxes generated by the sample code in the demonstration application are shown below, inserted within the application code.

```
MAIN

CALL fgl_winmessage("First fgl_winmessage test", "This is a Message Box with 1
line of text and an info icon", "info")
```



```
CALL fgl_winmessage("Second fgl_winquestion test", "This is a Message Box with 2  
\nlines of text and an exclamation icon", "exclamation")
```



```
CALL fgl_winmessage("Third fgl_winquestion test", "This is a Message\nBox with 3  
lines\nnof text and a stop icon", "Stop")
```



END MAIN

fgl_winprompt()

The fgl_winprompt() function displays a dialog box with a field that accepts a value.

→ fgl_winprompt (*x* , *y* , *text* , *default* , *length* , *type*) →

Figure 32 - fgl_winprompt() Function

Usage

The fgl_winprompt() function takes a series of arguments that define details of the dialog box that is displayed. This dialog box will have an input field, whose length can be specified and which can have some text displayed in it by default. The type of value that the field will accept is also definable.

The arguments are described in the table below.

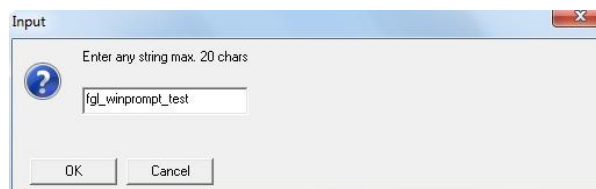
<i>x</i>	INTEGER	Column position.
<i>y</i>	INTEGER	Line position.
<i>text</i>	CHAR(n)	Message.
<i>default</i>	CHAR(n)	Default text placed in input field.
<i>length</i>	INTEGER	Maximum input length.
<i>type</i>	INTEGER	Data type code (0=CHAR, 1=SMALLINT, 2=INTEGER, 7=DATE, 255=invisible).
Returns	CHAR(n)	Input value.

Demonstration application: fgl_winprompt_function.exe

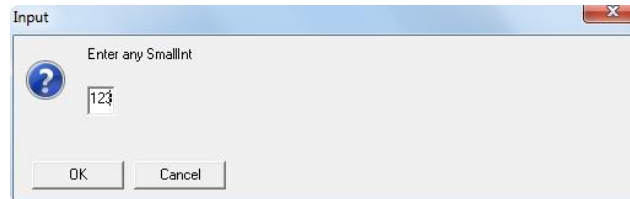
The prompt dialogs generated by the following demonstration application are shown within the code.

```
MAIN
  DEFINE value_string char(20)

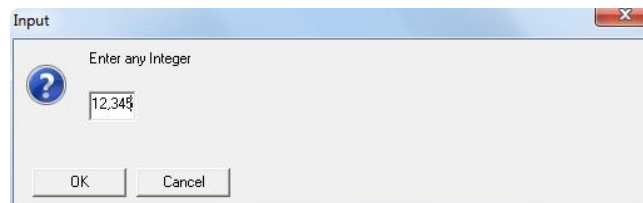
  LET value_string = fgl_winprompt(5,5, "Enter any string max. 20 chars", "", 20,
0)
  CALL value_entered(value_string)
```



```
LET value_string = fgl_winprompt(10,5,"Enter any SmallInt","", 4, 1)
CALL value_entered(value_string)
```



```
LET value_string = fgl_winprompt(15,5,"Enter any Integer","", 6, 2)
CALL value_entered(value_string)
```



```
END MAIN
```

```
FUNCTION value_entered(value_string)
  DEFINE value_string CHAR(20)
```

```
  CALL fgl_winmessage("Value Entered", "You entered the following value: ",
value_string, "info")
```

```
END FUNCTION
```

fgl_winquestion()

The fgl_winquestion() function displays an interactive message box in a separate window with a combination of system defined buttons.

→ fgl_winquestion (— title , text , default , buttons , icon , danger —) →

Figure 33 - fgl_winquestion()

Usage

The fgl_winquestion() function takes a series of arguments that define details of the message box that is displayed. This message box will have a maximum of three buttons, any one of which can be clicked. The function will return the value of the button clicked by the user.

The arguments are described in the table below.

<i>title</i>	CHAR(n)	Message box title.
<i>text</i>	CHAR(n)	Message box message (use \n for new line).
<i>default</i>	CHAR(n)	Defines the default button that is pre-selected.
<i>buttons</i>	CHAR(n)	A list of values, separated by pipes (), that are the text within each of the buttons to be displayed. Buttons can only be one of the following combinations: <ul style="list-style-type: none"> • OK • OK Cancel • Yes No • Yes No Cancel • Retry Cancel • Abort Retry Ignore
<i>icon</i>	CHAR(n)	Name of icon to be used (info, exclamation, question, stop).
<i>danger</i>	SMALLINT	This is for X11 use only, and is only present for compatibility.
Returns	CHAR(n)	The label of the button selected by the user.

The following code sample produces a series of windows, which are shown in the screenshots after the code:

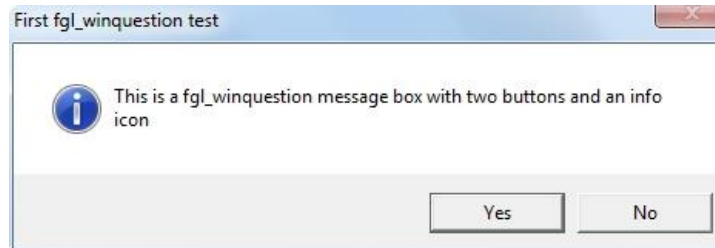
Demonstration Application 4GL File: fgl_winquestion.4gl

The question dialogs generated by the following demonstration application are shown within the code.

MAIN

```
    DEFINE button_string char(20)
```

```
    LET button_string = fgl_winquestion("First fgl_winquestion test", "This is a  
fgl_winquestion message box with two buttons and an info icon", "Yes", "Yes|No",  
"info", 1)
```



```
    CALL button_pressed(button_string)
```

```
    LET button_string = fgl_winquestion("Second fgl_winquestion test", "This is a  
fgl_winquestion message box with three buttons and an info icon", "Yes",  
"Yes|No|Cancel|", "info", 1)
```



END MAIN

```
FUNCTION button_pressed(button_string)
```

```
    DEFINE button_string CHAR(20)
```

```
    CALL fgl_winmessage("Button Pressed", "You Pressed the button: ",  
button_string, "info")
```

```
END FUNCTION
```

fgl_winsize()

The `fgl_winsize()` function takes as its argument a string which is the name of a window. It returns two INTEGER values, which are the number of screen columns and the number of screen rows that the specified window covers.

Usage

The code sample shown opens three different windows in sequence. The code then uses the `fgl_winmessage()` function to create a message box. Within that box the `fgl_winsize()` function is used to display 'msg' as its text. That 'msg' is configured to include the return values from `fgl_winsize` within it. When you acknowledge one winsize message the next window and message are displayed

The first winsize message box looks like this:

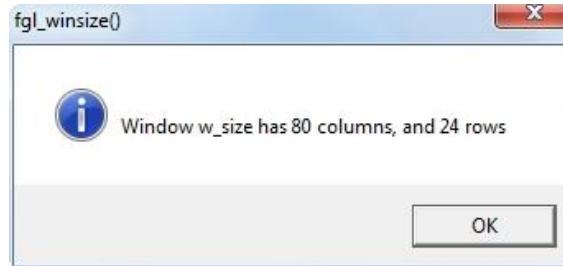


Figure 34 - fgl_winsize() Message Box

MAIN

```
OPEN WINDOW w_size1
  AT 1, 1
  WITH 24 ROWS, 80 COLUMNS
  ATTRIBUTE(GREEN)
  CALL report_winsize()

OPEN WINDOW w_size2
  AT 3, 3
  WITH 20 ROWS, 76 COLUMNS
  ATTRIBUTE(BORDER)
  CALL report_winsize()

OPEN WINDOW w_size3
  AT 5, 5
  WITH 16 ROWS, 72 COLUMNS
  ATTRIBUTE (WHITE, BORDER)

  CALL report_winsize()
```

```
#SLEEP 30
END MAIN

FUNCTION report_winsize()
  DEFINE y, x INTEGER
  DEFINE msg CHAR(60)
  CALL fgl_winsize() RETURNING y, x

  LET msg = "Window w_size has ", x USING "<&", " columns, and ", y USING "<&", "
rows"
  CALL fgl_winmessage("fgl_winsize()",msg,"info")

END FUNCTION
```


clear()

variable.Clear()

This method is used with a variable of the WINDOW data type to clear a window previously opened by the Open() method or by the OpenWithForm() method. If you use the method to clear the window containing a with form, the form will become invisible but won't be closed.

If the form was opened by OPEN FORM statement or .Open() method, it can be easily redisplayed by the DISPLAY FORM statement or .Display() method.

If the form was opened together with the window (OpenWithForm() method) and was cleared from the window, it cannot be redisplayed.

Usage

This method accepts no arguments and returns no values. It clears the window associated with the WINDOW variable it is used with regardless of which window is the current one.

```
CALL win_var.Clear()
```

close()

variable.Close()

This method is used with a variable of the WINDOW data type to close a window previously opened by the Open() method or by the OpenWithForm() method. Closing a window opened with form closes also the form. Closing a window frees a WINDOW variable used and it can be used with the Open() or the OpenWithForm() method again to open another window.

Usage

This method accepts no arguments and returns no values. It closes the window associated with the WINDOW variable it is used with regardless of which window is the current one.

```
CALL win_var.Close()
```

displayAt()

`variable.DisplayAt("message", y, x [, attributes])`

This method is used with a variable of the WINDOW data type and displays a text to a definite location in the window specified by the variable. To be able to use this method, you should first initialize the WINDOW variable with the help of the Open() method.

This method displays the specified character string to the window associated with the variable used. If this window wasn't the current one, it is made the current window.

Usage

This method accepts four arguments. The first argument is a quoted string or a variable of a character data type which is to be displayed to the window. *Y* stands for the row and *x* for the column at which the message will be displayed, these are both literal integers or integer variables. The *attributes* argument is a quoted string containing the standard display attributes of colour and intensity. It is optional.

There may be more than one attribute in the character string, and then they should be separated by commas. Here are the attributes you can use:

- Black, white, yellow, green, magenta, red, blue, cyan
- Invisible, underline, reverse, blink
- Normal, dim, bold

An example of the method call:

```
CALL win_var.DisplayAt("The message to display", 2, 5, "red, bold")
```

error()

variable.Error("message" [, attributes])

This method is used with the variable of the WINDOW data type and displays a text to the reserved error line of the window associated with the variable.

Usage

This method accepts two arguments. The first argument is a quoted string or a variable of a character data type which is to be displayed to the error line. The *attributes* argument is a quoted string containing the standard display attributes of colour and intensity, it is optional.

There may be more than one attribute in the character string, and they should be separated by commas. Here are the attributes you can use:

- Black, white, yellow, green, magenta, red, blue, cyan
- Underline, reverse, blink
- Normal, dim, bold

An example of the method call:

```
CALL win_var.Error("The error message", "red, reverse")
```

message()

`variable.Message("message" [, attributes])`

This method is used with a variable of the WINDOW data type and displays a text to the reserved message line of the window associated with the variable.

Usage

This method accepts two arguments. The first argument is a quoted string or a variable of a character data type which is to be displayed to the message line. The *attributes* argument is a quoted string containing the standard display attributes of colour and intensity, it is optional.

There may be more than one attribute in the character string, and they should be separated by commas. Here are the attributes you can use:

- Black, white, yellow, green, magenta, red, blue, cyan
- Underline, reverse, blink
- Normal, dim, bold

An example of the method call:

```
CALL win_var.Message("This is the message text")
```

move()

variable.Move(y, x)

This method is used with the variable of the WINDOW data type and moves the window associated with the variable to the new position specified by its arguments. The window needs to be opened with the help of the Open() method previously.

Usage

This method accepts two arguments. *y* stands for the line and *x* stands for the column to which the window will be moved, these are both literal integers or integer variables.

An example of the method call:

```
CALL win_var.Move(8,15)
```

open()

variable.Open("window name", y, x, rows, columns [, attributes])

This method is used with a variable of the WINDOW data type to initialize it and to open a window. The window opened in such a way can be referenced by other methods used for windows, if they are used with the same WINDOW variable.

Usage

This method accepts six arguments. The first argument is a quoted string or a variable of a character data type which represents the window name which will be displayed to the window titlebar. *Y* stands for the row and *x* for the column at which the upper left corner of the window will appear; these are both literal integers or integer variables. *Rows* stands for the window height and *columns* for the window width, and are represented by literal integers or integer variables. The *attributes* argument is a quoted string containing the standard display attributes of colour and intensity, the BORDER attribute and the attributes setting the positions of the reserved lines.

There may be more than one attribute in the character string, and they should be separated by commas. Here are the attributes you can use:

- Black, white, yellow, green, magenta, red, blue, cyan
- Invisible, underline, reverse, blink
- Normal, dim, bold
- Border
- form-line:<integer>, error-line:<integer>, message-line:<integer>, menu-line:<integer>, prompt-line:<integer>

The integer here stands for the number of the row where you want to position the reserved line. If the position for the reserved line is greater than the window height, the prompt will appear at the last line of the window

An example of the method call:

```
CALL win_var.Open("Window Name", 2, 5, 13, 35,  
                  "green, border, message-line:12, prompt-line:10")
```

openWithForm()

`variable.OpenWithForm("window name", form, y, x [, attributes])`

This method is used with a variable of the WINDOW data type to initialize it and to open a window with the specified form in it. The window opened in such a way can be referenced by other methods for window manipulation, if they are used with the same WINDOW variable.

Usage

This method accepts five arguments. The first argument is a quoted string or a variable of a character data type representing the window name which will be displayed to the window titlebar. The *form* argument is either a quoted character string or a variable of a character data type containing the name of the form file without the file extension, but with the relative path, if necessary. *Y* stands for the row and *x* for the column at which the upper left corner of the window will appear; these are both literal integers or integer variables. The *attributes* argument is a quoted string containing the standard display attributes of colour and intensity, the BORDER attribute and the attributes setting the positions of the reserved lines.

There may be more than one attribute in the character string, they should be separated by commas. Here are the attributes you can use:

- Black, white, yellow, green, magenta, red, blue, cyan
- Invisible, underline, reverse, blink
- Normal, dim, bold
- Border
- form-line:<integer>, error-line:<integer>, message-line:<integer>, menu-line:<integer>, prompt-line:<integer>

The integer here stands for the number of the row where you want to position the reserved line. If the position for the reserved line is greater than the window height, the prompt will appear at the last line of the window

An example of the method call:

```
CALL win_var.OpenWithForm("Window Name", "my_formfile", 3, 5,  
                           "border, form-line:1")
```


raise()

This method makes the window of the referenced WINDOW variable the current window and is analogous to the CURRENT WINDOW IS statement. It brings the window to the top of the window stack and requires no arguments.

An example of the method call:

```
CALL win_var.Raise()
```

ui.WINDOW Class

Window class is an interface class (the one belonging to ui namespace) which is designed to provide interface to windows objects.

To use an object of the Window class, you must first declare it with the DEFINE statement:

```
DEFINE wo ui.Window
```

Initializing a ui.WINDOW Object

The OPEN WINDOW statement is used to create a window and give a static name to it:

```
OPEN WINDOW orders WITH FORM "orders"
```

When a window is opened, you can bind it with a Window object. There are two methods which allow to do this: ForName() method and GetCurrent() method.

ui.Window.ForName()

The ForName() method is used to bind a window object to a window using the identifier specified in the OPEN WINDOW statement. It requires a single parameter that is the name of the window opened using the OPEN WINDOW statement. The method is used in combination with the LET statement and needs namespace specification:

```
DEFINE wo ui.Window
OPEN WINDOW orders AT 2,2 WITH FORM "f_orders"
LET wo = ui.Window.ForName("orders")
```

ui.Window.GetCurrent()

The GetCurrent() method is used to bind the current window to a Window object:

```
DEFINE wo ui.Window
LET wo = ui.Window.GetCurrent("orders")
```

Using a ui.WINDOW Object

The initialized object variable can be used with the help of the methods described below.

GetForm()

The GetForm() method is used to get a ui.Form instance belonging to the current window. This provides the user with the possibility to manipulate form objects and returns a variable of the ui.Form type.

```
DEFINE      wo ui.Window,
            fo ui.Form
OPEN WINDOW w1 AT 2,2 WITH FORM "my_form"
LET wo = ui.Window.ForName("w1")
LET fo = wo.GetForm()
```

CreateForm()

this method is used to create an empty form. This form is then displayed in the associated window. If the window had a form before that, the old form is replaced with the new empty one. The methods accepts a single parameter of a character data type that is the name of the created form. It returns a variable of the ui.Form type. If the form name set as a parameter is already used for another form in this application, the method returns NULL and does not create anything.

```
DEFINE      wo ui.Window,
           fo ui.Form
OPEN WINDOW w1 AT 2,2 WITH 10 ROWS, 50 COLUMNS
LET wo = ui.Window.ForName("w1")
LET fo = wo.SetForm("newform")
```

SetText()

The SetText() method is used to specify the window title. It requires a single argument of the character data type which represents the new title does not return anything.

```
CALL wo.SetText("My Window")
```

GetText()

The GetText() method is used to retrieve the name of the window. It does not requires any parameters and returns a character string containing the window title.

```
LET title = wo.GetText()
```

SetImage()

The SetImage() method is used to specify the icon for the window title bar. It accepts a single parameter of a character data type containing the image file name and path.

GetImage()

The GetImage() method is used to retrieve the icon of the window title bar. It accept no parameters and returns the image file name and path.

Example

Here is an example of a ui.Window object manipulation: a window is got by name and its title is changed:

```
MAIN
  DEFINE wo ui.Window
  OPEN WINDOW orders_w WITH FORM "orders" ATTRIBUTE(BORDER)
  LET wo = ui.Window.forName("orders_w")
  CALL wo.setText("Order list")
  MENU "mmenu"
    COMMAND "Back" EXIT MENU
  END MENU
  CLOSE WINDOW orders_w
END MAIN
```

Form

This section includes the standard 4GL built-in functions which handle forms implicitly without referring to them like to objects. It also includes the methods which are used with variables of the FORM data type and which treat forms as objects. FORM variables are declared in the same way as simple data types; the ways to use methods are described [above](#). The methods make the form manipulation more flexible and allow of the use of different scopes of reference for screen forms.

fgl_form_close()

Querix4GL provides the useful functionality of dynamic form functions. These are functions that replicate the functionality of form statements.

In order to have dynamic forms, Querix4GL has a dynamic function equivalent for every form statement. When declaring form definitions, namely OPEN, DISPLAY and CLOSE, it is necessary to specify the argument, such as the form name. With dynamic form functions these names can be a variable.

Usage

Dynamic form functions provide the same functionality as form definitions, except that they can use variables instead of specific names.

An example of dynamic function equivalent of the form CLOSE definition follows:

To close a form in 4GL you have the following code:

```
CLOSE form1
```

which would have, as an equivalent:

```
LET form_name = "form1"  
CALL fgl_form_close(form_name)
```

fgl_form_display()

Lycia provides the useful functionality of dynamic form functions. These are functions that replicate the functionality of form statements.

In order to have dynamic forms, Querix4GL has a dynamic function equivalent for every form statement. When declaring form definitions, namely OPEN, DISPLAY and CLOSE, it is necessary to specify the argument, such as the form name. With dynamic form functions these names can be a variable.

Usage

Dynamic form functions provide the same functionality as form definitions, except that they can use variables instead of specific names.

An example of dynamic function equivalent of the form DISPLAY definition follows:

To display a form in 4GL you have the following code:

```
DISPLAY form1
```

Which would have, as an equivalent:

```
LET form_name = "form1"  
fgl_form_display(form_name)
```

fgl_form_open()

Lycia provides the useful functionality of dynamic form functions. These are functions that replicate the functionality of form statements.

In order to have dynamic forms, Querix4GL has a dynamic function equivalent for every form statement. When declaring form definitions, namely OPEN, DISPLAY and CLOSE, it is necessary to specify the argument, such as the form name. With dynamic form functions these names can be a variable.

Usage

Dynamic form functions provide the same functionality as form definitions, except that they can use variables instead of specific names.

An example of dynamic function equivalent of the form OPEN definition follows:

To open a form in 4GL you have the following code:

```
OPEN FORM form1 FROM "filename"
```

which would have, as an equivalent:

```
LET form_name = "form1"  
fgl_form_open(form_name, "filename")
```

fgl_grid_header()

You can dynamically alter the grid headings using the `fgl_grid_header()` function. This function will apply changes to the named grid in the current form only. `fgl_grid_header()` can also be used to change the alignment of the header text and also to dynamically change the key associated with a mouse click on a particular header.

Usage

This function takes between three and five parameters – `grid_name`, `field_name`, `label`, `alignment` and `keyname`. It is used to change the label associated with the grid and field passed to it and either or both of its alignment and keypress. This keypress is the action that occurs when the grid header is clicked on.

The optional alignment parameter can be one of 'left', 'right' or 'center'.

The optional `key_name` parameter can be any valid 4GL key.

The following is an example of dynamically changing a grid column header.

```
CALL fgl_grid_header("sc_grid","f1","A new header label")
```

The next example shows the dynamic changing of a grid column header, its alignment and also the key press that occurs when the column header is clicked on. In this case clicking on the column header would generate a F32 key. A 4GL programmer could use this key press to perform an operation on the data, for example a sort on that particular column.

```
CALL fgl_grid_header("sc_grid","f1","This is my label","left", "F32")
```

These settings will apply to a grid in the current form only. If you do not specify the keypress, any existing keypress associated with the column will be cleared.

fgl_scr_size()

The `fgl_scr_size()` function accepts as its argument the name of a screen array in the currently opened form and returns an integer that corresponds to the number of screen records in that screen array.

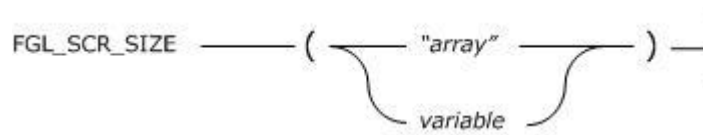


Figure 35 - fgl_scr_size() Function

`fgl_scr_size` takes an array as its argument, which must be placed between double speech marks. The *array* argument is an identifier for the screen array, which will be used in the INSTRUCTIONS section of the form that is current. It may, as an alternative, take a CHAR or VARCHAR variable, which contains the array identifier.

Usage

The built-in `fgl_scr_size()` function returns the declared size of a specified screen array at runtime. In the following example, a form specification file declares two screen arrays, called **s_rec1** and **s_rec2**:

Code Sample: fgl_scr_size_function.per

```
DATABASE FORMONLY
```

```
SCREEN
{
  [f1 ] [f2 ]
  [f1 ] [f2 ]
  [f1 ] [f2 ]
  [f3 ] [f4 ]
  [f3 ] [f4 ]
  [f5 ]
}
```



```
ATTRIBUTES
f1=FORMONLY.a;
f2=FORMONLY.b;
f3=FORMONLY.c;
f4=FORMONLY.d;
f5=FORMONLY.e;

INSTRUCTIONS
SCREEN RECORD s_rec1[3]  (
    FORMONLY.a,
    FORMONLY.b
)
SCREEN RECORD s_rec2 (
    FORMONLY.c,
    FORMONLY.d
)

DELIMITERS "[""]"
```

Code Sample: fgl_scr_size_function.4gl

```
MAIN
    DEFINE y,x INT
    DEFINE ch CHAR(10)

    OPEN WINDOW w1 AT 2,3 WITH FORM "fgl_scr_size_function" ATTRIBUTE (BORDER)

    CALL fgl_scr_size("s_rec1") RETURNING y
    CALL show_size(y,2)

    LET y = fgl_scr_size("s_rec1") -- Can also be called
    CALL show_size(y,3)

    -- in a LET statement
    CALL show_size(y,4)
    LET ch = "s_rec2"
    LET y = fgl_scr_size(ch)

    CALL show_size(y,5)

    LET y = fgl_scr_size(ch)

    -- in a LET statement
    CALL show_size(y,6)

    CLOSE WINDOW w1

END MAIN
```

```
FUNCTION show_size(y,row)
  DEFINE x, y INTEGER
  DEFINE row INTEGER

  DISPLAY "Columns = ", x AT row, 2
  SLEEP 2

END FUNCTION
```

The proper value is returned even though the array dimension is not specified in the form file.

An error is returned if no form is open or if the specified array is not in the form currently open.

References

ARR_CURR(), ARR_COUNT()

close()

variable.Close()

The Close() method is used with a variable of the FORM data type which has already been used with the Open() method. It closes the form described by the variable and frees the resources allocated to it. Once the form is closed, the corresponding FORM variable can be used to open and manipulate another form.

Usage

This method takes no arguments. It does not return anything and closes the form described by the form variable.

```
CALL form_v.Close()
```

display()

`variable.Display([attributes])`

This method is used with a FORM variable to display a previously opened form. Thus the OPEN() method needs to be executed prior to it. It displays the form to the current window.

Usage

This method accepts one optional parameter which is the display attributes of the form to be displayed. You need to specify the attributes as a quoted character string, they are case insensitive.

```
CALL form_v.Display("yellow, bold")
```

getWidth()

`variable.GetWidth()`

This method is used with a variable of the FORM data type to retrieve the width of the form in columns. It takes no parameters and returns the information about the form associated with the variable.

Usage

To be able to use this method, you need to use the same variable with the `Open()` method first. It returns an integer specifying the number of columns in the form SCREEN section.

```
CALL form_v.GetWidth()
```

getHeight()

variable.GetHeight()

This method is used with a variable of the FORM data type to retrieve the height of the form in rows. It takes no parameters and returns the information about the form associated with the variable.

Usage

To be able to use this method, you need to use the same variable with the Open() method first. It returns an integer specifying the number of rows in the form SCREEN section.

```
CALL form_v.GetHeight()
```

open()

variable.Open("form file" [, "form alias"])

This method is used with a variable of the FORM data type to open a form from a form file. It initializes the FORM variable and allows the programmer to use other methods to manipulate this form.

Usage

This method takes two parameters. The first parameter is the filename - the name of the form file without the extension but with a relative path, if needed. It can be either a quoted character string or a character variable. The second parameter is the form alias which is also either a character string or a character variable representing the form name, and is optional. It is used to reference the form in a script file (.qxs file). If the alias is not provided, the form can be referenced based on the name of the file without the file extension.

```
CALL form_v.Open("form_file002", "my_form")
```

ui.FORM Class

The Form class is a built-in class that belongs to ui namespace. The class is designed to provide an interface for manipulating forms at the program runtime. The objects of the Form class can be used to manipulate forms opened by OPEN FORM and OPEN WINDOW WITH FORM statements.



Note: the OPEN FORM statement does not create a un.Form object. It is created by the following DISPLAY FORM statement and is attached to the window in which the form is displayed.

To use Form class objects, you should first declare them with the DEFINE statement and then instantiate by means of the [GetForm\(\)](#) method of the [ui.WINDOW](#) object. As it was mentioned above, the form object is associated with the window object of the window where the form is displayed.

LoadActionDefaults ()

LoadActionDefaults() method is used to load the actions settings from a toolbar specified in the form file given as the method parameter. If the form includes objects besides the toolbar, these objects are ignored.

LoadToolbar()

It is possible to use a form file to store a toolbar which one can apply to other form files and windows.

To create a toolbar that can be added to other program objects, create a form which will contain only a toolbar and no other form objects besides the default ones.

The third-party files containing the toolbar definition are also converted into the .fm2 format.

The LoadToolbar() method is used to load a toolbar for the specified ui.Form object:

```
CALL MyForm.LoadToolbar("mytoolbar_form")
```

It is advised that you don't use the file extension name in the toolbar form specification. In this case, the runtime system will automatically add it. However, you still can specify the toolbar form as "mytoolbar_form.fm2"

The toolbar added this way will also be used in MDI containers.

LoadTopMenu()

It is possible to use a form file to store a Top Menu which one can apply to other form files and windows.

To have a Top Menu that can be added to other program objects, you should create a form containing only a Top Menu and no other form objects besides the default ones.

The third-party files containing the Top Menu definition are also converted into the .fm2 format.

The LoadTopMenu() method is used to load a Top Menu for the specified ui.Form object:

```
CALL MyForm.LoadTopMenu("mytopmenu_form")
```

If there is a Top Menu already specified for the target form, this Top menu is replaced by the one loaded from the .fm2 file.

It is advised that you don't use the file extension name in the Top Menu form specification. In this case, the runtime system will automatically add it. However, you still can specify the toolbar form as "mytopmenu_form.fm2"

SetDefaultInitializer()

The SetDefaultInitializer() method is used to specify a default initialization function which will be used to run global processing of all forms opened by a program. This method accepts a function name as a parameter. It is a class method, thus it is called with the ui.Form method preceding the function call.

The initializer function referenced by the method must have a single parameter which should be of the ui.Form data type. Each time an object of ui.Form class is initialized in the program (e.g. using getForm() method of the ui.Window class), the initialization function will be called and the new ui.Form object will be passed to the function as the parameter. Any form modifications made by the initialization function will affect the form associated with the ui.Form object passed as the parameter.

The setDefaultInitializer() method takes effect from the moment of its execution onwards. It has no effects on the form objects that were initialized before it was called.

```
CALL ui.Form.SetDefaultInitializer("initform")
...
FUNCTION initform(f)
  DEFINE f.ui.Form
    ...
END FUNCTION
```



Note: the initialization function name must be specified in lower-case characters.

SetElementText()

The SetElementText() method is used to change dynamically static form widgets. It changes the value of the TEXT attribute of a widget. The accepts two arguments: the name of the form widget and the new value for the TEXT attribute in a form of a character string. For example you can change the text displayed by a static label in this way.

```
DEFINE wo ui.Window,  
      fo ui.Form,  
      a string  
  
OPEN WINDOW w1 AT 2,2 WITH FORM "my_form"  
  
LET wo = ui.window.getCurrent()  
LET fo = wo.getForm()  
  
CALL fo.setElementText("main_label", "New Label Text")
```

The SetFieldText() method has a synonym - SetElementText () - that is used for compatibility purpose.

SetElementImage()

The SetElementImage() method is used to specify a new image for the IMAGE attribute of a static form widget. The accepts two arguments: the name of a static form widget and the new value for the IMAGE attribute that is a file name with the path.

```
CALL fo.setElementImage ("main_label", "/images/my_image.jpg")
```

The SetFieldImage() method has a synonym - SetElementImage() - that is used for compatibility purpose.

SetFieldStyle()

The SetElementStyle() method is used to modify the display style of a static form widget. The accepts two arguments: the first one is the name of a static form widget and the second one is the name of a class in a theme file (.qxtheme).

A class in a theme file is created using the With Class filter in the Lycia Theme Designer. The properties set for the specified class will then be applied to the selected widget, even if this class was not applied to this widget directly in the theme file. The function of this method is the same as the function of the Apply Class property of the Theme Designer used for the same widget, but the method can apply classes dynamically. For the detailed information about the Theme Designer and theme files see the "Graphical Clients Reference" guide.

```
CALL fo.setElementStyle("fonronly.main_label", "my_style_class")
```

The SetFieldStyle() method has a synonym - SetElementStyle() - that is used for compatibility purpose.

EnsureFieldVisible()

The EnsureFieldVisible() method is used to make certain that the user can see the specified form field.

You can specify the form field name only or use a prefix as well ("column" or "table.column").

However, the method does not actually focus on the specified field. The elements displayed by this method are not fixed on the screen, they can disappear when the user performs the next action. For

example, a form can contain two pages, and you can use the `EnsureFieldVisible()` method to make the one which is not currently used visible when the user presses some key (e.g., TAB)

If you want to activate a field on a tab that is not displayed, use NEXT FIELD option rather than `EndureFieldVisible()` method.

The `EnsureElementVisible()` is a synonym for this method used for compatibility purposes.

SetFieldHidden()

The `SetFieldHidden()` method is used to specify the value of the `VISIBLE` field attribute of the field or fields with the given name. It has basically the same effect as the `Visible` property of a theme file. For the detailed information about the Theme Designer and theme files see the "Graphical Clients Reference" guide.

It accepts two parameters: first is the name of the field and the second is the value of the `VISIBLE` attribute. The value of the `VISIBLE` attribute can be either 0 (then the field will be visible) or 1 (then the field will be hidden)

Note, that at least one field should remain visible, otherwise, the current user interaction process will stop.

The `EnsureElementHidden()` is a synonym for this method used for compatibility purposes.

Example

Here is an example of a Form object manipulation:

```
MAIN
  DEFINE wo ui.Window
  DEFINE fo ui.Form
  DEFINE prod_id INTEGER
  DEFINE article CHAR(20)
  OPEN WINDOW orders WITH FORM "ord_list"
  LET wo = ui.Window.getCurrent()
  LET fo = wo.getForm()
  INPUT BY NAME prod_id, article
    ON ACTION ("hide")
      CALL fo.setElementHidden("idlabel",1)
      CALL fo.setFieldHidden("products.prod_id",1)
    ON ACTION ("show")
      CALL fo.setElementHidden("idlabel",0)
      CALL fo.setFieldHidden("products.prod_id",0)
  END INPUT
END MAIN
```

ui.DRAGDROP Class

The DragDrop class is a built-in class that belongs to the ui namespace. The class is designed to control the events caused by Drag and Drop actions.

The Drag and Drop procedure is configured by the ui.DragDrop variable which is used as a parameter by the Drag and Drop dialog control blocks. The variable should be previously declared by the DEFINE statement.

SetOperation()

The dragged objects can be copied or moved to the new destination. The SetOperation() method is used in the ON DRAG_START clause to specify the default Drag and Drop action.

The SetOperation() method does not return anything and can pass one of the three arguments given in the table below:

Parameter	Description
Copy	Makes Drag and Drop actions copy the source object
Move	Makes Drag and Drop actions move the source object
NULL	Makes the program cancel/deny the Drag and Drop process

You can call the SetOperation() method within any of the Drag and Drop triggers. It is commonly used in ON DRAG_START block to specify the result of the Drag and Drop action, as well as in ON DRAG_ENTER and ON DRAG_OVER blocks to cancel Drag and Drop events, when the dragged object format does not correspond to the format of the target place. When the method is used in ON DRAG_ENTER block, it forces a defined Drag and Drop operation to be performed.

GetOperation ()

The dragged objects can be copied or moved to the new destination. The GetOperation() method is used to return the type of the current Drag and Drop operation. The method returns a string with the identifier of the current operation ("copy", "move"), or NULL if the operation was denied.

The program can use the returned value to change the data model. For example, when a row is dropped to a source list in a "copy" mode, the original row will be kept, while it will be removed if the drag and drop operation is performed in the "move" mode.

The GetOperation() method is mostly used within the ON DRAG_FINISHED block. The method needs no arguments.

AddPossibleOperation ()

The AddPossibleOperation() method is used to specify the additional operations for the Drag and Drop event.

The method needs a string argument specifying the operation type. The possible argument values are given in the table below:

Parameter	Description
Copy	Makes Drag and Drop actions copy the source object
Move	Makes Drag and Drop actions move the source object
NULL	Makes the program cancel/deny the Drag and Drop process

GetLocationRow ()

The GetLocationRow() method is used to return an integer value identifying the index of the target list row, to which the cursor was placed during the drag and Drop operation. You can use the method in the ON DROP block when you need to get the index of the target row and modify or replace it by the dragged object.

You can also use the method within the ON DRAG_OVER block to cancel the drop, when the value returned by GetLocationRow() does not meet special conditions.

The method does not need any arguments.

GetLocationParent ()

When a dragged object is the one from the TreeView, it can be dropped to the destination object as a sibling or a child node. The GetLocationParent() method is used to distinguish between these two options. The method returns an integer value identifying the index of the target node parent. The index of the target node itself is returned by the GetLocationRow() method. If the indices returned by the both methods are the same, the dropped object should be appended to the target node as a child.

When the value returned by GetLocationParent() identifies the parent node to where the dragged object is to be added as a child; the value returned by GetLocationRow() method specifies the sibling node. If these values are different, the dropped node should be inserted before the node, found by the GetLocationRow() method.

These methods can be used in ON DROP and in ON DRAG_OVER blocks to cancel the drop in case the returned indices do not meet special conditions.

SetFeedback ()

The SetFeedback() method is used to specify the appearance of the Drag and Drop target object. For example, you can specify different visual indicators for situations, when the mouse cursor is placed at the drag and drop target.

The SetFeedback() method does not return any values, but needs a string parameter to be passed. The list of the possible parameters and their descriptions is given in the table below:

Parameter	Description
All	The dropped object will be placed to some place on the target, the exact location is not important.
Insert	The dropped object will be inserted between the existing rows of the list.
Select	The dropped object will replace the row of the list that appears under the mouse.

SetMimeType ()

It is possible to use Drag and Drop not only to move data within one and the same 4GL application or between two 4GL applications running at once. It is also possible to move the dragged objects from and to an external application.

When the Drag and Drop is performed between different applications, it is necessary to identify the type of data passed to the Drag and Drop buffer in order to provide the proper data manipulation. The type of the data stored in Drag and Drop buffer is defined by the MIME (Multiple Internet Mail Extensions) type, a widely used internet standard specification.

In 4GL applications, only text data can be involved in Drag and Drop operations. The MIME type check is intended to see whether the format of the dragged object allows it to be inserted to the drop target. There are some standard MIME types, such as:

- text/plain
- text/uri-list
- text/x-vcard

The SetMimeType() method is used to set the MIME type that will be used during the Drag and Drop operations performance. The method does not return any values and needs a string value specifying the new MIME type as an argument. The method is usually called within the ON DRAG_START control block and is used together with the SetBuffer() method.

If the MIME type is not specified before the start of the Drag operation, the text/plain type is taken as the default one, and the selected rows will automatically be passed to Drag and Drop buffer. If you want to prohibit drag and drop between applications, you should specify a MIME type unique for the current application so that the other applications do not recognize it. Note that the MIME type defined this way should not be in conflict with standard MIME types:

- text/file-to-copy
- text/accounting-info

SetBuffer ()

The SetBuffer() method is used to pass the text data of a dragged object from the 4GL application to an external one. The method does not return any values and needs a string argument.

The SetBuffer() method is commonly used in the ON DRAG_START block together with the SetMimeType() method.

By default, the dialog treats the SetBuffer() argument string as a tab-separated list of values (according to the text/plain MIME type).

SelectMimeType ()

The SelectMimeType() method is used to get sure that the data is available in the format specified by the MIME type. The MIME type is passed to the method as a string value and the program checks whether the data in the Drag and Drop buffer are stored using this type.

The method returns TRUE, if the type of the data is available in the buffer. If the method returns TRUE, you can use the GetBuffer() method to return the data from the buffer.

The `SelectMimeType()` method is commonly used in `ON DRAG_ENTER` and `ON DRAG_OVER` clauses to cancel the drag and drop operation in cases when there are no supported MIME types in the buffer. Later, in the `ON DROP` block, you can check which MIME type was selected previously using the `GetSelectedMimeType()` method, get the data from the Drag and Drop buffer using the `GetBuffer()` method and then insert the new rows to the target fields.

GetSelectedMimeType ()

The `GetSelectedMimeType()` method is used to return a string value identifying the type of the MIME previously selected by the `SelectMimeType()` method. The method needs no arguments.

The method is typically placed in the `ON DROP` block and should be used before you retrieve the Drag and Drop buffer data with the `getBuffer()` method.

GetBuffer ()

The `GetBuffer()` method is used to retrieve the data from the Drag and Drop buffer. You have to identify the current MYME type with the `GetSelectedMimeType()` method before you use `GetBuffer()`. The `GetBuffer()` method returns a string value and does not need any arguments.

You should remember that Drag and Drop data can be reached only during the `ON DROP` block execution, therefore, the `GetBuffer()` method can be used only in this block.

Here is an example of the MIME type manipulation functions usage:

```
DISPLAY ARRAY pr_arr TO scr_arr.* ...
...
  ON DRAG_ENTER(dd)
    -- If the MIME type is unavailable, set operation to NULL
    CASE
      WHEN dd.selectMimeType("text/plain")
      WHEN dd.selectMimeType("text/uri-list")
      OTHERWISE
        CALL dd.setOperation(NULL)
    END CASE
  ...
  ON DROP(dd)
    -- Select MIME type and retrieve data from buffer
    LET s_row = dd.getLocationRow()
    CALL DIALOG.dropped_row("scr_arr", s_row)
    IF dn.getSelectedMimeType() == "text/plain" THEN
      LET pr_arr[s_row].dragegdtext = dd.getBuffer()
    END IF
  ...
END DISPLAY
```

DropInternal ()

The `DropInternal()` method is used within the `ON DROP` block to allow dragging and dropping within one and the same object to perform all the necessary changes within the array rows and move the row selection together with the corresponding cell attributes.

Drag and Drop events performed in tree views require much coding to manipulate parent-child relations: the dropped node can be inserted under parent to different levels of hierarchy and in different places among the siblings. However, you can also use the `DropInternal()` method with regular tables.

The `DropInternal()` method will be ignored without any error messages, if the element is dragged from one object and dropped to another one as well as in case it is called outside the ON DROP block.

Field

fgl_buffertouched()

The fgl_buffertouched() function is called by a BEFORE or an AFTER {FIELD | INPUT | CONSTRUCT} instruction; it returns an integer value of TRUE or FALSE, which show whether or not the last field has been modified.

Usage

This function must be used within a dialog function, such as INPUT, INPUT ARRAY, DISPLAY ARRAY, or PROMPT statements. If it is not you may find errors occurring at compile or runtime.

In the following example, the source code checks the AFTER INPUT block, to find out if the user has modified the field contents; that is, whether data has been entered or changed in this block during input. If data is changed in the f1 field a dialog box will be displayed, showing that "You modified the field contents"; if no data is entered or changed, a message appears stating that "You did not modify the field".

```
MAIN
  DEFINE x CHAR(20)

  OPTIONS INPUT WRAP,
    ACCEPT KEY RETURN
  DEFER INTERRUPT

  OPEN WINDOW w_test
    AT 2, 2
    WITH FORM "fgl_buffertouched_function"

  LET x = "This is the original text"

  INPUT x WITHOUT DEFAULTS FROM f1
    AFTER INPUT
      IF NOT fgl_buffertouched() THEN
        CALL fgl_message_box("You did not modify the field")
        CONTINUE INPUT
      ELSE
        CALL fgl_message_box("You modified the field contents")
      END IF
    END INPUT

  CLOSE WINDOW w_test
END MAIN
```

Form File: fgl_buffertouched_function.per

```
DATABASE formonly
```

```
SCREEN
```

```
{
```

```
Please edit this form field
```

```
[f1                                ]
```

```
}
```

```
ATTRIBUTES
```

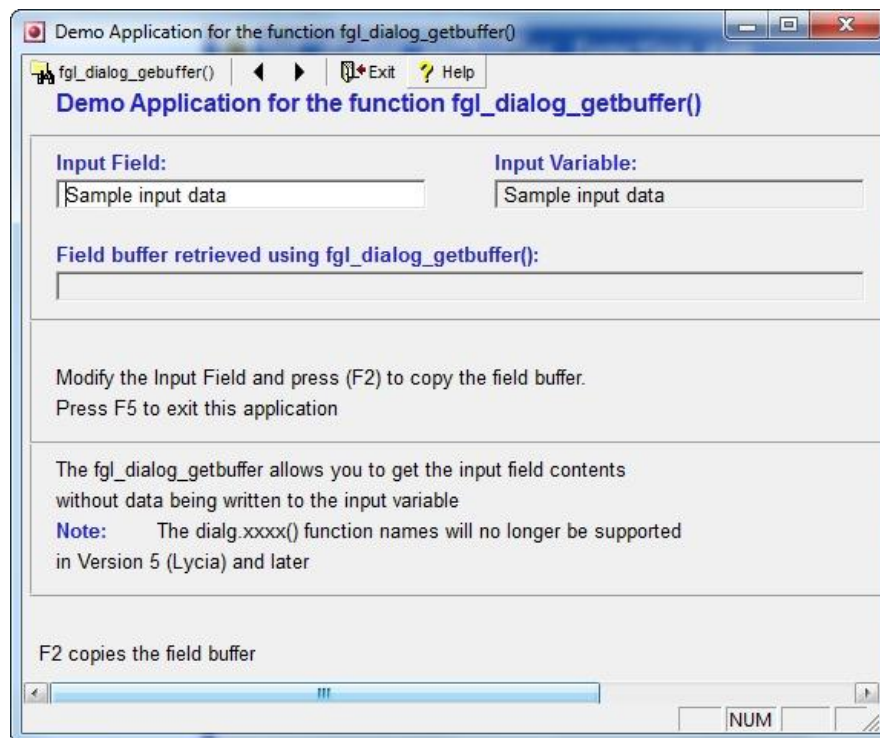
```
f1=formonly.f1;
```

```
INSTRUCTIONS
```

```
DELIMITERS "[ ]"
```

```
Project: functions
```

```
Program: fgl_dialog_getbuffer_function
```



fgl_dialog_infield()

The function `fgl_dialog_infield()` has been developed to test the position of the cursor. Specifically, this function tests whether the cursor is located within a particular field. This field is passed as the argument in the function call. It returns the Boolean value of TRUE if the cursor is in the desired field when the function is called and a FALSE value otherwise.

Usage

This function is used when you need to perform an action based on the cursor's current position.

For example, to test whether the cursor is in the 3rd field (f3):

```
IF fgl_dialog_infield("f3") THEN
  ERROR "CORRECT! You are in the third field!"
  CALL fgl_winmessage("Correct","Correct! You are in the third field","info")
ELSE
  ERROR "WRONG! You are not in the third field, try again!"
END IF
```

fgl_dialog_getbuffer()

The buffer is used to store the values of a field before they are written to the input variable. The contents are normally written to the variable when either an accept action is performed or the field is navigated away from. If the contents of the fields are required for any reason – without writing the values to the underlying input variable – then this function is used to get the contents of the buffer.

Usage

This function can be used with a dedicated *get* cursor or key; in this example the F2 keyboard key is used. After typing the required text into the Input Field press the F2 key and the function will write the value of the current field to Field Buffer field. In other words, the value returned by this function will be the value equal to the value shown in the key field when the function was invoked.

```
MAIN
  DEFINE x CHAR(20)
  DEFINE y CHAR(20)

  LET x = "Sample input data"

  OPEN WINDOW w_test
    AT 2, 2
    WITH FORM "fgl_dialog_getbuffer_function"

  OPTIONS INPUT WRAP

  INPUT x WITHOUT DEFAULTS FROM f1
    BEFORE INPUT
      DISPLAY x TO f2

  ON KEY (F2)
    LET y = fgl_dialog_getbuffer()
    DISPLAY y TO f3
    DISPLAY x TO f2
  END INPUT
END MAIN
```

References

fgl_dialog_getfieldname(), fgl_dialog_setbuffer(), fgl_buffertouched(), get_fldbuf().

fgl_dialog_getcursor()

The `fgl_dialog_getcursor()` function returns the position of the cursor, in the currently prompted field.

Usage

This function can be used with a dedicated *get cursor* button or key; in this code sample the F4 keyboard key is used. After typing the first few unique characters of the cursor name into the field of the dialog box, press the F4 key; the function will return the current position of the cursor to the message line.

```
MAIN
DEFINE text CHAR(512)
DEFINE pos INTEGER
OPEN WINDOW w1 AT 1,1 WITH FORM "fgl_dialog_getcursor_function"
INPUT BY NAME text
ON KEY (f4)
--# LET pos = fgl_dialog_getcursor()
--# MESSAGE" current position: ", pos
END INPUT
CLOSE WINDOW w1
END MAIN
```

fgl_dialog_getfieldname()

The function fgl_getfieldname() returns the name of the currently active field, i.e., the name of the field in which the cursor is currently positioned. It requires no argument passed to it.

Usage

The function returns a CHAR(n) datatype value that is the name of the currently active field.

```
MAIN
...
...
...

INPUT BY NAME rec1.* WITHOUT DEFAULTS HELP 100
  ON KEY (F2)
    CALL fgl_dialog_getfieldname()
      RETURNING txt
    LET msg = "Current field is: \"",txt CLIPPED,\""

    IF INFIELD(F1) THEN
      IF dialog_function_name = 1 THEN
        DISPLAY "Field name: ", dialog.getfieldname() AT 4,30
      ELSE
        DISPLAY "Field name: ", fgl_dialog_getfieldname() AT 4,30
      END IF
    END IF

    IF INFIELD(F2) THEN
      IF dialog_function_name = 1 THEN
        DISPLAY "Field name: ", dialog.getfieldname() AT 5,30
      ELSE
        DISPLAY "Field name: ", fgl_dialog_getfieldname() AT 5,30
      END IF
    END IF

    ...
    ...
    ...
  END MAIN
```




In case no input field is active the function returns a null value (as a zero-length string).

fgl_dialog_setbuffer()

The fgl_dialog_setbuffer() function takes a value or variable as its argument. That variable is the value to be set in the current field.

The function can also be written as dialog.setbuffer(), the result is the same.

This function has no return value.


	This function does not work in the BEFORE FIELD trigger.
	If this function is used in an INPUT statement, it modifies the input buffer for the current field; however, its input variable is only assigned on leaving the field.
	This function marks the flags for the current field and the dialog as 'touched'. If this function is called, both FIELD_TOUCHED() and FGL_BUFFERTOUCED() will return TRUE.

fgl_dialog_setcursor()

The `fgl_dialog_setcursor()` function sets the cursor at a defined position in the current field.

Usage

This function takes as its argument an INTEGER value which is the position at which the cursor should be placed within the current field.

	If you specify a cursor position greater than the length of the variable, the cursor will disappear.
---	--

fgl_dialog_setfieldorder ()

The fgl_dialog_setfieldorder function has the same effect as a 4GL OPTIONS statement. It determines the order in which values are placed in fields, as determined by an INPUT statement.

The function takes an integer as its argument, which is 0 for unconstrained, or 1 for constrained.

If the field order is unconstrained, the values in the INPUT statement will be placed in the fields in the logical order of the fields. If the order is constrained, the values will be placed in the fields in the order defined in The INPUT statement.

fgl_dialog_setselection()

The `fgl_dialog_setselection()` function is used to select a text string within the current field. The function can be invoked only from the control blocks operating with the current fields, not in AFTER FIELD, AFTER ROW, etc.

The syntax of the function invocation is as follows:

```
CALL fgl_dialog_setselection(start, end)
```

The *start* argument specifies the initial position of the edit cursor, the *end* argument specifies the position of the selection end.

Note, that the *end* argument can be more, less or equal to the *start*.

fgl_dialog_getselectionend()

The `fgl_dialog_getselectionend()` function is used to return the position of the last character of the string, selected within the current field.

The function returns an integer value. The function returns zero, if the whole text is selected.

The value returned by the `fgl_dialog_getselectionend()` function can be higher than that of the `fgl_dialog_getcursor()` function, if the user performed the selection from right to left.

fgl_dialog_update_data()

When entering information into a field, the data is kept in the 'field buffer' until the cursor is moved to another field or the Input statement is completed i.e., by pressing the Accept key. Only then the data stored in the field buffer will be written to the underlying variable. Sometimes, it is required to do some processing with the data from a field (field buffer) without terminating the input statement. For this situation, the function `fgl_dialog_update_data()` / `dialog.update_data()` can be called to write any data from the field buffers to the underlying variables.

Utilisation of the `fgl_dialog_update_data()` / `dialog.update_data()` function will allow the application developer to write data from a specific field to the input variable before saving all data on the entire page. Essentially this affords the ability to capture data in a particular field without capturing the data in every field on that page. In effect this script is the opposite of the Field Buffer Function, and can be utilised via implementation of either of the following syntax examples:

```
fgl_dialog_update_data()  
dialog.update_data()
```

Note: No arguments are required for the above function.

fgl_formfield_getoption()

The `fgl_formfield_getoption()` function allows you to retrieve information about the current field, during a dialog function:

```
fgl_formfield_getoption(option)
```

Where *option* is a CHAR(n) datatype variable which can be any one of the following:

- `x` The column position within the screen
- `y` The line position within the screen
- `length` The length, in character spaces, of the field

Usage

The two following code samples show information about the active input field as focus moves between the fields.

Code Sample: fgl_formfield_getoption1_function.exe

```
GLOBALS
  DEFINE my_record RECORD
    f1 CHAR(5),
    f2 CHAR(10),
    f3 SmallInt,
    f4 Integer
  END RECORD
END GLOBALS

MAIN
  DEFINE field_info SMALLINT
  DEFINE print_string CHAR(70)

  OPEN WINDOW win AT 3,1 with FORM "fgl_formfield_getoption_function"

  INPUT by name my_record.*
  BEFORE INPUT
    LET field_info = fgl_formfield_getoption("x")
    CALL fgl_winmessage("fgl_formfield_getoption", string_format("x",field_info),
"info")
    LET field_info = fgl_formfield_getoption("y")
    CALL fgl_winmessage("fgl_formfield_getoption", string_format("y",field_info),
"info")
    LET field_info = fgl_formfield_getoption("length")
    CALL fgl_winmessage("fgl_formfield_getoption",
string_format("length",field_info), "info")
  END INPUT
```

```
END MAIN
```

```
FUNCTION string_format(option,formfield_value)
  DEFINE option char(10)
  DEFINE formfield_value SMALLINT
  DEFINE return_value char(70)

  LET return_value = "fgl_formfield_getoption() with option " || option, "
returned: " || formfield_value
  RETURNING return_value

END FUNCTION
```

Code Sample: fgl_formfield_getoption2_function.exe

```
GLOBALS
DEFINE win_rec RECORD
  x, y, ht, wid, b, fl, ml, cl, msl, el, ik, dk, nk, pk, ak,
  hk, abk, iw, fo INTEGER
END RECORD
DEFINE field_rec RECORD
  fx, fy, fil INTEGER
END RECORD
DEFINE inp_rec RECORD
  f1 CHAR(10),
  f2 CHAR(15),
  f3 CHAR(20),
  f4 CHAR(25)
END RECORD
END GLOBALS

MAIN

DEFINE a CHAR(10)

CALL win_func()

INPUT BY NAME inp_rec.*
  BEFORE FIELD f1
    CALL update_field()
  BEFORE FIELD f2
    CALL update_field()
  BEFORE FIELD f3
    CALL update_field()
  BEFORE FIELD f4
    CALL update_field()
  AFTER FIELD f4
    NEXT FIELD f1
  ON KEY(F1)
    EXIT INPUT
END INPUT

END MAIN

FUNCTION win_func()
  OPEN WINDOW w_test
    AT 2, 2
    WITH FORM "fgl_window_getoption2_function"

  CALL update_window()
```

END FUNCTION

```
FUNCTION update_window()  
  LET win_rec.x = fgl_window_getoption("x")  
  LET win_rec.y = fgl_window_getoption("y")  
  LET win_rec.wid = fgl_window_getoption("width")  
  LET win_rec.ht = fgl_window_getoption("height")  
  LET win_rec.b = fgl_window_getoption("border")  
  LET win_rec.fl = fgl_window_getoption("formline")  
  LET win_rec.ml = fgl_window_getoption("menuline")  
  LET win_rec.cl = fgl_window_getoption("commentline")  
  LET win_rec.msl = fgl_window_getoption("messageline")  
  LET win_rec.el = fgl_window_getoption("errorline")  
  LET win_rec.ik = fgl_window_getoption("insertkey")  
  LET win_rec.dk = fgl_window_getoption("deletekey")  
  LET win_rec.nk = fgl_window_getoption("nextkey")  
  LET win_rec.pk = fgl_window_getoption("previouskey")  
  LET win_rec.ak = fgl_window_getoption("acceptkey")  
  LET win_rec.hk = fgl_window_getoption("helpkey")  
  LET win_rec.abk = fgl_window_getoption("abortkey")  
  LET win_rec.iw = fgl_window_getoption("inputwrap")  
  LET win_rec.fo = fgl_window_getoption("fieldorder")
```

DISPLAY BY NAME win_rec.*

END FUNCTION

```
FUNCTION update_field()  
  LET field_rec.fx = fgl_formfield_getoption("x")  
  LET field_rec.fy = fgl_formfield_getoption("y")  
  LET field_rec.fil = fgl_formfield_getoption("length")
```

DISPLAY BY NAME field_rec.*

END FUNCTION

fgl_formfield_getoption() Demo Application

◀ ▶ ⏏ Exit ? Help

fgl_formfield_getoption() Demo Application

Window:	Lines:	Keys:	
x: 2	formline: 1	insert: 3032	inputwrap: 1
y: 2	menuline: 1	delete: 3033	fieldorder: 1
ht: 22	commentline: 21	next: 3034	
wid: 73	errorline: 0	prev: 3035	
border: 0	messageline: 2	accept: 27	
		help: 3000	

Field Properties: @Press 'TAB' and read the field properties

X: 48	@Field1:	
Y: 14	@Field2:	
Length: 23	@Field3:	
	@Field4:	

Press 'Tab' to read field properties - F1 Help - F5 to exit

NUM

Figure 36 – Field Get Option Example

fgl_overwrite()

The function `fgl_overwrite()` allows the developer to be able to stop the contents of a field being highlighted when in edit mode. This can often lead to the first key press overwriting the current field contents; setting this script option to false will combat this issue, and mean that to edit the field contents the user would need to place the field in edit mode, i.e., select it, before it could be edited.

```
?..?..?..fgloverwrite: <Boolean>
```

field_touched()

The FIELD_TOUCHED() operator tests whether the user has changed any values in a specified field, or fields, of the current 4GL form.

This operator can only be used within CONSTRUCT, INPUT, and INPUT ARRAY statements.

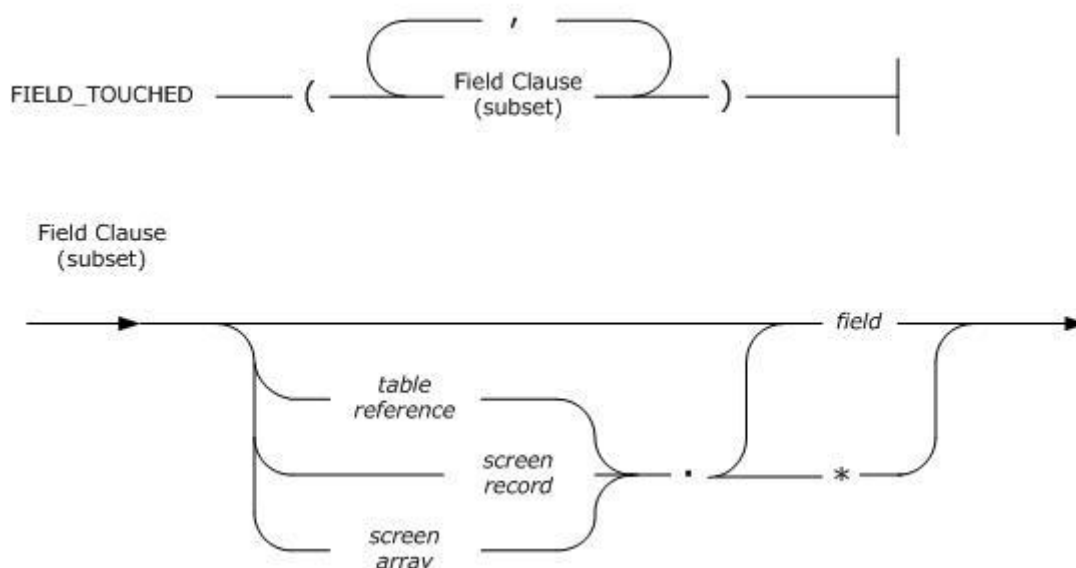


Figure 37 - FIELD_TOUCHED() Operator

In this syntax, *field* is the name of a screen field, which has been declared in the ATTRIBUTES section; *screen array* and *screen record* are the names that have been declared in the INSTRUCTIONS section; *table reference* is a table name, alias, or synonym, or the keyword FORMONLY, that has been declared in the TABLES section of the form specification.

Usage


This function can be used after a DISPLAY statement is used to display data in specified fields. If the user has changed any of the contents of a field, FIELD_TOUCHED() will return a value of TRUE.

The function can also return a TRUE value after a user has pressed any of the following keys, while the cursor is within the specified fields:

- Any printable character (including SPACEBAR)
- CONTROL-X (character delete)
- CONTROL-D (clear to end of field)

After any of these keystrokes, the FIELD_TOUCHED() operator returns TRUE, regardless of whether the keystroke actually changed the value in the field.

Otherwise, the `FIELD_TOUCHED()` operator returns `FALSE`. Moving through a field using any of the `RETURN`, `TAB`, or the arrow keys does not cause a field to be edited.

	<code>FIELD_TOUCHED()</code> is valid only in <code>CONSTRUCT</code> , <code>INPUT</code> , and <code>INPUT ARRAY</code> statements. When you use it, 4GL assumes that you are referring to the current screen record rather than to a different row of the screen array.
---	--

This operator does not register the effect of 4GL statements that appear in a `BEFORE CONSTRUCT` or `BEFORE INPUT` clause. You can assign values to fields in these clauses without marking the fields as touched.

In the following code sample, an `IF` statement is used to test whether the user has entered a value into any field. If no field has been touched, the program prompts the user to indicate whether to retrieve all customer records. If the user types `N` or `n`, the `CONTINUE CONSTRUCT` statement is executed, and the screen cursor is positioned in the form, giving the user another opportunity to enter selection criteria. If the user types any other key, the program terminates the `IF` statement and reaches the `END CONSTRUCT` keywords.

Code Sample: `field_touched_operator.4gl`

```
MAIN
  DEFINE rec1 RECORD
    f1 CHAR(10),
    f2 CHAR(10),
    f3 CHAR(10),
    f4 CHAR(10)
  END RECORD

  LET rec1.f1 = "Field f1"
  LET rec1.f2 = "Field f2"
  LET rec1.f3 = "Field f3"
  LET rec1.f4 = "Field f4"

  OPEN WINDOW w_test
    AT 2, 2
    WITH FORM "field_touched_operator"

    DISPLAY "Modify a field and press Return" AT 15,5

  INPUT BY NAME rec1.* WITHOUT DEFAULTS
  AFTER FIELD f1
    IF FIELD_TOUCHED(f1) THEN
      ERROR "You have altered field f1"
    END IF
  AFTER FIELD f2
```

```
        IF FIELD_TOUCHED(f2) THEN
            ERROR "You have altered field f2"
        END IF
    AFTER FIELD f3
        IF FIELD_TOUCHED(f3) THEN
            ERROR "You have altered field f3"
        END IF
    AFTER FIELD f4
        IF FIELD_TOUCHED(f4) THEN
            ERROR "You have altered field f4"
        END IF
    END INPUT

    CLOSE WINDOW w_test
END MAIN
```

This strategy is not as dependable as testing whether `query1 = " 1=1"` after the `END CONSTRUCT` keywords because the user might have left all the fields blank after first entering and then deleting query criteria in some field. In that case, the resulting Boolean expression (" `1=1`") can retrieve all rows, but `FIELD_TOUCHED()` returns `TRUE`, and the `PROMPT` statement is not executed.

Code Sample: `field_touched_operator.per`

```
DATABASE formonly

SCREEN
{
    Field f1: [f1      ]
    Field f2: [f2      ]
    Field f3: [f3      ]
    Field f4: [f4      ]
}

ATTRIBUTES
f1 = formonly.f1;
f2 = formonly.f2;
f3 = formonly.f3;
f4 = formonly.f4;

INSTRUCTIONS

DELIMITERS "[ ]"
```

References

Boolean Operators, `FGL_GETKEY`, `fgl_keyval()`, `fgl_lastkey()`, `GET_FLDBUF()`, `INFIELD()`

get_fldbuf()

The GET_FLDBUF() operator returns the character values of the contents of one or more fields in the currently active screen form.

This operator can only appear within the CONSTRUCT, INPUT, and INPUT ARRAY statements of 4GL.

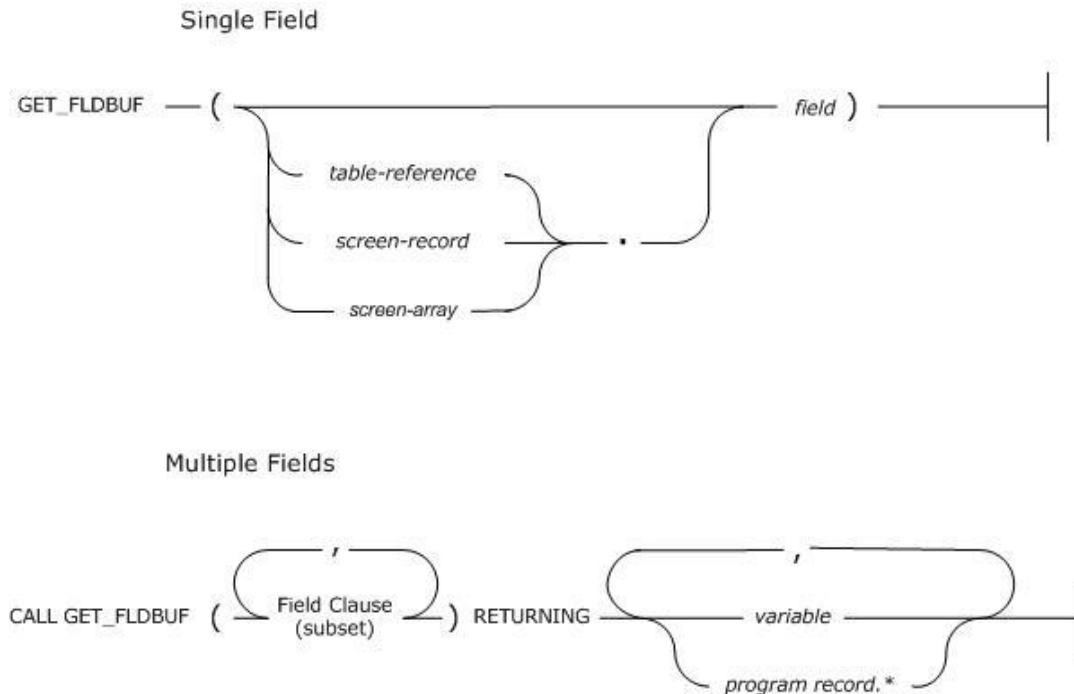


Figure 38 – GET_FLDBUF()

Element	Description
field	This is the name of a field in the current screen form.
program record	This is the name of a program record of CHAR and VARCHAR variables in which values from the specified fields can be stored.
screen-array	This is the name of a screen array that was defined in the INSTRUCTIONS section of the form specification file.
screen-record	This is the name of a screen record that is defined, either explicitly or by implication, in the form specification file.
table reference	This is the unqualified name, alias, or synonym of a table or view; or it is the keyword FORMONLY.
variable	This is a name, from a comma-separated list of character variables. The variables must be the same, and in the same order, as those specified in the field clause.

Usage

GET_FLDBUF() operates on a list of one or more fields. For example, you could use this LET statement to assign the value in the **cont_name** field to the variable **l_cont_name**:

```
LET l_cont_name = GET_FLDBUF(cont_name)
```

To specify a list of several field names as operands of GET_FLDBUF(), you must use the CALL statement with the RETURNING clause. Put commas between the each item in the list of field names and variables:

```
CALL GET_FLDBUF(cont_id, cont_comp, cont_name)  
RETURNING l_cont_id, l_cont_comp, l_cont_name
```

The following statement returns a set of character values which correspond to the contents of the **sc_cont** screen record and assigns these values to the **p_contact** program record:

```
CALL GET_FLDBUF(sc_cont.*) RETURNING p_contact.*
```

(The first asterisk (*) works as a wildcard that includes all the fields in the **sc_cont** screen-record; the second specifies all the members of the **p_contact** program record.)

You can use the GET_FLDBUF() operator to assist a user when entering a value in a field. For example, if you have an input field for last names, you can include an ON KEY clause that lets a user enter the first few characters of the desired last name. If the user calls the ON KEY clause, 4GL displays a list of last names that begin with the characters entered. The user can then choose a last name from the list.

The following program fragment demonstrates this use of the GET_FLDBUF() operator:

Sample Code: get_fldbbuf_operator.per

```
DATABASE formonly  
  
SCREEN  
{  
Input Field:           Input Variable:  
[f1                     ]   [f2                     ]  
  
Field buffer:  
[f3                     ]  
  
Press (F2) to copy the field buffer.  
\gp-----q \g  
\g| \gThe get_fldbbuf allows you to get the input field contents \g| \g  
\g| \gwithout data being written to the input variable           \g| \g  
\gb-----d \g  
  
}
```

ATTRIBUTES

f1=formonly.f1,comments="F2 copies the field buffer";

f2=formonly.f2;

f3=formonly.f3;

INSTRUCTIONS

DELIMITERS "[]"

Sample Code: get_fldbuf_operator.4gl

```
MAIN
  DEFINE x CHAR(20)
  DEFINE y CHAR(20)

  LET x = "Sample input data"

  OPEN WINDOW w_test
    AT 2, 2
    WITH FORM "get_fldbuf_operator"

  OPTIONS INPUT WRAP

  INPUT x WITHOUT DEFAULTS FROM f1
  BEFORE INPUT
    DISPLAY x TO f2

  ON KEY (F2)
    LET y = get_fldbuf(f1)
    DISPLAY y TO f3
    DISPLAY x TO f2

  END INPUT
END MAIN
```

If you assign the character string returned by the GET_FLDBUF() operator to a variable that is not defined as a character data type, 4GL tries to convert the string to the appropriate data type. Conversion is not possible in these cases:

- The field contains special characters (for example, date or currency characters) that 4GL cannot convert.
- GET_FLDBUF() is called from a CONSTRUCT statement, and the field contains comparison or range operators that 4GL cannot convert.

GET_FLDBUF() is valid only in CONSTRUCT, INPUT, and INPUT ARRAY statements. When 4GL finds this operator in an INPUT ARRAY statement, it assumes that the current row is being referenced. You cannot use a subscript within brackets to reference a different row of the screen array.

The following example uses the GET_FLDBUF() and FIELD_TOUCHED() operators in an AFTER FIELD clause in a CONSTRUCT statement. The FIELD_TOUCHED() operator checks whether the user has entered a value in the **cont_comp** field. If FIELD_TOUCHED() returns TRUE, GET_FLDBUF() retrieves the value entered in the field and assigns it to the **myint** program variable. A query is then performed to check that the company ID entered is valid.

```
CONSTRUCT BY NAME where clause ON contact.*
...
AFTER FIELD cont_comp
    IF FIELD_TOUCHED(cont_comp) THEN LET myint =\
GET_FLDBUF(cont_comp)

        SELECT COUNT(*)
            INTO I
            FROM company
            WHERE company.comp_id = myint
        IF  SQLCODE = NOTFOUND THEN
            ERROR "Company id is not valid"
        NEXT FIELD cont_comp
    END IF
END IF
```

References

FIELD_TOUCHED(), INFIELD()

infield()

The INFIELD() operator in CONSTRUCT, INPUT, and INPUT ARRAY statements tests whether its operand is the identifier of the current screen field.

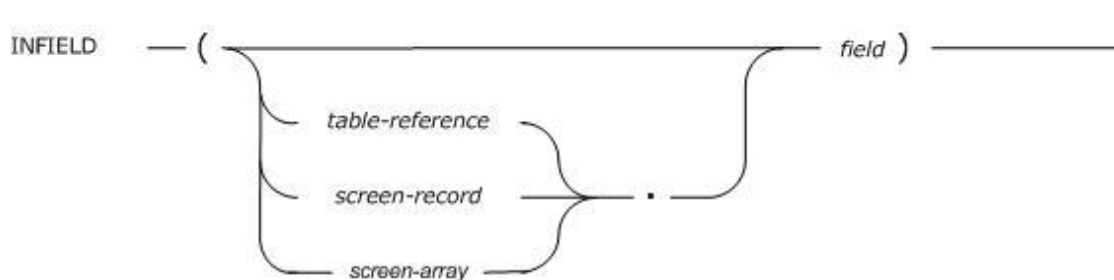



Figure 39 - INFIELD() Operator

Element	Description
<i>field</i>	This is the name of a field in the current screen form.
<i>screen-array</i>	This is the name of a screen array that was defined in the INSTRUCTIONS section of the form specification file.
<i>screen-record</i>	This is the name of a screen record that is defined, either explicitly or by implication, in the form specification file.
<i>table reference</i>	This is the unqualified name, alias, or synonym of a table or view; or it is the keyword FORMONLY.

Usage

INFIELD() is a Boolean operator that returns the value TRUE if *field* is the name of the current screen field. Otherwise, INFIELD() returns the value FALSE.

	You must specify a field name rather than a field tag as the operand.
---	---

You can use INFIELD() during a CONSTRUCT, INPUT, or INPUT ARRAY statement to take field-dependent actions.

The INFIELD() operator is typically part of an ON KEY clause, often with the built-in function SHOWHELP() to display help messages to the user.

Sample Code: infield_operator.per

```
DATABASE formonly

SCREEN
{
[f1      ]
[f2      ]
[f3      ]
[f4      ]

Press (F1) when you are in the third field.
}

ATTRIBUTES
f1 = formonly.f1, COMMENTS="This is the first field";
f2 = formonly.f2, COMMENTS="This is the second field";
f3 = formonly.f3, COMMENTS="This is the third field";
f4 = formonly.f4, COMMENTS="This is the fourth field";

INSTRUCTIONS

DELIMITERS "[" "]"

Project: functions
Program: fgl_dialog_infield_function
```

Sample Code: infield_operator.4gl

```
MAIN
  DEFINE input_rec RECORD
    f1 CHAR(10),
    f2 CHAR(10),
    f3 CHAR(10),
    f4 CHAR(10)
  END RECORD

  OPTIONS INPUT WRAP

  LET input_rec.f1 = "Field 1"
  LET input_rec.f2 = "Field 2"
  LET input_rec.f3 = "Field 3"
  LET input_rec.f4 = "Field 4"

  OPEN WINDOW w_test
    AT 2, 2
    WITH FORM "infield_operator"

  INPUT BY NAME input_rec.* WITHOUT DEFAULTS
    ON KEY (F1)
      IF INFIELD(f3) THEN
        ERROR "CORRECT! You are in the third field!"
      ELSE
        ERROR "WRONG! You are not in the third field, try again!"
      END IF
  END INPUT

  CLOSE WINDOW w_test
END MAIN
```

These source files display a form with four fields, of which the third is the 'correct' field in which to place your cursor. If you place your cursor in any of the other fields an error message is displayed, telling you your cursor is not in Field 3. The following screenshot shows the message displayed when you have your cursor in Field 3.

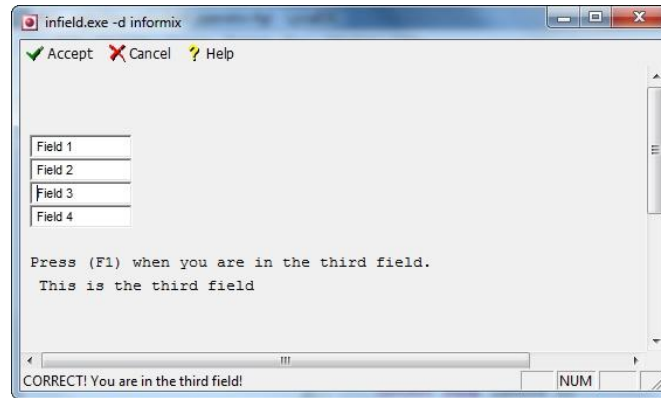


Figure 40 - Field Identifier Example

The next code example is from a program that uses `INFIELD()` to determine whether to call a function:

```
ON KEY (F5)
  IF INFIELD(ind_type) THEN
    CALL select_industry_type()
```

When a user presses F5 during the INPUT, the **`select_industry_type()`** function is invoked if the screen cursor is in the **`ind_type`** field.

In the following example, the INPUT statement uses the `INFIELD()` operator with the `SHOWHELP()` function to display field-dependent help messages.

```
INPUT l_contact_rec.* FROM sc_cont.*
  ON KEY(F1)
    CASE
      WHEN INFIELD(cont_name)
        CALL SHOWHELP(301)
      WHEN INFIELD(cont_comp)
        CALL SHOWHELP(302)
      WHEN INFIELD(ind_type)
        CALL SHOWHELP(303)
      ...
    END CASE
  END INPUT
```

References

`SCR_LINE()`, `SHOWHELP()`, `FIELD_TOUCHED()`, `GET_FLDBUF()`

COMBOBOX

This section contains the built-in functions, classes and methods used to manipulate the contents of a combobox at runtime. They allow a combobox to be managed dynamically and override the settings specified in the form file.


fgl_list_clear()

The fgl_list_clear() function clears combo list items from a list in accordance with the index value associated with each value for the named field.

Usage

This function takes three arguments: field_name, start_index and end_index. The field_name is expressed as a CHAR datatype while the index arguments are both INTEGERS.

The fgl_list_clear() function clears items from the list associated with the named field, from the start_index to end_index and will return a '1' if successful and a '0' in the case of a failure.

	If the start_index is set to '0', then all values in the item list will be cleared.
---	--

fgl_list_count()

The fgl_list_count() function returns the number of items in the value list associated with the named field.

Usage

The fgl_list_count() function takes a single argument: field_name(CHAR255), and will return the number of items listed in a particular combo box field.

Returns=1, failure=0.

fgl_list_find()

The fgl_list_find() function will retrieve the index number associated with a value from a combo box control.

Usage

This function takes two arguments: field_name which is a CHAR(255) datatype and value, which is a character string. The field_name argument refers to the combo control and value represents the item value searched for. If successful, the function will return the requested index value within the named field and '0' if no value is found.

fgl_list_get()

The fgl_list_get() function will retrieve a specific item from a Combo field control in accordance with the index value supplied.

Usage

This function takes two arguments: field_name and value. The first argument (field_name) is a CHAR datatype which identifies the combo box control. The second control is an INTEGER value that represents the index variable of the intended object. Use of this function will return a value as a CHAR datatype relevant to the specified index value where such a value exists. If the value is not associated with the combo/ list control then a NULL is returned.

fgl_list_insert()

The fgl_list_insert() function will insert a specified value into the Combo box control.

Usage

The fgl_list_insert function takes three arguments: field_name, index and value. The first argument, field_name, is a CHAR datatype whilst index and value are expressed as an INTEGER. The function will return the index number relating to where the insertion is made. Alternately, if the function is not successful a '0' value is returned.

fgl_list_sort()

The fgl_list_sort() function sorts the items displayed in a combo/ list control field in ascending/ descending order.

Usage

This function takes two arguments: field_name and direction. The field_name is a CHAR(255) datatype whilst direction is an INTEGER.

The specified integer value will determine the combo list ordering in accordance with >1=ascending. Returns=1, failure=0.

fgl_list_restore()

The fgl_list_restore() function is used to clear the item list for a Combo box control and create/ restore a new list based on the items specified as part of the INCLUDE attributes.

Usage

This function takes a single argument: field_name, which is a CHAR datatype. If the function is applied successfully the return value will be the size of the new list expressed as an INTEGER datatype. If the function is not executed successfully, a '0' is returned.

fgl_list_set()

The fgl_list_set() function is used to replace a combo field value with a specified value.

Usage

The fgl_list_set() function takes three arguments: field_name, index and value. The first argument, field_name, is a CHAR datatype whilst index and value are expressed as an INTEGER. The function will replace the specified combo value with a new value in accordance with the index value entered. If successful, the function returns the index of the inserted item. Alternately, if the function is not successful a '0' value is returned.

ui.COMBOBOX Class

The ComboBox class, which belongs to the ui namespace, is designed to provide a convenient interface to the ComboBox form widget.

SetDefaultInitializer()

The SetDefaultInitializer() method is used to specify the default function which will be used for combo box initialization each time the program creates a new ComboBox object. This class method needs the quoted name of the function or a variable containing the function name as an argument.

```
CALL ui.ComboBox.SetDefaultInitializer("init_function")
```

The initializing function should use the ComboBox object as the argument. The ComboBox object needs to be declared by a DEFINE statement within this function. Each time an object of ui.ComboBox class is initialized in the program, the initialization function is called and the new ui.ComboBox object is passed to it as a parameter. Thus it is possible to alter the combo box contents even before the corresponding form was opened.

Note, that the SetDefaultInitializer() method needs the function name specified in lower-case.

Here is an example of the method application:

```
CALL ui.ComboBox.setDefaultInitializer("init_func")
....
FUNCTION init_func(cbx)
  DEFINE cbx ui.ComboBox
  CALL cbx.clear()
  CALL cbx.addItem(1,"Village")
  CALL cbx.addItem(2,"Town")
  CALL cbx.addItem(3,"City")
END FUNCTION
```

ForName()

The ui.ComboBox.ForName() class method is used to search for a ComboBox widget by name in the current form. The found widget will be bound to a ui.ComboBox variable. The method needs only one argument, which specifies the name of the form field. The syntax of the method reference is as follows:

```
ui.ComboBox.ForName("[table.]field_name")
```

The ForName() method is typically used for initializing an object of the ComboBox class. Here is an example of the method usage:

```
DEFINE cbx ui.ComboBox
LET cbx = ui.ComboBox.ForName("place_of_living")
```

After the Combo Box is initialized, it can be referenced by other methods.

Clear()

The Clear() method is used to clear the dropdown list of the specified ComboBox Object.

AddItem()

The AddItem() method is used to add an item to the ComboBox dropdown list. The method needs two parameters:

```
CALL cbx.AddItem(value, "tag")
```

The value argument identifies the real value of the form field which will be recorded to the variable, if this combo box item is selected during the input. You should avoid trailing spaces in this argument. The values are truncated during the estimation, and the value which is got as the result will not match the ComboBox item name.

The tag argument specifies the value that will be actually displayed to the dropdown list. If this argument is NULL, the first argument will be used as a display value. If you use trailing spaces in this argument, the width of the ComboBox object can appear greater than you expect it to be. The VARCHAR or STRING data types as well as CLIPPED operator used with CHAR variables can help you to avoid this problem.

GetTableName()

The GetTableName() method is used to retrieve the table prefix of the specified form field. If no prefix is specified at the form field level, the returned value will be NULL.

GetColumnName()

The GetColumnName() method is used to retrieve the column prefix of the specified form field. If no prefix is specified at the form field level, the returned value will be NULL.

GetTag()

The GetTag() method is used to retrieve the value specified in the TAG attribute in the form file for the referenced combo box widget.

GetIndexOf()

The GetIndexOf() method is used return the position of the item in list. The method needs an argument specifying the item name and returns 0 if the item with the specified name does not exist.

```
CALL cbx.GetIndexOf("item1") RETURNING i
```

GetItemCount()

The GetItemCount() method is used to retrieve the total number of items currently present in the ComboBox dropdown list. The method returns zero, if the list is empty.

GetItemName()

The GetItemName() method is used to retrieve the value of the specified list item. The method needs an argument specifying the index of the item. The first item index is 1.

GetItemText()

The GetItemText() method is used to retrieve the value of the list item placed at the index specified as the method attribute. The index is a position of the item in the drop down list.

```
LET item_value = cbx.GetItemText(5)
```

GetTextOf()

The GetTextOf() method returns the value of an item whose tag is passed as a parameter. The tag of an item is the text that is displayed in the drop down list and the item value is the value that is actually written into a variable during the input if the specific item was selected.

```
LET item_value = cbx.GetTextOf("displayed name")
```

RemoveItem()

The RemoveItem() method is used to delete an item with the from the combo box list. The item is referenced by name which is passed to the method as an argument.

```
CALL cbx.RemoveItem("myitem")
```

Date/Time

current

The CURRENT operator returns the current date and time of day from the system clock as a DATETIME value of a specified or default precision.

Usage

The CURRENT operator reads the date and time from the system clock.

You can optionally specify the precision of the returned value by including a qualifier of the form first TO last, where first and last can be any of the following keywords:

- FRACTION (*n*) (currently, only Informix RDBMS supports this)
- SECOND
- MINUTE
- HOUR
- DAY
- MONTH
- YEAR

The first keyword must specify a time unit that is the same as or larger than what the last keyword specifies. For example, the following qualifier is valid:

```
CURRENT YEAR TO MONTH
```

But the following is not valid, because the unit Month is smaller than Year:

```
CURRENT MONTH TO YEAR
```

If FRACTION is the last keyword, you can include a digit *n* in parentheses to specify the scale (which can range from 1 to 5 digits) of the *seconds* value.

If no qualifier is specified, the default qualifier is YEAR TO FRACTION(3).

The following code sample illustrates the CURRENT Operand:

```
MAIN
  DEFINE
    inp_char CHAR(1),
    mydate DATE,
    d_yyff DATETIME YEAR TO FRACTION(5), --Note: Only Informix RDBMS supports DATE
    FRACTION
```

```
d_yymo DATETIME YEAR TO MONTH,
d_modd DATETIME MONTH TO DAY,
d_momi DATETIME MONTH TO MINUTE,
d_ddhh DATETIME DAY TO HOUR,
d_ddss DATETIME DAY TO SECOND,
d_hhmm DATETIME HOUR TO MINUTE,
d_hhss DATETIME HOUR TO SECOND,
d_mmff DATETIME MINUTE TO FRACTION

LET mydate = CURRENT
LET d_yyff = CURRENT YEAR TO FRACTION(5)
LET d_yymo = CURRENT YEAR TO MONTH
LET d_modd = CURRENT MONTH TO DAY
LET d_momi = CURRENT MONTH TO MINUTE
LET d_ddhh = CURRENT DAY TO HOUR
LET d_ddss = CURRENT DAY TO SECOND
LET d_hhmm = CURRENT HOUR TO MINUTE
LET d_hhss = CURRENT HOUR TO SECOND
LET d_mmff = CURRENT MINUTE TO FRACTION

DISPLAY "LET mydate = CURRENT" AT 3,3
DISPLAY mydate AT 3,50

DISPLAY "LET d_yyff = CURRENT YEAR TO FRACTION(5)" AT 4,3
DISPLAY d_yyff AT 4,50

DISPLAY "LET d_yymo = CURRENT YEAR TO MONTH" AT 5,3
DISPLAY d_yymo AT 5,50

DISPLAY "LET d_modd = CURRENT MONTH TO DAY" AT 6,3
DISPLAY d_modd AT 6,50

DISPLAY "LET d_momi = CURRENT MONTH TO MINUTE" AT 7,3
DISPLAY d_momi AT 7,50

DISPLAY "LET d_ddhh = CURRENT DAY TO HOUR" AT 8,3
DISPLAY d_ddhh AT 8,50

DISPLAY "LET d_ddss = CURRENT DAY TO SECOND" AT 9,3
DISPLAY d_ddss AT 9,50

DISPLAY "LET d_hhmm = CURRENT HOUR TO MINUTE" AT 10,3
DISPLAY d_hhmm AT 10,50

DISPLAY "LET d_hhss = CURRENT HOUR TO SECOND" AT 11,3
DISPLAY d_hhss AT 11,50

DISPLAY "LET d_mmff = CURRENT MINUTE TO FRACTION" AT 12,3
```

```
DISPLAY d_mmff AT 12,50

PROMPT "Press any key to close this demo application" FOR inp_char
END MAIN
```

If **CURRENT** is executed more than once in a statement, identical values might be returned at each call. Similarly, the order in which the **CURRENT** operator is executed in a statement cannot be predicted. For this reason, do not attempt to use this operator to mark the start or end of a 4GL statement or any specific point in the execution of a statement.

You can use the **CURRENT** operator both in SQL statements and in other 4GL statements. The following example is from an SQL statement:

```
SELECT * FROM activity
WHERE open_date = CURRENT YEAR TO DAY
```

This example is from a form specification file:

```
ATTRIBUTES -- 4GL field
timestamp = FORMONLY.tmstamp TYPE DATETIME HOUR \
TO SECOND,
DEFAULT = CURRENT HOUR TO SECOND;
```

The next example is from a report:

```
PAGE HEADER -- Report control block
PRINT COLUMN 40, CURRENT MONTH TO MONTH,
      COLUMN 42, "/",
      COLUMN 43, CURRENT DAY TO DAY,
      COLUMN 45, "/",
      COLUMN 46, CURRENT YEAR TO YEAR
```

The last example would not produce the correct results if its execution spanned midnight.

References

DATE, EXTEND(), TIME, TODAY

date

The DATE operator converts its CHAR, VARCHAR, DATETIME, or integer operand to a DATE value. If you supply no operand, it returns a character representation of the current date.

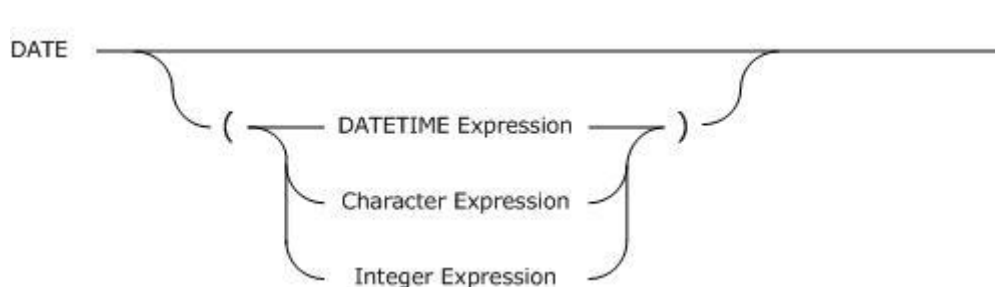


Figure 41 - DATE Operator

Usage

The DATE operator is able to convert other data type values to DATE values and can return the current date as a character string.

If no operand is used, the DATE operator reads the system clock calendar and returns the current date as a string in the following format:

`weekday month day year`

Where *weekday* and *month* are three character abbreviations of the day and month; *day* and *year* are, respectively, two and four figure numbers.

The following example uses the DATE operator to display the current date:

```
MAIN
  DEFINE my_date CHAR(15)
  LET my_date = DATE

  DISPLAY "This program was executed on ", my_date AT 5,5

  CALL fgl_winmessage("Exit","Press any key to close this demo application","info")
END MAIN
```

On Friday, 22 October, 2004, this example would have displayed the string:

`This program was executed on Fri Oct 22 2004`

The effect of the DATE operator is sensitive to the time of execution and to the accuracy of the system clock. An alternative way to format a DATE value as a character string is to use the FORMAT field attribute in a screen form, or with the USING operator.

The DATE operator can also perform data type conversion of a character, integer, or DATETIME operand to a DATE value, equivalent to a count of days since the beginning of the last year of the 19th century:

- Converting a properly formatted character string representation of a numeric date to a DATE value. (The default format is mm/dd/yy, but the DBDATE or GL_DATE environment variable can change this default.)
- Converting a DATETIME value to a negative or positive integer. The returned integer corresponds to the number of days between the specified date and 31 December, 1899.
- Obtaining a DATE value from a negative or positive integer that specifies the number of days between the specified date and 31 December, 1899.

The following program fragment illustrates uses of the DATE operator:

```
MAIN

DEFINE
    my_date DATE,
    my_date_time DATETIME YEAR TO DAY,
    char_inp char(1)

LET my_date = DATE (" 02/09/04 ") -- this line requires the default DATE format
DISPLAY "DATE (\\" 02/09/04 \") Result: ", my_date at 5,5

LET my_date = DATE (" 2004-02-09 ") -- it requires that DBDATE be set to Y4MDLET
DISPLAY "DATE (\\" 2004-02-09 \") Result: ", my_date at 6,5

LET my_date = DATE (" 09:04:02 ") -- this line requires that DBDATE be set to DY2M:
DISPLAY "DATE (\\" 09:04:02 \") Result: ", my_date at 7,5

LET my_date = DATE(my_date) -- The operand can be a DATE variable, as illustrated,
DISPLAY "DATE(my_date) Result: ", my_date at 8,5

-- or getting the integer number of days since the last day of the year 1899
LET my_date = DATE (0) -- The result is: 12/31/1899
DISPLAY "DATE (0) Result: ", my_date at 9,5

LET my_date = DATE (38000) -- The result is: 15/1/2004
DISPLAY "DATE (38000) Result: ", my_date at 10,5

-- Or the operand can also be of data type DATETIME
LET my_date_time = CURRENT
DISPLAY "CURRENT Result: ", my_date_time at 11,5

LET my_date = DATE (my_date_time) -- the result is today's date
```

```
DISPLAY "DATE (my_date_time) Result: ", my_date_time at 12,5

LET my_date = DATE(CURRENT) -- the same result as previous
  DISPLAY "DATE(CURRENT) Result: ", my_date_time at 13,5

PROMPT "Press any key to close this demo application" for char_inp

END MAIN
```

References

CURRENT, DATE, DAY, MDY(), MONTH, TIME, UNITS, WEEKDAY, YEAR

day()

The DAY() operator returns a positive integer, corresponding to the day portion of the value of its DATE or DATETIME operand.



Figure 42 - DAY() Operator

Usage

The DAY() operator can extract an integer value for the day of the month from a DATETIME or DATE operand. This feature is helpful in some applications because INTEGER values are easier than DATETIME or DATE values to manipulate with arithmetic operators.

The following program fragment extracts the day of the month from a DATETIME literal:

```
MAIN
  DEFINE
    d_var, m_var, y_var INTEGER,
    date_var DATETIME YEAR TO SECOND

  LET date_var = DATETIME (04-10-22 20:32:12) YEAR TO SECOND

  LET d_var = DAY(date_var)
  LET m_var = MONTH(date_var)
  LET y_var = YEAR(date_var)

  DISPLAY "*** Today's Date information ***" at 3,5

  DISPLAY "The number of the day is: ", d_var USING "##" at 5,5
  DISPLAY "The number of the month is: ", m_var USING "##" at 7,5
  DISPLAY "The number of the year is: ", y_var USING "####" at 9,5

  CALL fgl_winmessage("Exit","Press any key to close this demo application","info")
END MAIN
```

References

CURRENT, DATE, MONTH(), TIME, TODAY, WEEKDAY(), YEAR()

extend()

The EXTEND() operator converts its DATETIME or DATE operand to a DATETIME value of a specified (or default) precision and scale.

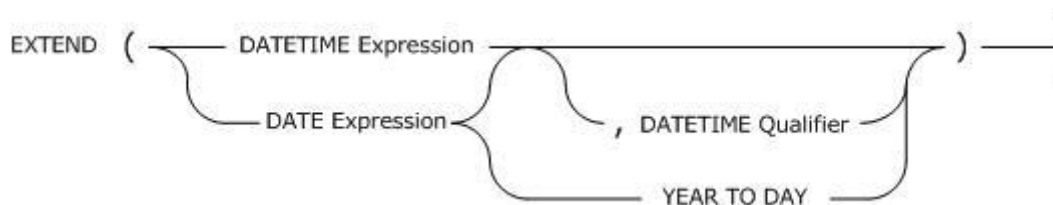


Figure 43 - EXTEND() Function

Usage

The EXTEND() operator returns the value of its DATE or DATETIME operand, but with an adjusted precision that you can specify by a DATETIME qualifier. The operand can be a DATE or DATETIME expression of any valid precision. 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" whose time units are ambiguous
- It cannot be a time expression that returns an INTERVAL value

DATETIME Qualifiers

A qualifier can specify the precision of the result (and the scale, if FRACTION is the last keyword in the qualifier). The qualifier follows a comma and is of the form first TO last, where first and last are keywords to specify (respectively) the largest and smallest time unit in the result. Both can be the same.

If no qualifier is specified, the following defaults are in effect, based on the explicit or default precision of the DATE or DATETIME operand:

- The default qualifier that EXTEND() applies to a DATETIME operand is YEAR TO FRACTION(3).
- The default qualifier for a DATE operand is YEAR TO DAY.

The following rules are in effect for DATETIME qualifiers that you specify as EXTEND() operands:

- If a first TO last qualifier is specified, the first keyword must specify a time unit that is larger than (or the same as) the time unit that the last keyword specifies.
- If first specifies a time unit larger than any in the operand, omitted time units are filled with values from the system clock-calendar. In the following fragment, the first keyword specifies a time unit larger than any in `t_stamp`, so the value of the current year would be used:

MAIN

```
DEFINE t_year_min DATETIME YEAR TO MINUTE

DISPLAY "t_year_min = EXTEND (DATETIME (2004-12-1) YEAR TO DAY, YEAR TO MINUTE)" AT
3,1
DISPLAY "- INTERVAL (720) MINUTE(3) TO MINUTE" AT 4,1

LET t_year_min = EXTEND (DATETIME (2004-12-1) YEAR TO DAY, YEAR TO MINUTE) -
INTERVAL (720) MINUTE(3) TO MINUTE
--result: DATETIME (2004-11-30 12:00) YEAR TO MINUTE

DISPLAY "t_year_min = ", t_year_min AT 7,1
CALL fgl_winmessage("Exit","Press any key to close this demo application","info")

END MAIN
```

- If last specifies a smaller time unit than any in the operand, the missing time units are assigned values according to these rules:

A missing MONTH or DAY is filled in with the value one (01).

Any missing HOUR, MINUTE, SECOND, or FRACTION is filled in with the value zero (00).

- If the operand contains time units outside the precision specified by the qualifier, the unspecified time units are discarded. For example, if you specify first TO last as DAY TO HOUR, any information about MONTH in the DATETIME operand is not used in the result.

Using EXTEND with Arithmetic Operators

If the precision of an INTERVAL value includes a time unit that is not present in a DATETIME or DATE value, you cannot combine the two values directly with the addition (+) or subtraction (-) binary arithmetic operators. You must first use the EXTEND() operator to return an adjusted DATETIME value on which to perform the arithmetic operation.

For example, you cannot directly subtract the 720-minute INTERVAL value in the next example from the DATETIME value that has a precision from YEAR to DAY. You can perform this calculation by using the EXTEND() operator:

MAIN

```
DEFINE t_year_min DATETIME YEAR TO MINUTE

DISPLAY "t_year_min = EXTEND (DATETIME (2004-12-1) YEAR TO DAY, YEAR TO MINUTE)" AT
3,1
DISPLAY "- INTERVAL (720) MINUTE(3) TO MINUTE" AT 4,1

LET t_year_min = EXTEND (DATETIME (2004-12-1) YEAR TO DAY, YEAR TO MINUTE) -
INTERVAL (720) MINUTE(3) TO MINUTE
```

```
--result: DATETIME (2004-11-30 12:00) YEAR TO MINUTE

DISPLAY "t_year_min = ", t_year_min AT 7,1
CALL fgl_winmessage("Exit","Press any key to close this demo application","info")

END MAIN
```

Here the EXTEND() operator returns a DATETIME value whose precision is expanded from YEAR TO DAY to YEAR TO MINUTE. This adjustment allows 4GL to evaluate the arithmetic expression. The result of the subtraction has the extended precision of YEAR TO MINUTE from the first operand.

In the next example, fragments of a report definition use DATE values as operands in expressions that return DATETIME values. Output from these PRINT statements would in fact be the numeric date and time without the DATETIME keywords and qualifiers that are included here to show the precision of the values that the arithmetic expressions return.

```
DEFINE date_val DATE
DEFINE dt_val DATETIME YEAR TO HOUR
LET date_val = TODAY
LET dt_val = EXTEND(date_val, YEAR TO HOUR)
```

You cannot directly combine a DATE with an INTERVAL value for which the last qualifier is smaller than DAY. But, as the previous example shows, you can use the EXTEND() operator to convert the value in a DATE column or variable to a DATETIME value that includes all the fields of the INTERVAL operand.

In the next example, the INTERVAL variable **how_old** includes fields that are not present in the DATETIME variable **t_stamp**, so the EXTEND() operator is required in the expression that calculates the sum of their values.

```
MAIN
START REPORT repl TO SCREEN
OUTPUT TO REPORT repl()
FINISH REPORT repl

DISPLAY "*** EXTEND Example 3 ***" AT 3,5
DISPLAY "LET calendar = \"05/18/1999\"" AT 5,5
DISPLAY "PRINT calendar - INTERVAL (5-5) YEAR TO MONTH" AT 6,5
DISPLAY "PRINT EXTEND(calendar, YEAR TO HOUR) - INTERVAL (4 8) DAY TO HOUR" AT 7,5

CALL fgl_winmessage("Exit","Press any key to close this demo application","info")

END MAIN

REPORT repl()
DEFINE calendar DATE
```

```
FORMAT
ON EVERY ROW
  LET calendar = "05/18/1999"
  PRINT calendar - INTERVAL (5-5) YEAR TO MONTH
--result: DATETIME (1993-12-18) YEAR TO DAY
  PRINT EXTEND(calendar, YEAR TO HOUR) - INTERVAL (4 8) DAY TO HOUR
--result: DATETIME (1999-05-13 16) YEAR TO HOUR
END REPORT

MAIN

DEFINE t_day_hour DATETIME YEAR TO HOUR
DEFINE t_day_min DATETIME DAY TO MINUTE
DEFINE t_difference INTERVAL DAY TO MINUTE

LET t_day_hour = "1989-12-04 17"
LET t_day_min = INTERVAL (28 9:25) DAY TO MINUTE
LET t_difference = EXTEND(t_day_hour, DAY TO MINUTE) + t_day_min

DISPLAY "*** EXTEND Example 3 ***" AT 3,5
DISPLAY "LET t_day_hour = \"1989-12-04 17\"" AT 5,5
DISPLAY "value t_day_hour : " , t_day_hour at 6,5

DISPLAY "LET t_day_min = INTERVAL (28 9:25) DAY TO MINUTE" AT 8,5
DISPLAY "value t_day_min : " , t_day_min at 9,5

DISPLAY "LET t_difference = EXTEND(t_day_hour, DAY TO MINUTE) + t_day_min" AT 11,5
DISPLAY "value t_difference : " , t_difference at 12,5

CALL fgl_winmessage("Exit","Press any key to close this demo application","info")
END MAIN
```

SQL statements can include a similar EXTEND() operator of SQL, whose first argument can be the name of a DATETIME or DATE database column.

Reference

UNITS

mdy()

The MDY() operator returns a value of the DATE data type from three comma-separated integer operands that represent the *month*, the *day* of the month, and the *year*.

Usage

The MDY() operator converts to a single DATE format a list of exactly three valid integer expressions. The three expressions correspond with the month, day, and year elements of a calendar date:

- The first expression must return an integer, representing the number of the month (from 1 to 12).
- The second must return an integer, representing the number of the day of the month (from 1 up to 31, depending on the month).
- The third must return a four-digit integer, representing the year.

If you specify values outside the expected ranges of days and months in the calendar or if you do not include three operands, an error will be returned.

The three integer expression operands must be placed between parentheses, and separated by commas.

The third expression cannot be the abbreviation for the year. For example, 99 would be taken to mean the year before 100 AD.

The following program uses MDY() to return a DATE value, which is then assigned to a variable and displayed on the screen:

```
MAIN
  DEFINE my_date DATE

  DISPLAY "*** MDY() Example ***" AT 1,5

  DISPLAY "LET my_date = MDY(06/3,3+9,2005)" AT 3,5
  LET my_date = MDY(06/3,3+9,2005)
  DISPLAY my_date at 5,5

  CALL fgl_winmessage("Exit","Press any key to close this demo application","info")
END MAIN
```

Reference

DATE()

month()

The MONTH() operator returns a positive whole number between 1 and 12, corresponding to the *month* portion of a DATE or DATETIME operand.

Usage

The MONTH() operator extracts an integer value for the month in a DATE or DATETIME value. You cannot specify an INTERVAL operand.

The following program example extracts the month time unit from a DATETIME literal expression. It evaluates MONTH(**date_var**) as an operand of a Boolean expression to test whether the month is earlier in the year than August.

Sample Code: month_operator.4gl

```
MAIN

DEFINE
    date_var DATETIME YEAR TO SECOND,
    current_month CHAR(10),
    month_var INT

LET current_month = CURRENT MONTH TO MONTH
LET date_var = DATETIME(04-12-24 18:47:32) YEAR TO SECOND

LET month_var = MONTH(date_var)

DISPLAY "*** MONTH Operator Example ***" AT 1,5

DISPLAY "The current month is: ", current_month at 3,5
DISPLAY "DATETIME(04-12-24 18:47:32) YEAR TO SECOND:" at 4,5
DISPLAY "The month of interest is month number: ",month_var USING "##" at 5,5

IF MONTH(date_var) < 7 THEN
    DISPLAY "Month of interest is in the range of Jan, Feb, Mar, Apr, May or Jun"
at 6,5
ELSE
    DISPLAY "Month of interest is in the range of Jul, Aug, Sep, Oct, Nov or Dec"
at 6,5
END IF

CALL fgl_winmessage("Exit","Press any key to close this demo application","info")
END MAIN
```

References

DATE(), DAY(), TIME(), WEEKDAY(), YEAR()

time()

The TIME() operator converts the time-of-day portion of its DATETIME operand to a character string. If you supply no operand, TIME reads the system clock and returns a character string value representing the current time of day.



Figure 44 - TIME Operator

Usage

TIME returns a character string that represents the time-of-day portion of its DATETIME operand in the format *hh:mm:ss*, based on a 24-hour clock. (Here *hh* represents the hour, *mm* the minute, and *ss* the second as 2-digit strings, with colons as separators.)

If you do not supply an operand, TIME returns a character string that represents the current time in the format *hh:mm:ss*, based on a 24-hour clock.

In the following program fragment, the value returned by TIME is assigned to the **my_time** variable and displayed:

Sample Code: time_operator.4gl

```
MAIN

    DEFINE my_time char(15)
    LET my_time = TIME

    DISPLAY "*** TIME Example ***" AT 3,1

    DISPLAY "The current time is: ", my_time at 5,5

    CALL fgl_winmessage("Exit","Press any key to close this demo application","info")
    END MAIN
```

If this code were executed one hour before midnight, the previous DISPLAY statement would produce output in the following format:

```
The time is 23:00:00
```

Like the values returned by the CURRENT, DATE, DAY, MONTH, TODAY, WEEKDAY, and YEAR operators, the value that TIME returns is sensitive to the time of execution and to the accuracy of the system clock-calendar.

References

CURRENT, DATE, DAY, MONTH, TODAY, WEEKDAY, YEAR

today

The TODAY operator reads the system clock and returns a DATE value that represents the current calendar date.

Usage

TODAY can return the current date in situations where the time of day (which CURRENT or TIME supplies) is not necessary. Like the CURRENT, DATE, DAY, MONTH, TIME, WEEKDAY, and YEAR operators, TODAY is sensitive to the time of execution and to the accuracy of the system clock-calendar. The following example uses TODAY in a simple program:

Sample Code: today_operator.4gl

```
MAIN
  DEFINE my_date DATE
  LET my_date = today

  DISPLAY "*** TODAY Example***" AT 1,5

  DISPLAY my_date at 5,5

  CALL fgl_winmessage("Exit","Press any key to close this demo application","info")
END MAIN
```

TODAY is useful in setting defaults and initial values in form fields. The next code fragment initializes a field with the current date if the field is empty. This initialization takes place before the user enters data into the field:

Sample Code: Initializing an Empty Field With Today's Date

```
LET invoice_rec.inv_date = TODAY
INPUT invoice_rec WITHOUT DEFAULTS FROM sc_inv.*
```

When using Lycia, the presentation of DATE values is determined by the setting of the LANG environment variable. This makes use of the native operating system's capabilities. This does mean that to display DATE values correctly you must have the appropriate locale installed on your local OS.

References

CURRENT, DATE, DAY, MONTH, TIME, WEEKDAY, YEAR

units

The UNITS operator converts an integer expression to an INTERVAL value, expressed in a single unit of time that you specify after the UNITS keyword. The only specified unit of time that takes an argument is the FRACTION keyword, which takes a literal integer of between 1 and 6, being the number of decimal places to which the fraction is measured.

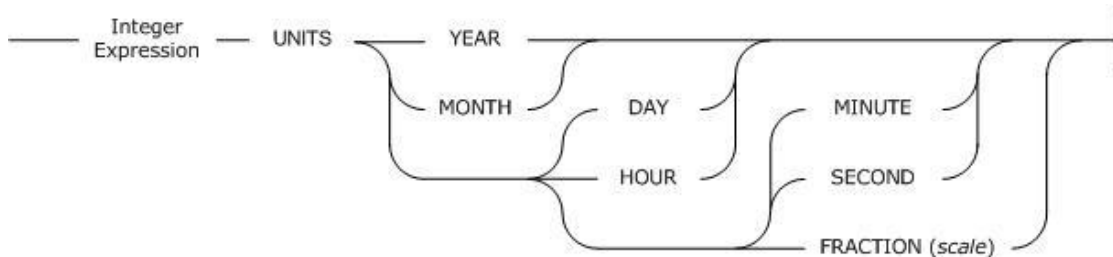


Figure 45 - UNITS Operator

Usage

The UNITS operator returns an INTERVAL value for a single unit of time, such as DAY TO DAY, YEAR TO YEAR, or HOUR TO HOUR. If you substitute a number expression for the integer operand, any fractional part of the returned value is discarded before the UNITS operator is applied.

Sample Code: `units_operator.4gl`

```
MAIN
  DEFINE dt1 DATETIME YEAR TO SECOND
  DEFINE dt2 DATETIME YEAR TO SECOND

  LET dt1 = CURRENT YEAR TO SECOND

  DISPLAY "Initial date value: ", dt1
  LET dt2 = dt1 + 5 UNITS YEAR
  DISPLAY "Increased by 5 years: ", dt2
  LET dt2 = dt1 + 1 UNITS MONTH
  DISPLAY "Increased by 1 month: ", dt2
  LET dt2 = dt1 + 5 UNITS DAY
  DISPLAY "Increased by 5 days: ", dt2
  LET dt2 = dt1 + 8 UNITS HOUR
  DISPLAY "Increased by 8 hours: ", dt2
  LET dt2 = dt1 + 5 UNITS MINUTE
  DISPLAY "Increased by 5 minutes: ", dt2
  LET dt2 = dt1 + 20 UNITS SECOND
  DISPLAY "Increased by 20 seconds: ", dt2

  SLEEP 3
END MAIN
```

UNITS has a higher precedence than any arithmetic or Boolean operator of 4GL. Any left-hand arithmetic operand that includes the UNITS operator must be enclosed within parentheses. The next example specifies a starting time for a meeting (DATETIME value) and a value for the duration of the meeting, which the program has already converted to a whole number of minutes (SMALLINT). The program calculates when the meeting will end (DATETIME value). UNITS in this case allows you to add the SMALLINT value to the DATETIME value and get a new DATETIME value.

```
LET end_time = (meeting_length UNITS MINUTE) \
+ start_time
```

Because the difference between two DATE values is an integer count of days rather than an INTERVAL data type, you might want to use the UNITS operator to convert such differences explicitly to INTERVAL values:

```
LET lateness = (date_due - TODAY) UNITS DAYS
```

Arithmetic operations with UNITS can return an invalid date. For example, the expression (1 UNITS MONTH) + DATETIME (2001-1 31) YEAR TO DAY returns February 31, 2001, and also a runtime error:

```
-1267: The result of a datetime computation is out of range.
```

weekday()

The WEEKDAY() operator returns a positive integer, corresponding to the day of the week implied by its DATE or DATETIME operand.



Figure 46 - WEEKDAY() Operator

Usage

This operator takes a DATETIME or DATE operand, and returns an integer in the range 0 through 6, where 0 represents Sunday, 1 represents Monday, and so on.

The next example calls a function that uses WEEKDAY with a CASE statement to assign a three-letter day-of-the-week abbreviation to each date in an array, omitting days that fall on weekends:

```
MAIN
  DEFINE
    test_day date,
    print_line, i, week_day_num smallint,
    day_name char(3),
    pa_days ARRAY[21] OF RECORD
      rdate char(3),
      dayo_week char(3)
    END RECORD

  LET test_day = TODAY

  DISPLAY "weekday() returns the weekday (0-6) of any given date" at 1,5

  DISPLAY "Date:" at 3, 5
  DISPLAY "weekday() return" at 3, 20
  DISPLAY "Day Name:" at 3, 40

  FOR i = 1 TO 21
    CALL size_theday(test_day)
    RETURNING day_name, week_day_num
    LET pa_days[i].dayo_week = day_name
    LET pa_days[i].rdate = test_day
    LET test_day = test_day + 1
    LET print_line = i + 3

  DISPLAY test_day at print_line, 5
```

```
        DISPLAY week_day_num at print_line, 20
        DISPLAY day_name at print_line, 40

END FOR

CALL fgl_winmessage("Exit","Press any key to close this demo application","info")
END MAIN

FUNCTION size_theday(test_day)
    DEFINE
        week_day_num SMALLINT,
        day_name CHAR(3),
        test_day DATE

    LET week_day_num = WEEKDAY(test_day)
    CASE week_day_num
        WHEN 1 LET day_name = "Mon"
        WHEN 2 LET day_name = "Tue"
        WHEN 3 LET day_name = "Wed"
        WHEN 4 LET day_name = "Thu"
        WHEN 5 LET day_name = "Fri"
        WHEN 6 LET day_name = "Sat"
        -- LET next_day = next_day + 2
        WHEN 0 LET day_name = "Sun"
        -- LET next_day = next_day + 1
    END CASE

    RETURN day_name, week_day_num
END FUNCTION
```

This operator is useful for determining the day of the week from dates in recent and future years.

The WEEKDAY() operator is among a group of 4GL operators that extract a single time unit value from a DATETIME or DATE value. The following *extraction* operators of 4GL accept a DATETIME or DATE operand.

- DAY() The day of the month
- MONTH() The month
- YEAR() The year
- WEEKDAY() The day of the week

In addition, the DATE() operator can extract the date portion of a DATETIME value that has YEAR TO DAY or greater precision, and the TIME operator can extract the time-of-day from a DATETIME expression that has HOUR TO FRACTION precision (or a subset thereof).

The USING operator can also return day-of-the-week information from a DATE operand (as described in “Formatting DATE Values” elsewhere in this volume).

For more information, see the **GL_DATE** environment variable elsewhere in the Reference Guide.

References

CURRENT, DATE, DAY, MONTH, TIME, TODAY, YEAR

year()

The YEAR() operator returns an integer corresponding to the *year* portion of its DATE or DATETIME operand.

Usage

The YEAR() operator returns all the digits of the year value (1999, not 99). The second example that follows illustrates how to obtain a two-digit value like 99 from a four-digit year like 1999.

The following example extracts the current year and stores the value in an integer variable:

Sample Code: year1_operator.4gl

```
MAIN
  DEFINE
    todays_date DATE,
    year_number integer

  LET year_number = year(TODAY)

  DISPLAY "*** YEAR() Function Example ***" AT 1,5
  DISPLAY "year(TODAY) The current year is: " , year_number AT 5,5

  CALL fgl_winmessage("Exit","Press any key to close this demo application","info")
END MAIN
```

You can produce a two-digit year abbreviation by using the MOD (modulus) operator:

Sample Code: year2_operator.4gl

```
MAIN
  DEFINE
    todays_date, celeb_date DATE,
    year_number integer

  DISPLAY "*** YEAR Function Example 2 ***" AT 1,5

  CALL fgl_putenv("DBDATE=mdy4/")

  LET celeb_date = "01/27/2014"
  LET year_number = YEAR(celeb_date)
  DISPLAY "Year is: " , year_number AT 3, 5
  LET year_number = year_number MOD 100
```

```
DISPLAY "The current year is: " , year_number AT 5,5

CALL fgl_winmessage("Exit","Press any key to close this demo application","info")
END MAIN
```

In the right-hand expression, the MOD operator yields the year modulo 100, the remainder when the value representing the actual year is divided by 100. For example, if the value of the DATE variable **celeb_date** is 01-27-2014, the YEAR() operator extracts the value 2014, and the following expression returns 14:

```
2014 MOD 100
```

That value is then assigned to the INT variable **celeb_yr**.

References

CURRENT, DATE, DAY, MONTH, TIME, TODAY, WEEKDAY

String

clipped

The CLIPPED operator takes a character operand and returns the same character value, but without any following spaces.

Usage

Character expressions often have a data length less than their total size. The following DISPLAY statement, for example, would produce output that included 200 trailing blanks if CLIPPED were omitted but displays only 22 characters when CLIPPED is included. The following code sample illustrates the CLIPPED operator:

```
MAIN

    DEFINE my_string CHAR(75)
    LET my_string = "This string is shorter than 75 characters"

    DISPLAY "To demonstrate the effect of clipped, we display numbers on the entire
line" AT 3,5
    DISPLAY "12345678901234567890123456789012345678901234567890123456789012<not
Clipped>" AT 5,5
    DISPLAY "12345678901234567890123456789012345678901234567890123456789012<was
Clipped>" AT 6,5

    CALL fgl_winmessage("Draw normal and clipped text","Press OK to draw the same
string normal and clipped","info")

    DISPLAY my_string at 5,5
    DISPLAY my_string CLIPPED at 6,5

    CALL fgl_winmessage("Exit","Press any key to close this demo application","info")
END MAIN
```

The CLIPPED operator can be useful in the following kinds of situations:

- After a variable in a DISPLAY, ERROR, LET, MESSAGE, or PROMPT statement, or in a PRINT statement of a REPORT program block
- When concatenating several character expression into a single string
- When comparing two or more character expressions and one or more of them is already clipped

The CLIPPED operator can affect the value of a character variable within an expression. CLIPPED does not affect the value when it is stored in a variable (unless you are concatenating CLIPPED values together). For example, if CHAR variable **z** contains a string that is shorter than the declared length of CHAR variable **y**, the following LET statement pads **y** with trailing blanks, despite the CLIPPED operator:

```
LET y = z CLIPPED
```

However, if CHAR variable **z** contains a string value no longer than the declared maximum size of VARCHAR variable **vc**, the following statement discards any trailing blanks from what it stored in **vc**:

```
LET vc = z
```

The following code segment is used to build a file name from various CHAR variables.

```
MAIN
  DEFINE file_path CHAR(1024)
  DEFINE file_source CHAR(1024)

  LET file_source = fgl_getenv("QUERIXDIR")
  LET file_source = file_source CLIPPED,
  "/documentation/Multiple_Database_Usage.pdf"

  CALL fgl_message_box("Ready to transfer file")

  CALL fgl_file_to_client(file_source,"sample_file.pdf")
  RETURNING file_path

  CALL fgl_message_box("File has been stored in " || file_path CLIPPED)
END MAIN
```

The following program fragment is from a sample program. Here CLIPPED is used to format a CHAR variable between two pipe symbols.

```
MAIN
  DEFINE str1 CHAR(50)

  LET str1 = "This is a text string"

  DISPLAY "Clipped string: |", str1 CLIPPED, "|" AT 3, 3
  DISPLAY "Unclipped string: |", str1, "|" AT 4, 3

  CALL fgl_message_box("Exit this application")
END MAIN
```

Relative to other 4GL operators, CLIPPED has a very low precedence. This can lead to confusion in some contexts, such as specifying compound Boolean conditions. For example, **qfgl** parses the condition:

```
IF LENGTH(f1) > 0 AND f1 CLIPPED != "first_name" THEN
```

as if it were delimited with parentheses as:

```
IF (((LENGTH(f1) > 0) AND f1) CLIPPED) != "first_name" THEN
```

To achieve the required result, you can write the expression as:

```
IF LENGTH(f1) > 0 AND (f1 CLIPPED) != " first_name " THEN
```

Reference

USING

Concatenation (||) Operator

The double pipe symbol (||) concatenation operator joins two operands of any simple data type, returning a single string.

Usage

The concatenation operator joins two strings, with a left-to-right association.

For example, (a || b || c) and ((a || b) || c) are equivalent expressions. The precedence of || is greater than LIKE or MATCHES, but less than the arithmetic operators. Like arithmetic operators, || returns a null value (as a zero-length string) if either operand has a null value.

A comma can be used in LET statements to concatenate strings, but it has a different rule for null values. If one string operand is NULL, the result is the other string. For example, the comma separator in the right-hand expression list of the LET statement (discussed in detail in the 4GL Statements volume) has concatenation semantics, that ignore any null values. However, if all of the operands in a comma-separated list are NULL, the LET statement returns a null value but represents it as a single blank space.

The following left-hand and right-hand expressions are equivalent:

x y + z	(x (y + z))
"xyz " NULL	NULL
"xyz " CLIPPED "AB"	"xyzAB"

Concatenation using the double pipe symbol || ignores any spaces that appear after the operands of integer and fixed-point number data types, but not after character or floating-point data types.

The CLIPPED operator can remove trailing blanks from values before concatenation in 4GL statements, but TRIM must replace CLIPPED in preparable SQL statements (for Version 7.x and later Informix databases).

Example Program:

```
MAIN
  DEFINE
    n,s,x,y,z CHAR(10),
    result CHAR(50),
    inp_char CHAR(1)

  LET s = "MyStr X  "
  LET x = "MyString X"
  LET y = "MyString Y"
  LET z = "MyString Z"
  ##LET n = NULL --keep it NULL

  DISPLAY "n = \"NULL      \" AT 5,5
  DISPLAY "s = \"\",s,  \"\" AT 6,5
  DISPLAY "x = \"\",x,  \"\" AT 7,5
```

```
DISPLAY "y = \"", y,  "\"\" AT 8,5
DISPLAY "z = \"", z,  "\"\" AT 9,5

LET result = x || y || z
DISPLAY "x || y || z      = \" AT 12,5
DISPLAY result AT 12,30 ATTRIBUTE(MAGENTA)

LET result = x || y + z
DISPLAY "x || y + z      = \" AT 13,5
DISPLAY result AT 13,30 ATTRIBUTE(MAGENTA)

LET result = (x || (y + z))
DISPLAY "(x || (y + z))  = \" AT 14,5
DISPLAY result AT 14,30 ATTRIBUTE(MAGENTA)

LET result = x || n
DISPLAY "x || NULL      = \" AT 15,5
DISPLAY result AT 15,30 ATTRIBUTE(RED)

LET result = x || s || z
DISPLAY "x || s || z    = \" AT 16,5
DISPLAY result AT 16,30 ATTRIBUTE(MAGENTA)

LET result = x || s CLIPPED || z
DISPLAY "x || s CLIPPED || z = \" AT 17,5
DISPLAY result AT 17,30 ATTRIBUTE(MAGENTA)

PROMPT "Press any key to close this demo application" FOR inp_char

END MAIN
```

References

CLIPPED, COLUMN, Substring ([]) operator

downshift()

The DOWNSHIFT() function takes a character expression as its argument, and returns a string value in which all uppercase characters in its argument are converted to lowercase.

Usage

The DOWNSHIFT() function is typically called to regularize character data. It could be used, for example, when handling US addresses data, to stop a state abbreviation entered as NY, Ny, or ny from resulting in different values, if within your application, they should all be considered as the same value

Non-alphabetic or lowercase characters are not altered by DOWNSHIFT(). The maximum data length of the argument (and of the returned character string value) is 32,766 bytes.

Where an expression allows it, the DOWNSHIFT() function can be used, or you can assign the value returned by the function to a variable.

As shown in the following code sample, the user enters a string input_string using the fgl_winprompt function. Next, the functions DOWNSHIFT and UPSHIFT convert the string to upper/lower case.

```
MAIN

    DEFINE input_string char(50)

    CALL fgl_winprompt(5,5,"Enter a string - max 50 characters", "not-specified", 50,
    0) returning input_string

    #convert to lower case
    CALL fgl_winmessage("DOWNSHIFT converts to lowercase", DOWNSHIFT(input_string),
    "info")

    #convert to upper case
    CALL fgl_winmessage("UPSHIFT converts to uppercase", UPSHIFT(input_string), "info")

END MAIN
```

Reference

UPSHIFT()

fgl_charbool_to_intbool()

This function converts an Informix Universal Server BOOLEAN value ("t", "f") to a standard 4GL INTEGER value (TRUE, FALSE). The function returns NULL if the given value does not match "T", "t", "F" or "f".

When you retrieve a BOOLEAN value from the database, you can convert that value with the *fgl_charbool_to_intbool()* as in the following example :

Code Sample Using fgl_charbool_to_intbool ()

```
DATABASE cms
MAIN
    DEFINE mbox RECORD LIKE mailbox.*
    SELECT mailbox.* INTO mbox.* FROM mailbox
    IF fgl_charbool_to_intbool( rtab1.read_flag ) THEN
        DISPLAY "TRUE"
    ELSE
        DISPLAY "FALSE"
    END IF
END MAIN
```

fgl_intbool_to_charbool()

This function converts a standard 4GL INTEGER value to a CHAR(1) value to be stored in a BOOLEAN column of an Informix Universal Server table. The function returns NULL if the given value is NULL.

Code Sample Using fgl_intbool_to_charbool()

```
DATABASE cms
MAIN

    DEFINE vbool CHAR(1)
    # Does CMS have a vbool ?
    LET vbool = fgl_intbool_to_charbool( 1 = 0 )

    UPDATE mailbox
        SET read_flag = vbool
        WHERE ...
END MAIN
```

length()

The LENGTH() function takes a character string argument and returns an integer, which represents the number of bytes in its argument, ignoring any blank spaces after the character string.

Usage

The LENGTH() function displays a dialog box with a field, into which you can type a string. The string length is then measured and a message window confirms the string and tells you, by returning an integer value, how many characters it contains.

Code Sample: length_function.4gl

```
MAIN
  DEFINE input_string CHAR(500)

  CALL fgl_winprompt(5, 2, "Enter a string to count its lengths (max. 40)", "",
40, 0)
  RETURNING input_string

  DISPLAY "Your specified string: " , input_string at 5,5
  DISPLAY "Hast got following char length: ", length(input_string) at 7,5

  CALL fgl_winmessage("Exit","Press any key to close this demo
application","info")
END MAIN
```

Statements in the next example center a report title on an 80-column page:

```
LET title = "Invoice for ", fname CLIPPED,
           " ", lname CLIPPED
LET offset = (80 - length(title))/2
PRINT COLUMN offset, title
```

These are some of the uses that the LENGTH() function can be used for:

- You can check whether a user has entered a database name and, if not, set a default name.
- Check whether the user has supplied the name of a file to receive the output from a report and, if not, set a default output.
- Use LENGTH(*string*) as the MAX() value in a FOR loop, and check each character in *string* for a specific character. For example, you can check for a period (.) to determine whether a table name has a prefix.

LENGTH() is also useful as a check on user input. In the following example, an IF statement is used to determine whether the user has responded to a displayed message:

```
IF LENGTH (l_contact.cont_name) = 0 THEN
  NEXT FIELD cont_name
  ERROR "You must enter a value for this field" ELSE ...
```

If its argument is a null string, LENGTH() returns zero.

Using LENGTH() in SQL Expressions

LENGTH() is one of the 4GL built-in functions that can also be used in SQL statements. It can also be called from a C function.

In a SELECT or UPDATE statement, the argument of LENGTH() is the identifier of a character column. In this context, LENGTH() returns the number of bytes in the CLIPPED data value (for CHAR and VARCHAR columns) or the full number of bytes (for TEXT and BYTE data types).

The LENGTH() function can also take the name of a database column as its argument but only within an SQL statement.

References

CLIPPED, USING

orf()

The ORD() function takes a character expression as an argument and returns the integer value of the first byte of that argument.

For the default (U.S. English) locale, the ORD() function is the logical inverse of the ASCII operator. Only the first byte of the argument is evaluated.

The following line assigns the value 66 to the integer **my_ord**:

```
LET my_ord = ORD ("Querix")
```

This built-in function is case sensitive; if the first character in its argument is an uppercase letter, ORD() returns a value different from that which it would return if its argument had begun with a lowercase letter.

Sample Code: ord_function.4gl

```
MAIN

    DEFINE input_string CHAR(500)

    CALL fgl_winprompt(5, 2, "Enter a string to count its lengths (max. 40)", "",
40, 0) returning input_string

    DISPLAY "*** ORD() Function Example ***" AT 1,5

    DISPLAY "Your specified string: " , input_string at 5,5
    DISPLAY "The value of the first byte ord(input_string) is: ", ord(input_string)
at 7,5

    CALL fgl_winmessage("Exit","Press any key to close this demo application","info")
END MAIN
```

References

ASCII, fgl_keyval()

sfmt()

The SFMT() operator can be used to pass a string with variable parameters in it. The position of the parameters within the string is defined by placeholders that are preceded by a percent (%) symbol. If the percent symbol needs to be displayed to the SFMT() string, it should go as a double percent combination (%%). The syntax of the SFMT() operator is as follows:

```
SFMT("string %1 [string %2 string %3... etc]", parameter1[,parameter2, parameter3...])
```

In the scheme above, the value returned by parameter1 will be passed to the %1 placeholder, the value returned by the parameter2 will be passed to the %2 placeholder, etc.

You can use one and the same placeholder for several times within the SFMT() string. This will make the same value be passed to several positions.

The parameters passed to the SFMT() operator can be represented by any valid expressions. Date and numeric expressions are converted into string according to the current DBDATE and DBMONEY settings.

Below is given an example of the operator performance:

```
DEFINE

    ct VARCHAR(20)

    LET ct = "London"

DISPLAY sfmt("Today is %1. The current time in %2 is %3", DATE,ct,TIME)
```

On the Wednesday, the 13th of June, the returned string would be:

```
Today is Wed Jun 19 2013. The current time in London is 18:15:24
```

There are also pre-defined placeholders that can be used during error handling within WHENEVER ERROR CALL/GOTO statements. These placeholders are listed in the table below:

Predefined Placeholder	Description
%(ERRORFILE)	Get the name of the module in which the last runtime error occurred
%(ERRORLINE)	Get the number of the line in the module where the last runtime error occurred
%(ERRNO)	Get the operation system number of the last error
%(STRERROR)	Get the operation system text of the last error

space

The SPACE operator is preceded by an integer expression and returns a string of a specified length, containing only blank (ASCII 32) characters. Either of the keywords SPACE or SPACES can be used.

Usage

This operator returns a blank string of a length corresponding to its positive integer argument, specifying a relative offset. The returned value is identical to a quoted string that contains the same number of blank spaces.

In a PRINT statement in the FORMAT section of a report definition, SPACE advances the character position by the specified number of characters.

The following example program assigns to the string variable my_space_string three characters ("-->"), 10 spaces using (5+5 SPACES) and another three character string ("<--"). We could also have used (10 SPACES).

Sample Code: space_function.4gl

```
MAIN
  DEFINE my_spaces_string char(80)

  LET my_spaces_string = "-->", (5+5 SPACES), "<--"

  DISPLAY "*** SPACES Example ***" AT 1,5
  DISPLAY "my_spaces_string = \"-->\", (5+5 SPACES), \"
  \<--\"" at 3,1
  DISPLAY " 1234567890" at 5,1
  DISPLAY my_spaces_string at 6,1

  CALL fgl_winmessage("Exit","Press any key to close this demo application","info")
END MAIN
```

The following statements from a fragment of a report definition use the SPACE operator to accomplish several tasks:

- To separate variables within two PRINT statements
- To concatenate eight blank spaces to the string "=ZIP"
- To print the resulting string after the value of the variable cont_zip

FORMAT


```
ON EVERY ROW
    LET mystring = (8 SPACES), "=ZIP"
    PRINT cont_fname, 2 SPACES, cont_lname
    PRINT comp_name
    PRINT comp_addr1
    PRINT comp_city, ", " , comp_zone, \
    2 SPACES, comp_zip, \ mystring
```

In a DISPLAY statement, the SPACE operator inserts the specified number of blank characters into the output.

Outside PRINT statements, the SPACE (or SPACES) keyword and its operand must appear within parentheses, as in the LET statement of the previous example.

References

LINENO, PAGENO

Substring ([])

The substring operator ([]) is preceded by a character expression and specifies a substring of the value returned by that character expression.

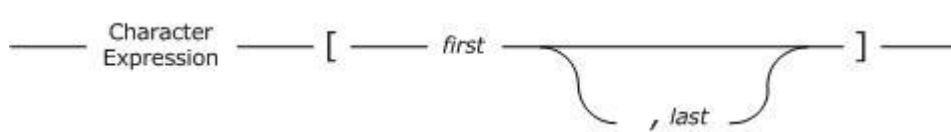


Figure 47 - Substring Operator

Usage

The square brackets will enclose one or two comma-separated operands. If the second operand is not specified, the single character in the *first* position is returned. If the *last* is specified, a comma must separate it from *first*, and a substring is returned whose first character is *first* and whose last character is *last*, including any intervening characters.

The integer expressions *first* and *last* must return values greater than zero. The value of *first* must be in the range from 1 to a figure no greater than *length*, where *length* is the length of the string returned by the character expression. Similarly *last* must be no less than *first* and no greater than *length*.

Sample Code: substring_operator.4gl

MAIN

```
DEFINE a, b CHAR(5), c ARRAY [3,4,5] OF CHAR(5)
```

```
LET a = "Smith"
```

```
LET b = a[2,4] -- mit
```

```
LET c[2,2,2] = a[3,4] --it
```

```
DISPLAY "*** Substring Example ***" AT 1,5
```

```
DISPLAY "LET a = \"Smith\"" AT 3,5
```

```
DISPLAY "a: " ,a at 4,5
```

```
DISPLAY "b = a[2,4]" AT 6,5
```

```
DISPLAY "b: " ,b at 7,5
```

```
DISPLAY "LET c[2,2,2] = a[3,4]" AT 9,5
```

```
DISPLAY "c: " ,c[2,2,2] at 10,5
```

```
CALL fgl_winmessage("Exit","Press any key to close this demo application","info")
```

```
END MAIN
```

Invalid Operands in Substring Expressions

Be careful to avoid specifying invalid operands for the substring ([]) operator, as in the following cases:

- When first has a zero or negative value
- When first is larger than last
- When first or last cannot be converted to an integer value
- When last has a value greater than the number of bytes returned by the left-hand character expression
- When the left-hand expression is a CHAR or VARCHAR variable (or ARRAY element, or RECORD member) of a declared size less than last
- When the left-hand character expression returns an empty string

In standard 4GL invalid operands can produce runtime error -1332. Querix4GL, however, will allow a null string to be returned, rather than produce an error message.

In some locales the substring operator is byte based. In East Asian locales that support multibyte characters, 4GL automatically replaces any partial characters that this operator attempts to create with single-byte white-space characters so that no partial character is returned.

Reference

CLIPPED

trim()

The trim() function in Querix4GL enables the developer to remove any whitespace (any characters that represents a space) from the beginning and the end of a string.

Usage

The trim() function takes an input string as its argument and removes all whitespace from the beginning and the end of that string. If the string is a sentence with spaces between words then the trim() function will leave the whitespace between the words:

```
trim(' The cat sat on the mat ')  
trim(' Spaceship ')
```

Would return the strings:

```
'The cat sat on the mat'  
'Spaceship'
```

Adding the two strings together would produce the following string:

```
'The cat sat on the matSpaceship'
```

upshift()

The UPSHIFT() function takes a character-string argument and returns a string in which any lowercase letters are converted to uppercase letters.

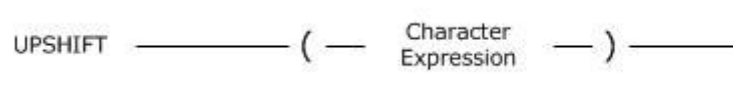


Figure 48 - UPSHIFT() Function

Usage

The UPSHIFT() function is most often used to regularize data; for example, to prevent the state abbreviation VA, Va, or va from resulting in different values if these abbreviations were logically equivalent in the context of your application.

You can use the UPSHIFT() function in an expression where a character string is valid, in DISPLAY and PRINT statements, and in assignment statements.

Non-alphabetic and uppercase characters are not altered by UPSHIFT(). The maximum data length of the argument (and of the returned character string value) is 32,766 bytes.

The following code sample shows how to convert a string to upper and lower case.

Code Sample: upshift_function1.4gl

```
MAIN

    DEFINE input_string char(50)

    DISPLAY "*** UPSHIFT / DOWNSHIFT Example ***" AT 1,5

    CALL fgl_winprompt(5,5,"Enter a string - max 50 characters", "not-specified", 50,
0) returning input_string

    #convert to lower case
    CALL fgl_winmessage("DOWNSHIFT converts to lowercase", DOWNSHIFT(input_string),
"info")

    #convert to upper case
    CALL fgl_winmessage("UPSHIFT converts to uppercase", UPSHIFT(input_string),
"info")

END MAIN
```

The following example demonstrates a function that was written to merge two privilege strings. Its output preserves letters in preference to hyphens (privileges over lack of privilege) and uppercase letters in preference to lowercase (privileges WITH GRANT OPTION over those without).

Code Sample: upshift2_function.4gl

```
MAIN
  DEFINE input_string1 CHAR(20)
  DEFINE count SMALLINT

  DISPLAY "Merge 2 strings and convert them to upper & lowercase" AT 1,5

  CALL fgl_winprompt(5,5,"Enter your input string - max 20 characters", "not-
specified", 20, 0)
  RETURNING input_string1

  CALL convert_string(input_string1)
  RETURNING input_string1, count

  DISPLAY "Your string in uppcase: ", input_string1
  DISPLAY count USING "<&", " characters were converted"

  SLEEP 5
END MAIN

FUNCTION convert_string(first_string)
  DEFINE first_string char(20)
  DEFINE k SMALLINT
  DEFINE count SMALLINT

  LET count = 0

  FOR k = 1 TO LENGTH(first_string)
    IF UPSHIFT(first_string[k]) <> first_string[k] THEN
      LET count = count + 1
      LET first_string[k] = UPSHIFT (first_string[k] )
    END IF
  END FOR

  RETURN first_string, count
END FUNCTION
```

In the next example, the CHAR variables **up_str** and **str** are equivalent, except that **up_str** substitutes uppercase letters for any lowercase letters in my_string:

```
LET up_str = UPSHIFT(my_string)
```

UPSHIFT() has no effect on non-English characters in most multibyte locales. In multibyte locales, UPSHIFT() and DOWNSHIFT() treat the first partial (or otherwise invalid) character in the argument as if it terminated the string. For example, suppose that € is an invalid character. The following expression would return the character string "ABCD EF " with any single-byte or multibyte white-space characters that immediately precede the first invalid character being included in the returned value, rather than being discarded:

```
UPSHIFT(ABCD ef €XYZ)
```

The UPSHIFT field attribute is discussed in the volume about "Screen Forms and Reports".

Reference

DOWNSHIFT()

using

The USING operator specifies a character-string format for a number, MONEY, or DATE operand and returns the formatted value.



Figure 49 - USING Operator

Usage

With a number or MONEY operand, you can use the USING operator to align decimal points or currency symbols, to right- or left-align numbers, to put negative numbers in parentheses, and to perform other formatting tasks. USING can also convert a DATE operand to a variety of formats.

USING is typically used in DISPLAY and PRINT statements, but you can also use it with LET to assign the formatted value to a character variable. If a value is too large for the field, 4GL fills it with asterisks (*) to indicate an overflow.

Symbols that the USING operator recognizes in *format-string* are described in "The USING Formatting Symbols for Number Values" section and the "Formatting DATE Values" section of this volume for number values and DATE values respectively.

Formatting Number Expressions

The USING operator takes precedence over the **DBMONEY** or **DBFORMAT** environment variables and is required to display the thousands separator of **DBFORMAT**. When 4GL displays a number value, it follows these rules:

- 4GL displays the leading currency symbol (as set by **DBFORMAT** or **DBMONEY**) for MONEY values. (But if the FORMAT attribute also specifies a leading currency symbol, 4GL displays that symbol for other data types.)
- 4GL omits the thousands separators, unless they are specified by a FORMAT attribute or by the USING operator.
- 4GL displays the decimal separator, except for INT or SMALLINT values.
- 4GL displays the trailing currency symbol (as set by **DBFORMAT** or **DBMONEY**) for MONEY values unless you specify a FORMAT attribute or the USING operator. In this case, the user cannot enter a trailing currency symbol, and 4GL does not display it.


The USING Formatting Symbols for Number Values

The format-string value can include the following characters.

Character	Description
*	The asterisk character will cause any spaces in the display field that would normally be blank, to be filled with asterisks
&	The ampersand character will cause any spaces that would normally be blank to be filled with zeros.
#	The hash character does not have any effect on blank spaces; it can be used to specify the maximum width of a field.
<	The greater than symbol causes numbers in a field to be aligned to the left-hand edge.
,	The comma is a literal character. USING displays it as a comma when there is a number to the left of it.
.	The period is a literal character. USING displays it as a period. Only one period can appear in a number format string, where it is used as the decimal point.
-	The hyphen is a literal character. USING displays it as a minus sign, but it only appears when the expression it relates to is a number with a negative value. When several hyphens are placed in a row, only one is displayed. This floats one character space to the left of the number being printed.
+	The plus sign is a literal character. USING displays it as a plus sign when the expression it relates to is a number that is zero or greater. When the number is less than zero it automatically displays as a minus sign. When several plus signs are placed in a row, only one is displayed. This floats one character space to the left of the number being printed.
\$	The dollar symbol is displayed as a literal character. When several dollar signs are placed in a row, only one is displayed. This floats one character space to the left of the number being printed.

(The left parenthesis symbol is displayed before a negative number. It is known as the accounting parenthesis because in accounting negative numbers are displayed within parentheses, rather than being preceded by a minus sign.
)	The right parenthesis symbol is used to close a negative number when the left parenthesis is used at the beginning to indicate a negative number.

The minus sign (-), plus sign (+), parentheses, and dollar sign (\$) *float*, meaning that when you specify multiple leading occurrences of one of these characters, 4GL displays only a single character immediately to the left of the number that is being displayed. Any other character in *format-string* is interpreted as a literal.

	These characters are not identical to the formatting characters that you can specify in the format-strings of the FORMAT or PICTURE field attributes, described in the volume that covers "Screen Forms and Reports".
---	---

For examples of using format strings for number expressions, see "Examples of the USING Operator" further on in this section. Because format strings interact with data to produce visual effects, you might find that the examples are easier to follow than the descriptions on the previous page of USING format string characters.

The following example prints a MONEY value using a format string that allows values up to \$9,999,999.99 to be formatted correctly:

Code Sample: using1_operator.4gl

```
MAIN
DEFINE mon_val1, mon_val2 MONEY(8,2)
  LET mon_val1 = 43243.18
  LET mon_val2 = -18589.57

  DISPLAY "*** USING Example 1 ***" AT 1,5

  DISPLAY "43243.18 - $#,###,##&.&& - The current balance is :", mon_val1
  USING "$#,###,##&.&&" at 3,5

  DISPLAY "43243.18 - +$#,###,##&.&& - The current balance is :", mon_val1
  USING "+$#,###,##&.&&" at 4,5

  DISPLAY "-18589.57 - -$#,###,##&.&& - The current balance is :", mon_val2
  USING "-$#,###,##&.&&" at 5,5

  DISPLAY "-18589.57 - $#,###,##&.&& - The current balance is :", mon_val2
  USING "$#,###,##&.&&" at 6,5

  DISPLAY "-18589.57 - $(#,###,##&.&&) - The current balance is :", mon_val2
  USING "$(#,###,##&.&&)" at 7,5

  DISPLAY "-18589.57 - €('#'###'##&.&&) - The current balance is :", mon_val2
  USING "€('#'###'##&.&&)" at 8,5

  DISPLAY "43243.18 - $<<, <<<, <<&.&& - The current balance is :", mon_val1
  USING "€<<, <<<, <<&.&&" at 9,5

  CALL fgl_winmessage("Exit", "Press any key to close this demo
  application", "info")
END MAIN
```

The above examples also use the # and & fill characters. The # character provides blank fill for unused character positions, while the & character provides zero filling. This format ensures that even if the number is zero, any positions marked with & appear as zero, not blank.

Dollar signs can be used instead of # characters, as in the following statement:

```
DISPLAY "Total amount due: ", inv_total
  USING "$$, $$$, $$$&.&&"
```

In this example, the currency symbol floats with the size of the number so that it appears immediately to the left of the most significant digit in the display. This example would produce the following formatted output, if the value of the **inv_total** variable were 14187.00:

```
Total amount due: $14,187.00
```

By default, 4GL displays numbers right aligned. You can use the < symbol in a USING format string to override this default. For example, specifying

```
DISPLAY "  
Total amount due: ", inv_total  
USING "$<<, <<<, <<&.&&"
```

produces the following output when the value of **inv_total** is 14187.00:

```
The current balance is $14,187.00
```

Formatting DATE Values

When you use it to format a DATE value, USING takes precedence over any **DBDATE** or **GL_DATE** environment variable settings. The *format-string* value for a date can be a combination of the characters m, d, and y.

Symbols	Resulting Time Unit in Formatted DATE Display
dd	Day of the month, from 01 up to 31, as appropriate.
ddd	Day of the week, as a three letter abbreviation.
mm	Month of the year, from 01 to 12, as appropriate.
mmm	Month of the year, as a three letter abbreviation.
yy	The year of the century, from 00 to 99, as appropriate.
yyyy	The full numerical year description, from 0001 to 9999.

In these instances lowercase letters must be used; uppercase will not be recognized.

Any other characters within a USING formatting mask for DATE values are interpreted as literals.

The following examples show valid format-string masks for December 25, 1999, and the resulting display for the default U.S. English locale.

Format String	Formatted Result
"mmddyy"	122599
"ddmmyy"	251299
"yyymmdd"	991225
"yy/mm/dd"	99/12/25
"yy mm dd"	99 12 25
"yy-mm-dd"	99-12-25
"mmm. dd, yyyy"	Dec. 25, 1999
"mmm dd yy"	Dec 25 1999
"yyyy dd mm"	1999 25 12
"mmm dd yyyy"	Dec 25 1999
"ddd, mmm. dd, yyyy"	Sat, Dec. 25, 1999
"(ddd) mmm. dd, yyyy"	(Sat) Dec. 25, 1999

This example programs demonstrates all different date formats:

Code Sample: using2_operator.4gl

```

MAIN
  DEFINE my_date DATE, inp_char char(1)
  LET my_date = today

  DISPLAY "*** USING Example 2 ***" AT 3,5

  DISPLAY "using mmddyy:           ", my_date USING "mmddyy" at 5,5
  DISPLAY "using ddmmyy:          ", my_date USING "ddmmyy" at 6,5
  DISPLAY "using yyymmdd:         ", my_date USING "yyymmdd" at 7,5
  DISPLAY "using yy/mm/dd:        ", my_date USING "yy/mm/dd" at 8,5
  DISPLAY "using yy mm dd:        ", my_date USING "yy mm dd" at 9,5
  DISPLAY "using yy-mm-dd:        ", my_date USING "yy-mm-dd" at 10,5
  DISPLAY "using mmm. dd, yyyy:    ", my_date USING "mmm. dd, yyyy" at 11,5

```

```
DISPLAY "using mmm dd yyy:           ", my_date USING "mmm dd yyy" at 12,5
DISPLAY "using yyyy dd mm:          ", my_date USING "yyyy dd mm" at 13,5
DISPLAY "using mmm dd yyyy:         ", my_date USING "mmm dd yyyy" at 14,5
DISPLAY "using ddd, mmm. dd, yyyy:  ", my_date USING "ddd, mmm. dd, yyyy" at 15,5
DISPLAY "using mmm dd yyyy:         ", my_date USING "mmm dd yyyy" at 16,5

Prompt "Press any key to close this demo application" FOR inp_char
END MAIN
```

Using non-default locales, the NUMERIC and MONETARY categories in the locale files affect how the format string of the USING operator is interpreted for formatting number and currency data values.

In *format string*, the period (.) is not a literal character but a placeholder for the decimal separator specified by environment variables. Likewise, the comma is a placeholder for the thousands separator specified by environment variables. The dollar sign (\$) is a placeholder for the leading currency symbol. The @ symbol is a placeholder for the trailing currency symbol. Thus, the format string \$#,###.## formats the value 1234.56 as £1,234.56 in a U.K. English locale, but as €1.234,56 in a French locale. Setting either **DBFORMAT** or **DBMONEY** overrides these locale settings.

The mmm and ddd specifiers in a format string can display language-specific month-name and day-name abbreviations. This needs the LANG environment variable to be set, and the appropriate locale to be installed on your local operating system.

Examples of the USING Operator

Tables that follow illustrate some of the capabilities of the USING operator with number or currency operands (for the default U.S. English locale). Each table has the following format:

- The first column shows a format string (the left-hand operand).
- The second column shows a data value (the right-hand operand).
- The third column shows the resulting formatted display.
- The fourth column provides a comment (for some rows).

In the following table, the character b in the **Formatted Result** column represents a blank space.

Format String	Data Value	Formatted Result	Comment on Result
"#####"	0	bbbbbb	No zero symbol
"&&&&&"	0	00000	
"\$\$\$\$\$"	0	bbbb\$	No zero symbol
"*****"	0	*****	No zero symbol
"<<<<<"	0		(NULL string)
"<<<, <<<"	12345	12,345	
"<<<, <<<"	1234	1,234	
"<<<, <<<"	123	123	
"<<<, <<<"	12	12	
"##,###"	12345	12,345	
"##,###"	1234	b1,234	
"##,###"	123	bbb123	
"##,###"	12	bbbb12	
"##,###"	1	bbbbbb1	
"##,###"	-1	bbbbbb1	No negative sign
"##,###"	0	bbbbbbb	No zero symbol

Format String	Data Value	Formatted Result	Comment on Result
"&&, &&&"	12345	12,345	
"&&, &&&"	1234	01,234	
"&&, &&&"	123	000123	
"&&, &&&"	12	000012	
"&&, &&&"	1	000001	
"&&, &&&"	-1	000001	No negative sign
"&&, &&&"	0	000000	
"&&, &&&. &&"	12345.67	12,345.67	
"&&, &&&. &&"	1234.56	01,234.56	
"&&, &&&. &&"	123.45	000123.45	
"&&, &&&. &&"	0.01	000000.01	
"\$,\$,\$,\$"	12345	*****	(Overflow)
"\$,\$,\$,\$"	1234	\$1,234	
"\$,\$,\$,\$"	123	bb\$123	
"\$,\$,\$,\$"	12	bbb\$12	
"\$,\$,\$,\$"	1	bbbb\$1	
"\$,\$,\$,\$"	0	bbbbb\$	No zero symbol
"** , ***"	12345	12,345	
"** , ***"	1234	*1,234	
"** , ***"	123	***123	
"** , ***"	12	****12	
"** , ***"	1	*****1	
"** , ***"	0	*****	No zero symbol

In the following table the character b in the **Formatted Result** column represents a blank space.

Format String	Data Value	Formatted Result	Comment on Result
"###,###.###"	12345.67	12,345.67	
"###,###.###"	1234.56	b1234.56	
"###,###.###"	123.45	bbb123,45	
"###,###.###"	12.34	bbbb12.34	
"###,###.###"	1.23	bbbbb1.23	
"###,###.###"	0.12	bbbbb0.12	
"###,###.###"	0.01	bbbbbb0.01	No leading zero
"###,###.###"	-0.01	bbbbbb0.01	No negative sign
"###,###.###"	-1	bbbbb1.00	No negative sign
"\$\$\$\$. \$"	12345.67	*****	(overflow)
"\$\$\$\$. \$"	1234.56	\$1,234.56	
"\$\$\$\$.###"	0.00	\$.00	No leading zero
"\$\$\$\$.###"	1234.00	\$1,234.00	
"\$\$\$\$.&&"	0.00	\$.00	No leading zero
"\$\$\$\$.&&"	1234.00	\$1,234.00	
"-\$\$\$\$.&&"	-12345.67	-\$12,345.67	
"-\$\$\$\$.&&"	-1234.56	-b\$1,234.56	
"-\$\$\$\$.&&"	-123.45	-bbb\$123.45	
"--\$\$\$\$.&&"	-12345.67	-\$12,345.67	
"--\$\$\$\$.&&"	-1234.56	-\$1,234.56	
"--\$\$\$\$.&&"	-123.45	-bb\$123.45	
"--\$\$\$\$.&&"	-12.34	-bbb\$12.34	
"--\$\$\$\$.&&"	-1.23	-bbbbb\$1.23	

Format String	Data Value	Formatted Result	Comment on Result
"-##,###.##"	-12345.67	-12,345.67	
"-##,###.##"	-123.45	-bbb123.45	
"-##,###.##"	-12.34	-bbbb12.34	
"--#,###.##"	-12.34	-bbb12.34	
"---,###.##"	-12.34	-bb12.34	
"---,-#.##"	-12.34	-12.34	
"---,--#.##"	-1.00	-1.00	
"-##,###.##"	12345.67	12,345.67	
"-##,###.##"	1234.56	1,234.56	
"-##,###.##"	123.45	123.45	
"-##,###.##"	12.34	12.34	
"--#,###.##"	12.34	12.34	
"---,###.##"	12.34	12.34	
"---,-#.##"	12.34	12.34	
"---,---.##"	1.00	1.0	
"---,---.---"	-.01	-0.01	
"---,---.&&"	-.01	-0.01	

Here the character b in the **Formatted Result** column represents a blank space.

Format String	Data Value	Formatted Result	Comment on Result
"----,--\$.&&"	-12345.67	-\$12,345.67	
"----,--\$.&&"	-1234.56	-\$1234.56	
"----,--\$.&&"	-123.45	-\$123,45	
"----,--\$.&&"	-12.34	-\$12.34	
"----,--\$.&&"	-1.23	-\$1.23	
"----,--\$.&&"	-.12	-\$12	
"\$***,***.&&"	12345.67	\$*12,345.67	
"\$***,***.&&"	1234.56	\$**1,234.56	
"\$***,***.&&"	123.45	\$***123.45	
"\$***,***.&&"	12.34	\$*****12.34	
"\$***,***.&&"	1.23	\$*****1.23	
"\$***,***.&&"	.12	\$*****.12	
"(\$\$\$,\$\$\$.&&)"	-12345.67	(\$12,345.67)	Accounting parentheses
"(\$\$\$,\$\$\$.&&)"	-1234.56	(b\$1,234.56)	
"(\$\$\$,\$\$\$.&&)"	-123.45	(bb\$123.45)	
"((\$\$,,\$\$\$.&&)"	-12345.67	(\$12,345.67)	
"((\$\$,,\$\$\$.&&)"	-1234.56	(b\$1,234.56)	
"((\$\$,,\$\$\$.&&)"	-123.45	(bb\$123.45)	
"((\$\$,,\$\$\$.&&)"	-12.34	(bbb\$12.34)	
"((\$\$,,\$\$\$.&&)"	-1.23	(bbbb\$1.23)	
"(((,((\$.&&)"	-12345.67	(\$12345.67)	
"(((,((\$.&&)"	-1234.56	(\$1234.56)	
"(((,((\$.&&)"	-123.45	(\$123.45)	
"(((,((\$.&&)"	-12.34	(\$12.34)	
"(((,((\$.&&)"	-1.23	(\$1.23)	
"(((,((\$.&&)"	-.12	(\$12)	
"(\$\$\$,\$\$\$.&&)"	12345.67	\$12,345.67	
"(\$\$\$,\$\$\$.&&)"	1234.56	\$1,234.56	
"(\$\$\$,\$\$\$.&&)"	123.45	\$123.45	
"((\$\$,,\$\$\$.&&)"	12345.67	\$12,345.67	
"((\$\$,,\$\$\$.&&)"	1234.56	\$1,234.56	
"((\$\$,,\$\$\$.&&)"	123.45	\$123.45	
"((\$\$,,\$\$\$.&&)"	12.34	\$12.34	
"((\$\$,,\$\$\$.&&)"	1.23	\$1.23	
"(((,((\$.&&)"	12345.67	\$12,345.67	

"(((,((\$.&&)"	1234.56	\$1,234.56	
"(((,((\$.&&)"	123.45	\$123.45	
"(((,((\$.&&)"	12.34	\$12.34	
"(((,((\$.&&)"	1.23	\$1.23	
"(((,((\$.&&)"	.12	\$.12	

A third sample code example has been prepared to show how various values would be displayed using the USING operator to define different output formats. This can be found within the 'functions' demonstration application packaged with your Lycia installation. It is the program called "using3_operator.exe". If you want to see the program working, or the code in the .4gl file please look at the demonstration application using Lycia.

Project: functions

Program: using3_operator.exe

wordwrap

The WORDWRAP operator divides a long text string into segments that appear in successive lines of a 4GL report. (This operator can appear only in the PRINT statement in the FORMAT section of a REPORT program block.)

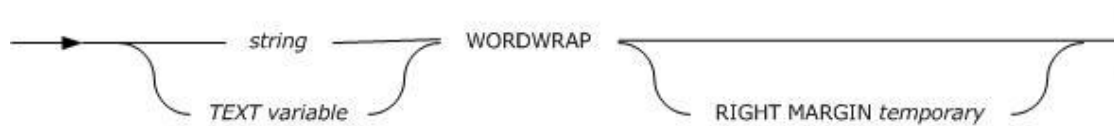


Figure 50 - WORDWRAP Operator

The *string* variable is a character expression that will be printed in the report output. The *TEXT variable* is the name of a TEXT datatype variable that is also to be printed in the report output. The *temporary* variable is an integer expression, which returns a value which specifies the absolute position, from the left margin, of a temporary right-hand margin; measured in character units.

Usage

The WORDWRAP operator automatically *wraps* successive segments of long character strings onto successive lines of output from a 4GL report. The string value of any expression or TEXT *variable* that is too long to fit between the current character position and the specified or default 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 *temporary*, the right margin defaults to 132 or to the size from the RIGHT MARGIN clause of the OUTPUT section of the report definition.

These temporary values override the specified or default left and right margins from the OUTPUT section.

After the PRINT statement has executed, any explicit or default margins from the OUTPUT section are restored. "PRINT" is discussed in more detail in the 4GL Statements volume of this guide.

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 4GL variable called **mynovel**:

```
print column 10, mynovel WORDWRAP RIGHT MARGIN 70
```

Tabs, Line Breaks, and Page Breaks with WORDWRAP

The data string can include printable ASCII characters. It can also include the TAB (ASCII 9), new line character (ASCII 10), and RETURN (ASCII 13) characters that partition the string into *words*, consisting of substrings of other printable characters. Other non-printable characters might cause runtime errors. If the data string cannot fit between the margins of the current line, 4GL breaks the line at a *word* division, padding the line with blanks at the right.


From left to right, 4GL 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, 4GL takes these actions:

- Prints blank characters only to the right margin
- Discards any remaining blank characters from the blank string or tab
- Starts a new line at the temporary left margin
- Processes the next word

4GL 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, an even left margin results. 4GL applies the following rules (in descending order of precedence) to the portion of the data string within the right margin:

- Break at any new line character, RETURN, or new line character and RETURN 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, RETURN, TAB, or new line character.

4GL maintains page discipline with the WORDWRAP operator. If the character string or TEXT value operand is too long for the current page of report output, 4GL executes the statements in any PAGE TRAILER and PAGE HEADER control blocks before continuing output onto a new page.

	The WORDWRAP keyword can also be used to specify a field attribute that supports data display and data entry in a multi-segment field within a 4GL form.
---	--

The following example formats a long character variable by wrapping it up to column 60 of the generated 4GL report.

Sample Code: wordwrap_report_function.4gl

```
MAIN
  START REPORT repl TO SCREEN
  OUTPUT TO REPORT repl ( )
  FINISH REPORT repl
END MAIN

REPORT repl()
  DEFINE long_string CHAR(256)

  FORMAT
    ON EVERY ROW
      LET long_string = "This is a long string of text data which will wrap neatly
in the report output",
        " due to the use of the 'WORDWRAP' operator in 4GL\n\n",
        "\tPRINT string_var WORDWRAP [ RIGHT MARGIN int_var ]"
      PRINT long_string WORDWRAP RIGHT MARGIN 60
END REPORT
```

References

CLIPPED, SPACES, USING

String methods

Besides the functions, there is also a number of methods that can be used to manipulate variables of STRING data type.

append()

The Append() method is used to add a new string to the end of the current string value. The method returns a string value, consisting of the initial string concatenated with the newly specified one:

```
DEFINE str, str_app STRING
LET str = "My string"
LET str_app = str.Append(" with an appended part")
DISPLAY str
DISPLAY str_app -- Displays "My string with an appended part"
```

equals()

The Equals() method is used to compare the current string to another string value. The method returns an integer value, which is 1 (TRUE) if the compared values are equal and 0 (FALSE) if not:

```
DEFINE pass, pass_ch STRING
DEFINE equa INT
...

LET equa = pass.Equals(pass_confirm)
```

The method is case sensitive

equalsIgnoreCase()

The EqualsIgnoreCase() is similar to Equals() method, but is not case sensitive. It is used to compare the current string to another string value. The method returns an integer value, which is 1 (TRUE) if the compared values are equal and 0 (FALSE) if not:

```
DEFINE city, cust_city STRING
DEFINE equa INT
...

LET equa = cust_city.Equals(city)
```


getCharAt()

The GetCharAt() method is used to return a character that takes the specified position in the string variable value. The method needs an integer argument that specifies the position. Here, 1 stands for the first character in the buffer sting. If the specified character does not exist, the method returns NULL.

```
LET mystr = "My string variable value"
DISPLAY mystr.GetCharAt(5) -- displays "t"
DISPLAY mystr.GetCharAt(3) -- displays a whitespace " "
```

Note, that the GetCharAt() method is based on byte-length semantics. The method used in a multi-byte environment takes the argument not as a character position, but as a parameter. An invalid multi-byte position will make the method return a blank.

getIndexOf()

The GetIndexOf() method is used to return an integer number which identifies the position of a specified substring. The method needs two arguments:

```
String_var.GetIndexOf(substring, position)
```

The *substring* argument specifies a quoted string or a variable containing the substring that will be searched for.

The *position* argument is an integer value indicating the position in the buffer string at which the search should begin. To start the search from the very beginning of the buffer string, specify the position argument as 1. The following extract of a source code will return "5":

```
DEFINE pos INTEGER
CALL string_val.Append("a23456789b")
LET pos = string_val.GetIndexof("567",1)
DISPLAY "567 starts from the ", pos, " position"
```

Note that, in a multi-byte environment the GetIndexOf() method will return the byte position instead of the character position.

getLength()

The GetLength() method is used to return an integer value identifying the total number of bytes in the string. Trailing spaces are also taken into account. If the string is empty, the method will return 0. In non multi-byte environment the number of bytes corresponds to the number of characters in the string. This method accepts no parameters.

The example of the method invocation is given below:

```
LET ln = string_val.GetLength()
```

The `GetLength()` method used in a multi-byte environment returns not the number of characters in the string, but the number of bytes.

subString()

The `SubString()` method is used to retrieve a substring from a string variable value. The two arguments required by the method are integers that identify the beginning and the ending positions of the substring. If the specified positions are invalid, the method will return NULL.

In the extract below, the `SubString()` method will return a substring "long" which begins at the 3rd and ends at the 6th position:

```
DEFINE substr STRING
CALL string_val.Append("A long string")
LET substr = string_val.subString(3,6)
```

The `SubString()` method used in a multi-byte environment will treat the specified positions as parameters. If the multi-byte positions are invalid, the method will return blanks.

toLowerCase()

The `ToLowerCase()` method is used to convert the characters of the string value to the lower case. The lower case characters, symbols and numbers will not be converted. The example of the method usage is given below:

```
DISPLAY string_val.ToLowerCase()
```

The method does not need any arguments and does not return anything.

If the string is NULL, the method returns NULL

toUpperCase()

The `ToUpperCase()` method is used to convert the characters of the string to the upper case. The upper case characters, symbols and numbers will not be converted. The example of the method usage is given below:

```
DISPLAY string_val.ToUpperCase()
```

The method does not need any arguments and does not return anything.

If the string is NULL, the method returns NULL

trim()

The `Trim()` method is used to remove leading and trailing spaces from a STRING variable value. Its effect is the same as of the `trim()` built-in function. The method does not need any arguments.

```
LET string_value = "   Some STRING variable text   "
```

```
CALL string_value.Trim()
```

The string_value STRING variable will get the value "Some STRING variable text" after the Trim() method is applied.

If the string is NULL, the method returns NULL

trimLeft()

The TrimLeft() method is used to remove only the leading spaces from the string value and leaves the trailing spaces intact.

If the string is NULL, the method returns NULL

trimRight()

The TrimRight() method is used to remove only the trailing spaces from the string value, leaving any leading spaces untouched.

If the string is NULL, the method returns NULL

base.STRINGBUFFER Class

The StringBuffer class was introduced to provide the user with some additional abilities of manipulating character strings.

The StringBuffer class has a number of advantages. The objects of the StringBuffer class allow to manipulate directly the internal string buffer, whereas the system, when working with variables of STRING data type, needs to create new buffers for manipulating them. The difference is not noticeable in programs with a user interface or batch programs performing SQL operations. However, it can be seen when processing large character strings.

For example, processing of 700Kb of text will be performed faster with a StringBuffer object then with a variable of STRING data type.

The methods which can be applied to objects of StringBuffer class are based on byte-length semantics. The GetLength() method used in a multi-byte environment returns the number of bytes. This number can be different from the actual number of characters composing the object value.

create()

To create a StringBuffer object, you must first declare it with a DEFINE statement and then create with the Create() method used in a LET statement. The Create() is a class method, it should be called with the name of the class used as a prefix.

```
DEFINE stb base.StringBuffer
LET stb = base.StringBuffer.Create()
```

The StringBuffer object cannot be referenced by such statements as LET or DISPLAY directly and it cannot take part in the input. To display its value or manipulate it in 4GL expressions, you should use the ToString() method described below. To assign a string to this object you should also use the Append() method and not the LET statement.

append()

The Append() method is used to append a string to the existing internal string buffer. The method needs one argument which is the character string that is to be appended:

```
CALL stb.Append("string")
```

The line above will append the string given in quotes to the string buffer. If the buffer is empty, the string will be added to it so that all the next values will be appended to it.

clear()

The Clear() method is used to clear the specified string buffer. The method does not need any arguments to be specified. It removes any values stored in the object.

```
CALL stb.Clear()
```

toString()

The ToString() method is used to convert the string buffer into a STRING value. This method is helpful in displaying the value stored in a StringBuffer object or passing it to a variable of the STRING data type. The method does not need any arguments.

```
DEFINE my_string STRING
CALL stb.Append("Some text")
LET my_string = stb.ToString()
DISPLAY my_string -- displays "Some text"
```

Without this method it is impossible to use the value stored in this object in other 4GL expressions and statements.

equals()

The Equals() method is used to compare the value of a StringBuffer object with a specified string. The method is case sensitive; it returns TRUE if the compared values are equal and FALSE if not and accepts a single argument that is the character string with which the object will be compared. The syntax of the method invocation is:

```
stb.Equals(string)
```

The *string* argument can be represented by both a quoted string and a variable. Therefore, to compare values of two StringBuffer objects, you should use the ToString() method to convert one of them into a string.

Below is given an example of the Equals() method usage in a conditional statement:

```
MAIN
DEFINE stb, stb2 base.StringBuffer
LET stb = base.StringBuffer.Create()
CALL stb.Append("value")
LET stb2 = base.StringBuffer.create()
CALL stb2.Append ("value2")

#Comparing to a string
IF stb.equals("value") THEN
    DISPLAY "stb matches 'value'" -- this string will be displayed
ELSE
    DISPLAY "stb doesn't match 'value'"
END IF

#Comparing two StringBuffer objects
IF stb.equals(stb2.toString()) THEN
    DISPLAY "stb matches stb2"
ELSE
    DISPLAY "stb does not match stb2" -- this string will be displayed
END IF END MAIN
```

equalsIgnoreCase()

The EqualsIgnoreCase() method performs the same way the Equals() method does, but it is not case sensitive.

getCharAt()

The GetCharAt() method is used to return a character that takes the specified position in the string buffer value. The method requires an integer argument specifying the position. 1 stands for the first character in the buffer sting. If the specified character does not exist, the method returns NULL.

```
CALL stb.Append("My buffered string")
DISPLAY buf.GetCharAt(4) -- displays "b"
DISPLAY buf.GetCharAt(3) -- displays a whitespace " "
```

Note, that the GetCharAt() method is based on byte-length semantics. The method used in a multi-byte environment takes the argument not as a character position, but as a parameter. An invalid multi-byte position will make the method return a blank.

getIndexOf()

The GetIndexOf() method is used to return an integer number which identifies the position of a specified substring. The method needs two arguments:

```
Stringbuffer_object.GetIndexOf(substring, position)
```

The *substring* argument is a quoted string or a variable containing the substring that will be searched for.

The *position* argument is an integer value indicating the position in the buffer string at which the search should begin. To start the search from the very beginning of the buffer string, specify the position argument as 1. The following extract of a source code will return "5":

```
DEFINE p INTEGER
CALL stb.Append("a23456789b")
LET pos = stb.GetIndexof("567",1)
DISPLAY "567 starts from the ", p, " position"
```

Note that, in a multi-byte environment the GetIndexOf() method will return the byte position instead of the character position.

getLength()

The GetLength() method is used to return an integer value identifying the total number of bytes in the specified string buffer. Trailing spaces are also taken into account. If the string buffer is empty, the method will return 0. In non multi-byte environment the number of bytes corresponds to the number of characters in the string. This method accepts no parameters.

The example of the method invocation is given below:

```
LET ln = stb.GetLength()
```

The GetLength() method used in a multi-byte environment returns not the number of characters in the buffer string, but the number of bytes.

subString()

The SubString() method is used to retrieve a substring from a string buffer. The two arguments required by the method are integers that identify the beginning and the ending positions of the substring. If the specified positions are invalid, the method will return NULL.

In the extract below, the SubString() method will return a substring "long" which begins at the 3rd and ends at the 6th position:

```
DEFINE substr STRING  
CALL stb.Append("A long string")  
LET substr = buf.subString(3,6)
```

The SubString() method used in a multi-byte environment will treat the specified positions as parameters. If the multi-byte positions are invalid, the method will return blanks.

toLowerCase()

The ToLowerCase() method is used to convert the characters of the buffer string to the lower case. The lower case characters, symbols and numbers will not be converted. The example of the method usage is given below:

```
CALL stb.ToLowerCase()
```

The method does not need any arguments and does not return anything.

toUpperCase()

The ToUpperCase() method is used to convert the characters of the buffer string to the upper case. The upper case characters, symbols and numbers will not be converted. The example of the method usage is given below:

```
CALL stb.ToUpperCase()
```

The method does not need any arguments and does not return anything.

trim()

The Trim() method is used to remove leading and trailing spaces from the string buffer. Its effect is the same as of the trim() built-in function. The method does not need any arguments.

```
CALL stb.Append("  Some string buffer text  ")  
CALL stb.Trim()
```

The `stb StringBuffer` object will get the value "Some string buffer text" after the `Trim()` method is applied.

trimLeft()

The `TrimLeft()` method is used to remove only the leading spaces from the string buffer value and leaves the trailing spaces intact.

trimRight()

The `TrimRight()` method is used to remove only the trailing spaces from the string buffer value, leaving any leading spaces untouched.

replaceAt()

The `ReplaceAt()` method is used to replace a part of the string buffer value with another string. The method syntax is:

```
CALL stb.ReplaceAt(start, bytes, "string")
```

- The *start* argument should contain an integer value that indicates the starting position of the replacement.
- The *bytes* argument specifies the number of bytes which are to be replaced. It need not be equal to the number of bytes in the new string.
- The *string* is the string that is to be inserted instead of the specified substring.

Here is a simple example of the method usage:

```
CALL stb.Append("1234567")
CALL stb.ReplaceAt(2, 3, "qwerty")
DISPLAY stb.toString() -- displays "1qwerty567"
```

The `ReplaceAt()` method used in a multi-byte environment reads the integer arguments not as character positions, but as parameters. If the multi-byte positions are invalid, the method won't perform any replacement.

replace()

The `Replace()` method is used to replace the a substring within a sting buffer with another string. The method syntax is:

```
CALL stb.Replace("string", "new_string", occurences)
```

- The *string* argument identifies the substring which is to be replaced.
- The *new_string* argument specifies the string that should replace the selected substring.

- The *occurrences* argument identifies the number of the substring occurrences that should be replaced. To replace all the occurrences in the string buffer, use 0 as the argument.

The example of the method application is given below:

```
CALL stb.Append("CDEzxcCDEmnbCDEijk")
CALL stb.Replace("CDE", "!", 2)
DISPLAY stb.toString() -- displays "!zxc!mnbCDEijk"
```

insertAt()

The InsertAt() method is used to insert a string to the string buffer before the specified position.

The syntax of the method invocation is as follows:

```
CALL stb.InsertAt(position, "string")
```

The position argument specifies the position in the string buffer before which a new string should be inserted. The string argument specifies the string to be inserted.

```
CALL stb.Append("123456")
CALL stb.InsertInto(5, "aaa")
DISPLAY stb.toString() -- displays "1234aaa56"
```

When the InsertAt() method is used in a multi-byte environment, it reads the byte position not as a character position, but as a parameter. An invalid multi-byte position can result in a string, invalid in the current code set.

base.STRINGTOKENIZER Class

The StringTokenizer class was introduced to provide the possibility to split a string into tokens according to delimiters. The objects of the class should be declared and initialized before they can be used.

To declare a StringTokenizer object, use the DEFINE statement:

```
DEFINE tok base.StringTokenizer
```

create()

The Create() method is used to initialize a declared StringTokenizer object. This is a class method that requires two arguments:

```
base.StringTokenizer.Create(sting, delimiter)
```

The string argument contains a quoted string (or a variable containing a string value) which is to be parsed, including delimiters.

The delimiter argument is a quoted string or a variable specifying the symbol which will be treated as a delimiter. If you want the backslash to be a delimiter, you should double it in both the delimiter specification and the source string.

Here is an example:

```
LET tok = base.StringTokenizer.Create("aaa|AAA|zzz", "|")
```

The line above will create 3 tokens: "aaa", "AAA", and "zzz".

StringTokenizer object can use different delimiters in one string. Each delimiter can be only one character long. The list of the delimiters is specified in the delimiter argument of the Create() method:

```
LET tok = base.StringTokenizer.Create("aaa|AAA%zzz\\ZZZ", "|\\%")
```

The Create() method for StringTokenizer objects has several special rules:

- the empty tokens are not taken into account; leading, trailing and consequent delimiters between tokens do not change the number of tokens
- there are no escape characters for the delimiters
- NextToken() method can never return NULL strings

createExt()

The CreateExt() method is an extended version of the Create() method which provides the user with some additional parsing options. The method has the following syntax:

```
base.StringTokenizer.CreateExt(string, delimiter, esc, nulls)
```

If a delimiter symbol is passed to the `esc` parameter, it will be escaped during the string tokenizing.

When the `nulls` argument is set to `TRUE`, the empty tokens are not ignored. As the result, the `NextToken()` method can return a `NULL` string. The leading, trailing and consequent delimiters between tokens influence the number of resulting tokens.

countTokens()

The `CountTokens()` method is used to return the number of tokens which are still to be returned.

hasMoreTokens()

The `HasMoreTokens()` method is used to check whether the string has any tokens left. Returns `TRUE`, if it does, and `FALSE` if not.

nextToken()

The `NextToken()` method returns the next token from the string.

Here is an example of the methods application:

```
LET tok = base.StringTokenizer.createExt("||\\|aaa|bbc|", "|", "\\|", TRUE)
WHILE tok.hasMoreTokens()
  DISPLAY tok.nextToken()
END WHILE
```

In this example, `NULL` tokens will be taken into account and the backslash sign will be skipped.

The os.Path Methods

The os.Path methods allow to handle files and directories on the server side.

Their usage makes it possible to check, change or return different file/directory properties, check the path type, return or join its segments, copy, create, delete or rename files/directories and perform some other operations required to handle files and directories.

Take notice that some of the functions described below (such as *os.Path.chown* method) can be used on UNIX systems only.

os.Path.atime

The os.Path.atime method is returning the date and time of the latest access to the file specified. The syntax of the method is as follows:

```
CALL os.Path.atime(f_name) RETURNING atime
```

where:

- *f_name* = a STRING variable specifying the name of the file,
- *atime* = a STRING variable that stands for the date and time executed from the system clock when the file was accessed for the last time in the standard format YYYY-MM-DD HH:MM:SS

Usage

In the case of a failure, the function returns a NULL string.

os.Path.basename

The os.Path.basename method is used to return a string value standing for the last path element. The syntax of the method is as follows:

```
CALL os.Path.basename(file_name) RETURNING base_name
```

where:

- *file_name* = a STRING variable standing for the file name,
- *base_name* = a STRING variable standing for the last path element.

Usage

As the function invocation extracts the last path element of the path, passing "`\\root\\dir001\\file.txt`" as the parameter will return "`file.txt`", for example.

To remove the last component of the path, the os.Path.dirname method should be used.

os.Path.chdir

The os.Path.chdir changes the current working directory. The syntax of the method is as follows:

```
CALL os.Path.chdir(new_dir) RETURNING result
```

where:

- *new_dir* = a STRING variable standing for the directory to select,
- *result* = an INTEGER variable standing for TRUE in case the current directory is successfully selected, otherwise FALSE will be returned.

os.Path.chvolume

The os.Path.chvolume is used to change the actual working volume. The syntax of the method is as follows:


```
CALL os.Path.chvolume(new_volume) RETURNING result
```

where:

- *new_volume* = a variable of STRING data type standing for the volume to select as a current one,
- *result* = a variable of INTEGER data: TRUE when the working volume can be successfully changed and FALSE otherwise.

Here is an example of the os.Path.chvolume method calling:

```
#changes the current working volume to "C:\"  
CALL os.Path.chvolume ("C:\\") RETURNING result
```

	<p>As the backslash symbol is a default escape symbol which makes the compiler treat the symbol following it as a literal character and ignore its special functionality, to add a backslash symbol to a character string, you have to use a double backslash in the source code or four symbols in a row to add a double backslash:</p> <pre>DISPLAY "This is a backslash: \\" DISPLAY "This is a double backslash: \\\\"</pre>
---	--

os.Path.copy

The os.Path.copy function copies an existing file to create a new one. The syntax of the method is as follows:

```
CALL os.Path.copy(source_file, dest) RETURNING result
```

where:

- *source_file* = a string containing the path name to the file to copy,
- *dest* = a string containing the destination and the name of the newly created (copied) file,

- *result* = TRUE when the file has been copied successfully or FALSE otherwise.

Here is an example of the os.Path.copy method calling:

```
#copies the file "file_1.txt" from " C:\New folder" to " C:\" with the same name
CALL os.Path.copy("C:\\Files\\file_1.txt", "C:\\file_1.txt") RETURNING result1
#copies the file "file_1" from " C:\New folder" to " C:\" with the name changed to
#'file_2.txt'
CALL os.Path.copy("C:\\Files\\file_1.txt", "C:\\file_2.txt") RETURNING result2
```

os.Path.delete

The os.Path.delete method is used to delete a file or a directory. The syntax of the method is as follows:

```
CALL os.Path.delete(d_name) RETURNING result
```

where:

- *d_name* = a string containing the path name to the file or directory to be deleted,
- *result* = TRUE whether the deletion has been carried out successfully or FALSE otherwise.

Note that only empty directory can be deleted.

os.Path.dirclose

The os.Path.dirclose is used to close the directory referenced by the directory handle. The syntax of the method is as follows:

```
CALL os.Path.dirclose(dir_handle)
```

where:

- *dir_handle* = an integer standing for the directory handle of the directory to be closed.

os.Path.dirfmask

The os.Path.dirfmask defines a filter mask for diropen call. The syntax of its invocation is as follows:

```
CALL os.Path.dirfmask(filt_mask)
```

where:

- *filt_mask* = an INTEGER variable defining the filter mask.

Usage

This function calling defines the filter mask for any subsequent diropen.

The selection of all the kinds of directory entries is carried out by the diropen function by default. A filter mask is used to abridge the number of entries.

The `dirmask` function parameter is a combination of the bits described below:

- `0x01` = for excluding hidden files (`.*`),
- `0x02` = for excluding directories,
- `0x04` = for excluding symbolic links,
- `0x08` = for excluding regular files.

Here is an example of the `os.Path.dirmask` method calling:

```
#retrieves only regular files  
CALL os.Path.dirmask(1 + 2 + 4)
```

os.Path.dirname

The `os.Path.dirname` method is used to return all components of a path excluding the last one. The syntax of its invocation is as follows:

```
CALL os.Path.dirname(file_name STRING) RETURNING dir_name STRING
```

where:

- *file_name* = a STRING variable standing for the file name,
- *dir_name* = a STRING variable containing all the path elements except the last one.

Usage

As the function invocation removes the last path component, passing "`\root\dir001\file.txt`" as the parameter, for example, will result in "`\root\dir001`".

os.Path.dirnext

The purpose of `os.Path.dirnext` method is reading the next entry in the directory. The syntax of its invocation is as follows:

```
CALL os.Path.dirnext(dir_handle) RETURNING dir_entry
```

where:

- *dir_handle* = a variable of INTEGER data type that stands for the directory handle of the directory to be read,
- *dir_entry* = a variable of STRING data type that stands either for the name of the entry read or for NULL in case all of the entries have been read.

os.Path.diropen

The `os.Path.diropen` method is used to open a directory and to return an integer handle to it. The syntax of the method invocation is as follows:

```
CALL os.Path.diropen(dir_name) RETURNING dir_handle
```

where:

- *dir_name* = a STRING variable standing for the directory name,
- *dir_handle* = an INTEGER variable standing for the directory handle.

Usage

Being executed the function creates a directory list.

In case of a failure in the directory opening, the function returns '0'.

os.Path.dirsort

The os.Path.dirsort method defines the sort criteria and sort order for the opened directory. The syntax of the function is as follows:

```
CALL os.Path.dirsort(criteria, order)
```

where:

- *criteria* = a variable of STRING data type standing for the sort criteria. This parameter must be defined by one of the following strings:
 - *'undefined'* = means that no sort criteria is set which is the default. In this case all the entries are read as returned by the os.Path methods,
 - *'name'*, *'size'*, *'extension'* or *'type'* = sort the directory by file name, size, extension or type (directory, link, regular file) respectively,
 - *'atime'* = sort the directory by access time,
 - *'mtime'* = sort the directory by modification time;
- *order* = a variable of INTEGER data type which specifies whether the sort order is ascending or descending:
 - *'1'* to define the ascending order,
 - *'-1'* to define the descending order.

Usage

By this function invocation, the sort criteria and sort order are defined for any subsequent **diopen** call. When using the sort criteria *"name"*, the directory entries are ordered with regard to the current locale. When sorting by any other *criteria*, entries of the same value for the defined criteria are sorted by name following the value of the *order* parameter.

Here is an example of the os.Path.dirsort method calling:

```
#sort by name, ascending  
CALL os.Path.dirsort("name", 1)
```


os.Path.executable

The os.Path.executable method checks the file executability. The syntax of the function is as follows:

```
CALL os.Path.executable(f_name) RETURNING result
```

where:

- *f_name* = a string standing for the file name to check,
- *result* = TRUE or FALSE if the file is executable or not respectively.

os.Path.exists

The os.Path.exists method checks the file existence. The syntax of the function is as follows:

```
CALL os.Path.exists(f_name) RETURNING result
```

where:

- *f_name* = a string standing for the file name to check,
- *result* = TRUE or FALSE if the file exists or not, respectively.

os.Path.extension

The os.Path.extension method is used to return the extension of the file. The syntax of the function is as follows:

```
CALL os.Path.extension(f_name) RETURNING extension
```

where:

- *f_name* = a string standing for the file name with its extension after the last dot,
- *extension* = a STRING variable containing the part of the *f_name* following the last dot in it.

Usage

If no extension is specified in the *f_name*, the function returns NULL.

Here is an example of the os.Path.dirsort method calling:

```
#returns "txt"
```

```
CALL os.Path.extension ("File_name.txt") RETURNING extension
```

os.Path.homedir

The os.Path.homedir method is used to return the path to the HOME directory of the current user. The syntax of the function is as follows:

```
CALL os.Path.homedir() RETURNING home_dir
```

where:

- *home_dir* = a string containing the full path to the user HOME directory.

os.Path.isdirectory

The os.Path.isdirectory is used to check whether a file is a directory or not. The syntax of the function is as follows:

CALL `os.Path.isdirectory(f_name)` RETURNING `result`

where:

- *f_name* = a string containing the file name to check,
- *result* = TRUE if the file is a directory or FALSE otherwise.

os.Path.isfile

The os.Path.isfile method is used to check whether a file is a regular one or not. The syntax of the function is as follows:

CALL `os.Path.isfile(f_name)` RETURNING `result`

where:

- *f_name* = a string containing the name of the file to check,
- *result* = TRUE if the file is a regular one or FALSE otherwise.

os.Path.isHidden

The os.Path.isHidden is used to check whether a file is hidden or not. The syntax of the function is as follows:

CALL `os.Path.isHidden(f_name)` RETURNING `result`

where:

- *f_name* = a string containing the name of the file to check,
- *result* = TRUE if the file is a hidden one or FALSE otherwise.

os.Path.isroot

The os.Path.isroot method is used to check whether a file path is a root one. The syntax of the function is as follows:

CALL `os.Path.isroot(path)` RETURNING `result`

where:

- *path* = a string containing the path to check,
- *result* = TRUE if the path is a root one or FALSE otherwise.

os.Path.join

The `os.Path.join` method is used to join two segments of the path adding the platform-dependent separator to the returning path. The syntax of the function is as follows:

```
CALL os.Path.join(begin, end) RETURNING newpath
```

where:

- *begin* = a string containing the first path segment,
- *end* = a string containing the last path segment,
- *newpath* = a string containing the whole path joined.

Usage

Typically this function is called for when there is a need to construct a path with no system-specific code using the correct path separator:

```
CALL os.Path.join(os.Path.homedir(), file_name) RETURNING newpath
```

os.Path.mkdir

The `os.Path.mkdir` method is used to create a new directory. The syntax of the function is as follows:

```
CALL os.Path.mkdir(dname) RETURNING result
```

where:

- *dname* = a variable of a STRING data type standing for the name of the directory to create,
- *result* = TRUE or FALSE depending on the success of the function execution.

os.Path.mtime

The `os.Path.mtime` method is used to return the time of the latest modification of the file. The syntax of the function is as follows:

```
CALL os.Path.mtime(f_name) RETURNING mtime
```

where:

- *f_name* = a string containing the file name,
- *mtime* = a string containing the time of the last *f_name* modification in the standard format YYYY-MM-DD HH:MM:SS.

In the case of a failure NULL is returned.

os.Path.pathseparator

The `os.Path.pathseparator` method is used to return the path separator - the character used in environment variables to separate elements of the path. The syntax of the function is as follows:

```
CALL os.Path.pathseparator() RETURNING separator
```

where:

- *separator* = a string containing the path separator:
 - ':' = on UNIX,
 - ';' = on Windows.

os.Path.pathtype

The `os.Path.pathtype` is used to check whether a path is a relative or an absolute one. The syntax of the function is as follows:

```
CALL os.Path.pathtype(path) RETURNING pathtype
```

where:

- *path* = a string containing the path to check,
- *pathtype* = a string of "absolute" or "relative" if the path is absolute or relative respectively.

os.Path.pwd

The `os.Path.pwd` is used to return the current working directory. The syntax of the function is as follows:

```
CALL os.Path.pwd() RETURNING cw_dir
```

where:

- *cw_dir* = a STRING variable standing for the current working directory.

os.Path.readable

The `os.Path.readable` method is used to check whether a file is readable or not. The syntax of the function is as follows:

```
CALL os.Path.readable(f_name) RETURNING result
```

where:

- *f_name* = a STRING variable standing for the file name,
- *result* = TRUE if the file is a readable one or FALSE otherwise.

os.Path.rename

The os.Path.rename is used to change a file or a directory name. The syntax of the function is as follows:

```
CALL os.Path.rename(old_name, new_name) RETURNING result
```

where:

- *old_name* = a STRING variable standing for the current name of the file or directory,
- *new_name* = a STRING variable standing for the name to be assigned to the file or directory,
- *result* = TRUE if the function has been successfully executed and the file or directory has received the new name or FALSE otherwise.

os.Path.rootdir

The os.Path.rootdir method is used to return the root directory of the current working path. The syntax of the function is as follows:

```
CALL os.Path.rootdir() RETURNING root_dir
```

where:

- *root_dir* = a string containing the root directory of the current working path:
 - '/' = on UNIX,
 - '[a-zA-Z]:\' = on Windows.

os.Path.rootname

The os.Path.rootname is used to return the file path with its last element without an extension. The syntax of the function is as follows:

```
CALL os.Path.rootname(f_path) RETURNING r_name
```

where:

- *f_path* = a string containing the file path,
- *r_name* = a string containing the file path with its last element without an extension.

Usage

Being executed, the function removes the last file extension.

For example, passing "`\\root\\dir\\f_name.ext`" as the parameter will result in "`\\root\\dir\\f_name`".

os.Path.separator

The os.Path.separator method is used to return the character that separates path segments. The syntax of the function is as follows:

```
CALL os.Path.separator() RETURNING separator
```

where:

- *separator* contains the separator:
 - `'/'` = on UNIX,
 - `'\'` = on Windows.

os.Path.size

The `os.Path.size` is used to return the file size. The syntax of the function is as follows:

CALL `os.Path.size(f_name)` RETURNING `f_size`

where:

- *f_name* = a variable of a STRING data type standing for the file name,
- *f_size* = an integer variable standing for the file size.

Usage

In case the function fails to get the file size it returns the negative integer (-1).

os.Path.type

The `os.Path.type` is used to return the file type as a string. The syntax of the function is as follows:

CALL `os.Path.type(f_name)` RETURNING `f_type`

where:

- *f_name* = a string containing the file name,
- *f_type* = one of the following:
 - `'file'` = stands for the regular file,
 - `'directory'` = stands for the directory,
 - `'socket'` = stands for the socket,
 - `'fifo'` = stands for the fifo,
 - `'block'` = stands for the block device,
 - `'char'` = stands for the character device.

os.Path.volumes

The `os.Path.volumes` method is used to return the available volumes. The syntax of the function is as follows:

CALL `os.Path.volumes()` RETURNING `volumes`

where:

- *volumes* = a string containing the list of all available volumes separated by a pipe symbol (|).

os.Path.writable

The os.Path.writable method is used to check whether the file is writable. The syntax of the function is as follows:

CALL `os.Path.writable(f_name)` RETURNING `result`

where:

- *f_name* = a string containing the name of the file to check,
- *result* = TRUE if the file exists and it is writable or FALSE otherwise.

The os.Path methods used on UNIX only

os.Path.chown

The os.Path.chown method is used to change the UNIX owner and group of the specified file. The syntax of the function is as follows:

CALL `os.Path.chown(f_name, user_id, gui)` RETURNING `result`

where:

- *f_name* = a string containing the file name,
- *user_id* = an integer standing for the user id,
- *gui* = an integer standing for the group id,
- *result* = TRUE if success or FALSE in case of a failure.

os.Path.chrwx

The os.Path.chrwx method is used to change the UNIX permissions of the specified file. The syntax of the function is as follows:

CALL `os.Path.chrwx(f_name, mode)` RETURNING `result`

where:

- *f_name* = a string containing the file name,
- *mode* = an integer variable standing for the UNIX permission combination in decimal (not octal!),
- *result* = TRUE if success or FALSE in case of a failure.

Usage

The *mode* stands for the combination of read (r), write (w) and execution (x) bits for the owner, group and other part of the UNIX file permission. Only the decimal value must be used there!

For example, in order to have -rwx-rw-r- permissions set, you must pass to this method (((4+2+1) *64) + ((4+2) * 8) + 4) = 500



Though the *chmod* UNIX command takes an octal value as a parameter, the *mode* must take a decimal one.

os.Path.gid

The `os.Path.gid` method is used to return the UNIX group id of the specified file. The syntax of the function is as follows:

CALL `os.Path.gid(f_name)` RETURNING `group_id`

where:

- *f_name* = a string containing the file name,
- *group_id* = an integer standing for the group id.

Usage

In case the function fails to get the user id, it returns the negative integer (-1).

os.Path.islink

The `os.Path.islink` method is used to check whether a file is a UNIX symbolic link. The syntax of the function is as follows:

CALL `os.Path.islink(f_name)` RETURNING `result`

where:

- *f_name* = a string variable standing for the file name,
- *result* = TRUE if the file is a symbolic link or FALSE otherwise.

os.Path.rwx

The `os.Path.rwx` method is used to return the UNIX file permissions of a file. The syntax of the function is as follows:

CALL `os.Path.rwx(f_name)` RETURNING `mode`

where:

- *f_name* = a string variable standing for the file name,
- *mode* = an integer standing for the combination of permissions for user, group or other.

Usage

The combination of permissions is always returned as a decimal value.

For example, if a file has the `-rw--r--r` permissions, you will get $((4+2) * 64 + 4 * 8 + 4) = \underline{420}$

In case of a failure the function returns the negative integer (-1).



It is important to keep in mind that the *mode* always gets a decimal value.

os.Path.uid

The `os.Path.uid` method is used to return the UNIX user id of a file. The syntax of the function is as follows:

CALL `os.Path.uid(f_name)` RETURNING `id`

where:

- *f_name* = a string variable standing for the file name,
- *id* = an integer standing for the user id.

Usage

In case of a failure the function returns the negative integer (-1).

Large data types methods

Querix 4GL supports the following methods manipulating large data types.

GetLength()

The GetLength() method returns an integer value indicating the size of a TEXT variable in bytes.

```
LET txtlength = txtvar.GetLength()
```

The GetLength() method returns an integer value indicating the size of a TEXT variable in bytes.

ReadFile()

The ReadFile() method is used to read data from the specified file and pass it to the memory or to the variable of a large data type indicated by the LOCATE statement.

The method requires an argument which identifies the source file and the path to it.

```
CALL txtvar.ReadFile("/textfiles/list.txt")
```

GetLength() method returns an integer value indicating the size of a TEXT variable in bytes

WriteFile()

The WriteFile() method is used to write data from the variable (source file or memory) to the specified file. The destination file path and name are to be passed to the method as an argument.

```
CALL txtvar.WriteFile("/bytefiles/picture1.jpg")
```

If the destination file does not exist in the specified folder, it is created automatically.

Application Launch


run

The RUN statement is used to invoke an operational system command or run a 4GL or any other application. Its syntax is as follows:

```
RUN command [WITHOUT WAITING | RETURNING result]
```

where:

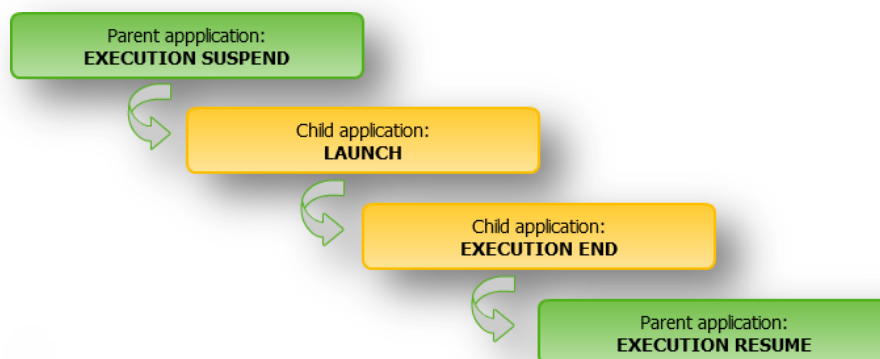
- *command* = a quoted character string specifying the command to be executed or the application to launched, or a variable of the character data type (which can also be a record member or an array element), containing an application name and the path to it if required,
- *result* = an INTEGER variable standing for the termination status code of the command invoked (TRUE in case of success, and FALSE otherwise).

	The RUN statement cannot include both the WITHOUT WAITING keywords and the RETURNING clause, as the latter presupposes, that a parent application requires the result of a child application execution.
--	---

Usage

The commands specified in the RUN statement are executed on a server side. For example, on the RUN "notepad" statement execution, the notepad will be launched on the server (not on the client).

The RUN statement can be used to execute a child 4GL application from within the parent one. When used for this purpose, it affects an application workflow as follows (unless the WITHOUT WAITING keywords are specified):



If the RUN statement contains the WITHOUT WAITING clause, it means that the two applications are being executed concurrently (no suspend in the parent application execution takes place).

The example below illustrates how an application can be executed from within another application:

```
RUN "app_name.exe"
```

By default, the RUN statement allows only 4GL applications execution. To enable the opportunity of any other program launch or operational system commands invocation, the value of the `progonly` property in the `listener.xml` file should be set to `false` (for the changes to take effect, please, restart Querix Application Server):

```
<service>
  <name>default-1889</name>
  <port>1889</port>
  <applicationdir>C:\ProgramData\Querix\Lycia 6\progs</applicationdir>
  <logdir>C:\ProgramData\Querix\Lycia 6\logs</logdir>
  <envfile>C:\ProgramData\Querix\Lycia 6\etc\inet.env</envfile>
  <authmode>none</authmode>
  <auth_module>modauthnt.dll</auth_module>
  <logging>true</logging>
  <description>Default no authentication listener</description>
  <accessfilter>C:\Program
Files\Querix\AppServer\filterip.xml</accessfilter>
  <enabled>true</enabled>
  <toolset>lycia2</toolset>
  <progonly>false</progonly>
</service>
```



The `progonly` property set to `false` allows launching of any application, that can be normally executed from the console. The application must be located in the Lycia AppServer work directory (`progs`) or the path to it must be specified by the environment variable `$PATH`.

exec_local()

The `exec_local()` function is used to run a local application, such as notepad on Windows. It takes 2 arguments and has the following syntax:

```
exec_local(command, user_id)
```

where:

- *command* = a variable of the VARCHAR data type or a quoted character string, specifying the path to an application to be launched,
- *user_id* = a variable of the VARCHAR data type or a quoted character string, specifying the user account to run an application for.

Here is the code snippet illustrating how Notepad can be invoked using this function:

```
CALL menu_publish()
LET id_action = 0
WHILE (TRUE)
    LET id_action = execute_menu()
    IF id_action = 0 THEN
        CONTINUE WHILE
    ELSE
        IF id_action = 1 THEN
            CALL exec_local("notepad", "admin")
        ELSE
            EXIT WHILE
        END IF
    END IF
END WHILE
```

exec_program()

The `exec_program()` function is used to run a 4GL program on a GUI server. It takes 4 arguments and has the following syntax:

```
exec_program(command, server, port, user_id)
```

where:

- *command* = a variable of VARCHAR data type or a quoted character string, specifying the path to an application to be run,
- *server* = a variable of the VARCHAR data type or a quoted character string, specifying the IP or the domain name of an application server.
- *port* = a variable of the VARCHAR data type or a quoted character string, specifying the port used by an application server.
- *user_id* = a variable of the VARCHAR data type or a quoted character string, specifying the user account to run an application for.

Usage

Here are examples of the `exec_program()` function:

```
CALL exec_program("my_program", "190.155.207.10", "1891", "admin")  
CALL exec_program("my_program", "192.168.10.213", "1889", "SYSTEM")
```

winexec() and winexecwait()

The `winexec()` and `winexecwait()` functions request the GUI client to execute a command on the operating system the parent application is being run on.

The syntax of the functions are similar to the one of the RUN statement:

```
CALL winexec(app-name)           (= RUN app_name WITHOUT WAITING)
CALL winexecwait(app_name)      (= RUN app_name)
```

Usage

The main difference between RUN and `winexec()/winexecwait()` is that the child application launched this way is executed on the client side.

If the `winexec()` or `winexecwait()` functions are invoked in a character mode process, they simply execute a standard RUN statement.

Here is the code snippet illustrating how the functions are used:

```
MAIN
  OPEN WINDOW win AT 1,1 WITH 15 ROWS , 60 Columns
  CALL winexec("notepad")

  CALL fgl_winmessage("End","WinExec() Demo Application - Click OK to close
demo application","info")
END MAIN
```

winshellexec() and winshellexecwait()

The `winshellexec()` and `winshellexecwait()` functions are used to open documents on the client PC. These functions are similar to `winexec()` and `winexecwait()`. The key difference is that they are executed via a system shell. This can be especially useful on Windows, as it allows an automatic document opening.

Usage

For example,

```
CALL winshellexec("c:\\program files\\querix\\hydra\\progs\\myfile.pdf")
```

Is equivalent to:

```
CALL winexec("explorer c:\\program files\\querix\\hydra\\progs\\myfile.pdf")
```

Here is the code snippet illustrating how the functions can be used within the 4GL code:

```
MAIN
  DEFINE file_name VARCHAR(200)

  OPEN WINDOW win
    AT 1,1
    WITH 24 ROWS , 80 Columns

  LET file_name = fgl_getenv("QUERIXDIR")
  DISPLAY "QUERIXDIR =", file_name AT 5,5

  LET file_name = file_name CLIPPED,
"/documentation/Multiple_Database_Usage.pdf"
  DISPLAY "Multi Database Manual is located in:" AT 6,5
  DISPLAY file_name AT 7,5

  CALL winshellexec(file_name)
  CALL fgl_winmessage("End","WinExec() Demo Application - Press OK to close
demo application","info")
END MAIN
```

Toolbar

fgl_dialog_keydivider()

Using this function, it is possible to specify a divider for dynamic toolbars. A divider is a separator between keys in the shape of a thin indented line. This feature can be used to make clearer the distinction between keys and provides more design choices for the developer.

There are 2 functions for adding dividers:

```
fgl_keydivider()  
fgl_dialog_keydivider()
```

The reason for having two functions is so that, as mentioned previously, there is a temporary function available for only accepting input, i.e., the `fgl_dialog_keydivider()` function.

Usage

This function draws a divider between two keys. The location of the divider is based on the keys' ordinals and specifying, in the function call, a value between the two ordinals that you wish to have the divider appear. It is not possible to set a key divider on a location already occupied by a key label.

To clarify, imagine we have 2 keys defined:

```
key.accept.text = "Accept Key"  
key.accept.order = 10  
key.f1.text "F1 Key"  
key.f1.order = 20
```

We can place a divider between the two keys by passing any location between 10 and 20 to the function, so a simple call would be:

```
CALL fgl_dialog_keydivider(15)  
  
# As 10 < 15 < 20 a divider will be drawn  
# between the Accept and F1 keys
```

fgl_dialog_setkeylabel()

The `fgl_dialog_setkeylabel()` allows for the creation of toolbars dynamically and automatically in an input state. The `fgl_dialog_setkeylabel()` function takes up to four arguments. The first refers to the key identifier, such as "F2". The second argument is the label name. The optional third refers to the icon name and the optional fourth specifies the key label position.

```
fgl_dialog_setkeyLabel("f2", "Login", "my_icon", 1, FALSE)
```

The *icon name* and *position* definitions, displayed above, are two optional settings and refer to the icon image to be used (e.g., .jpeg, .png, .tif, .ico, .bmp), and the item position order on the toolbar, respectively. The icon position can be modified simply by editing the relevant argument shown in the above example, e.g., from '1' to '3'.

There is no change to the way key labels are defined; these are created by designating the key identifier and label name arguments, as illustrated above.

The last argument is optional and specifies whether the key label will be visible, if it is not referenced in any ON KEY clause or on the COMMAND menu clause, or it will be hidden. The default value is FALSE (0), it means that if this argument is omitted, or if it is set to FALSE, the key label will not be visible at all. If it is set to TRUE(1), the key label with no action assigned will be displayed as grey and inactive. If some action is assigned to the key, it will be displayed as active regardless of the value of this attribute.

If you want to hide the label and/or the icon, you can call this function as shown below. However, if the key specified in the function is pressed, the event assigned to the toolbar button will be still triggered, even though the button is invisible. The event will be disabled only if the specified key is not referenced in the code (e.g. there is no COMMAND clause or ON KEY clause for this key in the currently executed code).

```
fgl_dialog_setkeyLabel("f2", "", "")
```

Note: If a key has been associated with, for example, HELP (as below) then it is recommended that you use the key's reference in this function. This will save changing all references to the key throughout the code.

So if you have the following somewhere in your code:

```
OPTIONS  
HELP KEY F1
```

Then you should use this function as below:

```
fgl_dialog_setkeylabel("HELP", "This is the label for the F1 key", "my_icon", 1)
```

fgl_getkeylabel()

The `fgl_getkeylabel()` function passes a single argument that relates to the dynamic toolbar name/identifier. Utilising this function will return the label associated to that particular key label. The syntax is outlined, as follows:

```
fgl_getkeylabel(keyname)
```

The Keyname relates to the logical keyname specified such as "F2", and returns the corresponding key label as a text string.

fgl_keydivider()

Using this function, it is possible to specify a divider for dynamic toolbars. A divider is a separator between keys in the shape of a thin indented line. This feature can be used to make clearer the distinction between keys and provides more design choices for the developer.

There are 2 functions for adding dividers:

```
fgl_keydivider()  
fgl_dialog_keydivider()
```

The reason for having two functions is that, as mentioned previously, there is a temporary function available for only accepting input, i.e., the `fgl_dialog_keydivider()` function.

Usage

This function draws a divider between two keys. The location of the divider is based on the keys' ordinals and specifying, in the function call, a value between the two ordinals that you wish to have the divider appear. It is not possible to set a key divider on a location already occupied by a key label.

To clarify, imagine we have 2 keys defined:

```
key.accept.text = "Accept Key"  
key.accept.order = 10  
key.f1.text "F1 Key"  
key.f1.order = 20
```

We can place a divider between the two keys by passing any location between 10 and 20 to the function, so a simple call would be:

```
CALL fgl_keydivider(15)  
  
# As 10 < 15 < 20 a divider will be drawn  
# between the Accept and F1 keys
```

fgl_setkeylabel()

The `fgl_setkeylabel()` allows for the creation of toolbars dynamically and automatically in the global state. The `fgl_setkeylabel()` function takes up to four arguments. The first refers to the key identifier, such as "F2". The second argument is the label name. The optional third refers to the icon name and the optional fourth specifies the key label position. The optional fifth is a Boolean value FALSE by default. If you set it to TRUE, the label and icon will be visible (but disabled) when no corresponding key event currently exists. By default the label is hidden, if there is no corresponding key for it and the last argument is omitted.

```
fgl_setkeyLabel("f2", "Login", "my_icon", 1, TRUE)
```

The icon name and position definitions, displayed above, are two optional settings and refer to the icon image to be used (e.g., .jpeg, .png, .tif, .ico, .bmp), and the item position order on the toolbar, respectively. The icon position can be modified simply by editing the relevant argument shown in the above example, e.g., from '1' to '3'.

There is no change to the way key labels are defined; these are created by designating the key identifier and label name arguments, as illustrated above.

If you want to hide the label and/or the icon, you can call this function as shown below. However, if the key specified in the function is pressed, the event assigned to the toolbar button will be still triggered, even though the button is invisible. The event will be disabled only if the specified key is not referenced in the code (e.g. there is no COMMAND clause or ON KEY clause for this key in the currently executed code).

```
fgl_setkeyLabel("f2", "", "")
```

Note: If a key has been associated with, for example, HELP (as below) then it is recommended that you use the key's reference in this function. This will save changing all references to the key throughout the code.

So if you have the following somewhere in your code:

```
OPTIONS  
HELP KEY F1
```

Then you should use this function as below:

```
fgl_setkeylabel("HELP", "This is the label for the F1 key", "my_icon", 1)
```

fgl_setkeystatic()

The fgl_setkeystatic() function allows for the inclusion of persistent (static) toolbar items. Such icons will appear as disabled if their associated keypress has not been assigned. This function modifies the toolbar on a global scale.

The function takes two parameters – the name of the key and a Boolean value representing whether it is static or not. The default value for this function is false, meaning keys are not automatically static, and disappear when not assigned to an event.

To make the F1 key appear in the dynamic toolbar even when its event isn't defined:

```
CALL fgl_setkeystatic("F1", "TRUE")
```

The dynamic toolbar should now always display the label 'F1 key'. However it would appear disabled if the key hadn't been assigned an event.

Ring Menu

create_menu()

The create_menu() function is used to create a ring menu. It does not require any arguments.

It generates a menu_id (of INTEGER data type) which is then used as a reference for adding options and submenus. This function can be used in the following way:

```
LET menu_id=create_menu()
```

In this case the variable 'menu_id' will contain the integer which identifies this menu.

menu_add_option()

The menu_add_option() function is used to create an option within a menu, created by the create_menu() function. This function takes four parameters one of which is optional. It has the following syntax:

```
menu_add_option(menu_id, label, action_id [,enabled])
```

- Menu_id = a variable of INTEGER data type. It refers to the menu ID generated by the create_menu() function to which you want to add the option.
- Label = a variable of VARCHAR data type or a quoted character string which specifies the name of the menu option.
- Action_id = a variable of INTEGER data type or an integer. You can assign any action_id to a menu option, but the action IDs used in the program must be unique.
- Enabled = an integer or a variable of the INTEGER data type which specifies whether the menu option should be visible or not. This is an optional parameter and the option is visible by default, if it is absent. It can accept two values:
 - 1 – to make the menu option visible.
 - 0 – to hide the menu option.

Here is an example of the menu_add_option() function:

```
LET menu_id = menu_create()  
LET action_id = 1  
  
CALL menu_add_option(menu_id, "first option", action_id)
```

This function creates a new option called "first option" in the menu which menu_id is used, and assigns "1" as the action ID for this option. This action_id will later be used by the execute_menu() function. The menu_add_option returns 'option_id'.

menu_add_submenu()

The menu_add_submenu() function is used to add a submenu to an existing menu. This process is similar to the process of adding an option to the menu. It has the following syntax:

```
menu_add_submenu(menu_id, menu_label [,enabled])
```

This function accepts three parameters, one of which is optional:

- Menu_id = a variable of the INTEGER data type. It refers to the menu ID generated by the create_menu() function to which you want to add a submenu.
- Menu_label = a variable of VARCHAR data type or a quoted character string which specifies the name of the submenu.
- Enabled = an integer or a variable of the INTEGER data type which specifies whether the submenu should be enabled or disabled. This is an optional parameter and the option is visible by default, if it is absent. It can accept two values:
 - 1 – to enable the submenu.
 - 0 – to disable the submenu.

The menu_add_submenu() function returns a submenu_id which should be used instead of the menu_id in the menu_add_option() function, if you want to add an option to the submenu. Here is an example of this function:

```
menu_add_submenu(menu_id, "first submenu", 0)
```

The new submenu called "first submenu will be disabled.

menu_publish()

The menu_publish() function is used to publish your menu. Use this function when you are contented with your menu and there are no more options to add, while after this function is called, the menu cannot be modified. Only after this function is executed you can use the menu_execute() function.

This function does not accept any arguments and does not return anything. The menu_add_option() and menu_add_submenu() functions must be placed between the menu_create() function and menu_publish() function.

execute_menu()

The `execute_menu()` function is used to monitor the state of the menu and to return the `action_id` of the menu option activated by the user. This option does not require any arguments. See the snippet of code above for the usage example.

Code Sample:

```
MAIN

DEFINE
  main_menu_id INT,
  submenu1_id,exit_id INT,
  action_id INT

LET main_menu_id = create_menu()

LET submenu1_id = menu_add_submenu(main_menu_id, "menu1")
CALL menu_add_option(submenu1_id, "menu1_submenu1 ",101)
CALL menu_add_option(submenu1_id, "menu1_submenu2 ",102)
CALL menu_add_option(submenu1_id, "menu1_submenu3 ",103)

LET exit_id = menu_add_option(main_menu_id, "exit",199)

LET action_id = 0

WHILE TRUE
CALL menu_publish()
  LET action_id = execute_menu()

  IF action_id = 0 THEN
    CONTINUE WHILE
  END IF

  CASE action_id
  WHEN 101
    DISPLAY "you pressed 'menu1_submenu1'" AT 5,1
  WHEN 102
    DISPLAY "you pressed 'menu1_submenu2'" AT 5,1
  WHEN 103
    DISPLAY "you pressed 'menu1_submenu3'" AT 5,1 ATTRIBUTE (BLUE)
  WHEN 199
    EXIT WHILE
  END CASE

END WHILE

END MAIN
```

Dialog Box

fgl_file_dialog()

The `fgl_file_dialog()` function allows an application developer to call a message box dialog allowing a user to save or open a particular file. The function takes between two and five parameters, the first two parameters are obligatory. They are outlined in the following syntax example:

```
fgl_file_dialog("<type>", "<multiselect>", "<title>", "<default_path>",  
"<default_filename>", "<format>")
```

- **Type** = Refers to the type of dialog box: open or save. This is an obligatory parameter.
- **Multiselect** = Specifies how many files can be selected at a time: 1 = single file, 0 = multiple files. This is an obligatory parameter.
- **Title** = Allows you to specify the title of the dialog. This is an optional parameter.
- **Default Path** = Refers to the default file path contained in the dialog box. This is an optional parameter.
- **Default Filename** = Refers to the default filename that is stored in the dialog box's filename field. This is an optional parameter.
- **Format** = Allows you to specify the files that are selectable from the dialog box. This is an optional parameter.

A working example of the `fgl_file_dialog()`, using all parameters is as follows:

```
fgl_file_dialog("open", "1", "hello please select a file", "defaultpath",  
"defaultfilename", "*.jpg")
```

fgl_message_box()

The fgl_message_box() function displays a simple message box, with a specifiable range of button options.



Figure 51 - fgl_message_box() Function

Usage

The function takes up to three arguments, only the first of which must be included. The first argument is a CHAR(255) datatype which is the message to be displayed in the message box. The second argument, also a CHAR(255) datatype, is the title of the message box.

The third argument is an INTEGER value which defines the buttons that appear in the message box. This last argument can take any of the following values:

- 0: OK
- 1: OK, Cancel
- 2: Retry, Cancel
- 3: Yes, No, Cancel
- 4: Yes, No

If no value is defined for the third argument, the default value used is a 0, which will display only an "OK" button.

The function returns an integer value for the button pressed; which will be one of the following:

- 1: OK, Yes
- 2: Cancel
- 3: No, Retry

Code Sample: fgl_message_box_function

This code sample defines the text strings that set the message_box title and text, but sets the style of the button options as a variable called *button_style*, which you can choose, by entering a value between 0 and 4.

```
MAIN

    DEFINE int_return, button_style int

    LET button_style = fgl_winprompt(2,2, "Enter the button type value (0-4)", "", 1,
1)

    LET int_return = fgl_message_box("This is the message box title","This is the
actual message text", button_style)

CASE int_return
    WHEN 1
        DISPLAY "Return value: 1 = You pressed OK/YES" at 8,5
        EXIT CASE
    WHEN 2
        DISPLAY "Return value: 2 = You pressed Cancel" at 8,5
        EXIT CASE
    WHEN 3
        DISPLAY "Return value: 3 = You pressed No, Retry" at 8,5
        EXIT CASE
    OTHERWISE
        Error "Error - undefined case"
    EXIT CASE
END CASE

sleep 5
END MAIN
```

fgl_winbutton() §

The fgl_winbutton() function displays an interactive message box in a separate window with multiple choices. The syntax is shown below; the following table describes each of the variables that the function takes as an argument.

```
fgl_winbutton( title, text, default, buttons, icon, danger )
```





<i>title</i>	CHAR(n)	Message box title.
<i>text</i>	CHAR(n)	Message box message (use \n for newline).
<i>default</i>	CHAR(n)	Default button selected.
<i>buttons</i>	CHAR(n)	List of values separated by a pipe ().
<i>icon</i>	CHAR(n)	Name of icon to be used (info, exclamation, question, stop).
<i>danger</i>	SMALLINT	This argument is for dynamic 4gl compatibility only – (Number of the warning item (for X11 only).)
Returns	CHAR(n)	Label of the button selected.

If you want to create a button using a label with several words, you will have to use underscores, full stops, or hyphens, to separate the words. You cannot use spaces, the function interprets these as separators, and will create one button for each word.

You can declare a maximum of 7 buttons with 10 characters each.

Subject to the two rules above, you can put anything in the definition of a button.

By using any of the *icon* names your system should find locally stored icons similar to those shown below.

info	exclamation	question	stop
			

Code Sample: fgl_winbutton_function.4gl

```
MAIN
  DEFINE button_string char(20)

  LET button_string = fgl_winbutton("First fgl_winbutton test", "Open a dialog box
with 1 line of text, two buttons and information icon", "Yes", "Yes|No", "info", 1)
  CALL button_pressed(button_string)

  LET button_string = fgl_winbutton("Second fgl_winbutton test", "Open a dialog box
with 2 lines of text\nthree buttons and stop icon", "OK", "Ok|Cancel|Help", "stop",
1)
  CALL button_pressed(button_string)

  LET button_string = fgl_winbutton("Third fgl_winbutton test", "Open a dialog box
with 3 lines of text\nfour buttons \nand question icon", "OK",
"Ok|Cancel|Help|Something else", "Question", 1)
  CALL button_pressed(button_string)

  LET button_string = fgl_winbutton("Fourth fgl_winbutton test", "Open a dialog box
with 3 lines of text\nseven buttons \nand Exclamation icon", "OK",
"Ok|Cancel|Help|But4|But5|But6|But7", "Exclamation", 1)
  CALL button_pressed(button_string)

END MAIN
```

Environment

fgl_arch()

The `fgl_arch()` function returns a `char(n)` value, which shows the architecture that the compiled binary is running on.

For example:

- Windows NT = 'nt'
- Linux = 'lnx'
- Solaris = 'sun'
- HP-UX = 'hp'

```
MAIN
```

```
    DISPLAY "This system architecture is: " , fgl_arch() at 1,5
```

```
    DISPLAY "nt  = Windows" AT 3, 5
```

```
    DISPLAY "lnx = Linux" AT 4, 5
```

```
    DISPLAY "sun = Sun" AT 5, 5
```

```
    DISPLAY "hp  = HP-Unix" AT 6, 5
```

```
    CALL fgl_winmessage("Exit","Press any key to close this demo application","info")  
END MAIN
```

fgl_getenv()

The `fgl_getenv()` function takes a character expression argument which is the name of the variable and returns a character string, which is the value of this environment variable.



Figure 52 - fgl_getenv()

Usage

The argument of `fgl_getenv()` must be a character expression that returns the name of an environment variable. To evaluate a call to `fgl_getenv()`, 4GL takes the following actions at runtime:

1. Evaluates the character expression argument of `fgl_getenv()`
2. Searches among environment variables for the returned value

If the requested value exists, the function returns it as a character string and then returns control of execution to the calling context.

The identifier of an environment variable is not a 4GL expression, so you typically specify the argument as a quoted string or character variable. For example, this call evaluates the **DBFORMAT** environment variable:

```
fgl_getenv("DBFORMAT")
```

You can assign the name of the environment variable to a character variable and use that variable as the function argument. If you declare a CHAR or VARCHAR variable called **env_var** and assign to it the name of an environment variable, a `fgl_getenv()` function call could look like this:

```
fgl_getenv(env_var)
```

If the argument is a character variable, be sure to declare it with sufficient size to store the character value returned by the `fgl_getenv()` function; otherwise 4GL will truncate the returned value.

If the specified environment variable is not defined, `fgl_getenv()` returns a null value. If the environment variable is defined but does not have a value assigned to it, `fgl_getenv()` returns blank spaces.

You can use the `fgl_getenv()` function anywhere within a 4GL program to examine the value of an environment variable. The following program segment displays the value of the **INFORMIXDIR** environment variable. The environment variable is identified in the `fgl_getenv()` call by enclosing the name **INFORMIXDIR** between quotation marks:

Sample Code: fgl_getenv1_function.4gl

```
MAIN

    DEFINE informix_path CHAR(64)

    LET informix_path = fgl_getenv("INFORMIXDIR")

    IF informix_path THEN -- check if environment variable exists (not exists
returns NULL)
        DISPLAY "Informix installed in ", informix_path CLIPPED
    ELSE
        DISPLAY "Informix is not installed on your system" at 5,5
    END IF

    CALL fgl_winmessage("Exit","Press any key to close this demo application","info")
END MAIN
```

The next example also displays the value of an environment variable, which the user specifies. In this case, the environment variable is identified by the **env_variable** character variable, and its contents are stored in a variable called **env_path**:

Sample Code: fgl_getenv2_function.4gl

```
MAIN

  DEFINE
    env CHAR(500),
    env_variable CHAR(60),
    output_string CHAR(80)

  CALL fgl_winprompt(5, 2, "Enter the name of the environment variable", "", 60,
0) returning env_variable
  LET env = fgl_getenv(env_variable)

  LET output_string = "The environment variable " || env_variable || " is set to "
|| env

  IF env is not NULL THEN -- check if environment variable exists (not exists
returns NULL)
    DISPLAY "The environment variable was found: ", output_string CLIPPED AT 5,5
  ELSE
    DISPLAY "The environment variable was not found !   " , env_variable  at 5,5
  END IF

  CALL fgl_winmessage("Exit","Press any key to close this demo application","info")
END MAIN
```

References

For environment variables that control features of an Informix database, see the *Informix Guide to SQL: Reference*. For descriptions of environment variables that can affect the visual displays of Querix 4GL programs in general, see the Environment Variables appendix in the 'Lycia Developers Guide'.

fgl_getpid()

To retrieve the system process identifier (pid) for the current program use the *fgl_getpid()* function. It returns an INTEGER value which is the process identifier.



On Windows systems, this function returns a positive number, simulating the UNIX behaviour.

fgl_getresource()


The *fgl_getresource()* function returns the value of an FGLPROFILE resource. If the entry is not defined in fglprofile, the function returns NULL.

```
fgl_getresource( resname )
```

Where *resname* is a CHAR(n) datatype variable that defines the name of the FGLPROFILE resource. It returns the resource value.

Using fgl_getresource ()

```
DEFINE def CHAR(100)  
LET def = fgl_getresource("fglrun.defaults")
```

	<p>Users are free to add custom fglprofile entries but must take care when choosing the resource names so as not to conflict with standard entries. It is strongly recommended that custom fglprofile entries start with "custom.*".</p>
---	--

fgl_getversion()

The `fgl_getversion_function()` returns the version number of the 4GL runtime library currently in use. It returns a `CHAR(60)` which is the build version number.

Code Sample: `fgl_getversion_function.4gl`

```
MAIN
  DEFINE msg CHAR(60)

  LET msg = "You are using version: ", fgl_getversion()

  CALL fgl_message_box(msg)
END MAIN
```

fgl_putenv()

The function `fgl_putenv` allows the developer to set environment values in the 4GL processes environment. The syntax of the function invocation is as follows:

```
CALL fgl_putenv("LYCIA_DB_DRIVER=informix")
```

fgl_setenv()

The `fgl_setenv()` function allows the environment value setting and modification. The syntax of the function invocation is as follows:

```
fgl_setenv(var_name, var_value)
```

where:

- `var_name` = a STRING standing for an environment variable name,
- `var_value` = a STRING standing for the value of the `var_name` to be set.

Usage

Here is an example of the `fgl_setenv` function calling:

```
CALL fgl_setenv("LYCIA_DB_DRIVER", "informix")
```

Be careful when passing the NULL value to an environment variable as this will affect your application a little bit different depending on the operation system it is being executed by.

On the NULL value setting on Windows, the variable is removed from the environment. And On Unix, the same action results in an empty value setting for the variable specified as `var_name`.

fgl_username()

The fgl_username() function returns a CHAR(n) value, which is the user name under which the current process is being run.

This function should be used in preference to the SQL keyword USER in database statements, because the returned value of USER can be dependent on the database engine.

Code Sample: fgl_username_function.4gl

```
MAIN

    DISPLAY "This process is currently run under the username: ", fgl_username() at
5,5
    CALL fgl_winmessage("Exit","Press any key to close this demo application","info")

END MAIN
```


Help

err_get()

The ERR_GET() function takes an integer expression as its argument and returns a character string containing the text of the 4GL or SQL error message for the specified argument.



Figure 53 - ERR_GET() Function

Usage

ERR_GET() is most useful when you are developing a program. The message that it returns is probably not helpful to the user of your 4GL application. The argument of ERR_GET() is typically the value of the **status** variable, which is affected by both SQL and 4GL errors, or else a member of the global **SQLCA.SQLCODE** record. The LET statement in the following program fragment assigns the text of a 4GL error message to **my_error_text**, a CHAR variable:

```
MAIN
  DEFINE
    current_error_status integer,
    my_error_text char(100)

  DISPLAY "*** Function ERR_GET() - Example error -1110 - no Form file found ***" AT
  5,5

  LET current_error_status = -1110

  IF current_error_status < 0 THEN
    LET my_error_text = ERR_GET(current_error_status)
    ERROR my_error_text
  END IF

  CALL fgl_winmessage("Exit","Press any key to close this demo application","info")
END MAIN
```

The value of status can be assigned by a wide range of different statement variables. In order to avoid the status being changed unexpectedly it is always advisable to copy it as soon as possible so that the value you want to maintain does not get changed.

References

ERR_PRINT(), ERR_QUIT(), ERRORLOG(), fgl_gethelp(), STARTLOG()

err_print()

The ERR_PRINT() function takes an integer expression as its argument and displays on the Error line the text of an SQL or 4GL error message, corresponding to a negative integer argument.



Figure 54 - ERR_PRINT() Function

Usage

The argument of ERR_PRINT() specifies an error message number, which must be less than zero. It is typically the value of the global **status** variable, which is affected by both SQL and 4GL errors. For SQL errors only, you can examine the global **SQLCA.SQLCODE** record.

ERR_PRINT() is most useful when you are developing a 4GL program. The message that it returns is probably not helpful to the user of your application.

The following program segment sends any error message to the Error line:

```
MAIN
  DEFINE
    current_error_status integer,
    my_error_text char(100)

    LET current_error_status = -1110

  DISPLAY "*** Function ERR_PRINT() - Example error -1110 - no Form file found ***"
  AT 5,5

  IF current_error_status < 0 THEN
    CALL ERR_PRINT(current_error_status)
  END IF

  CALL fgl_winmessage("Exit","Press any key to close this demo application","info")
END MAIN
```

The value of status can be assigned by a wide range of different statement variables. In order to avoid the status being changed unexpectedly it is always advisable to copy it as soon as possible so that the value you want to maintain does not get changed.

If you specify the WHENEVER ANY ERROR CONTINUE compiler directive (or equivalently, the **anyerr** command-line flag), **status** is reset after certain additional 4GL statements, as described in "The ANY ERROR Condition" section of the 4GL Statement volume.

References

ERR_GET(), ERR_QUIT(), ERRORLOG(), STARTLOG()

err_quit()

The ERR_QUIT() function takes an integer expression as its argument and displays on the Error line the text of an SQL or 4GL error message, corresponding to the error code specified by its negative integer argument, and terminates the program.



Figure 55 - ERR_QUIT() Function

Usage

The argument of ERR_QUIT() specifies an error message number, which must be less than zero. It is typically the value of the global **status** variable, which is reset by both SQL and 4GL errors. For SQL errors only, you can examine the global SQLCA.SQLCODE record.

The ERR_QUIT() function is identical to the ERR_PRINT() function, except that ERR_QUIT() terminates execution once the message is printed. ERR_QUIT() is primarily useful when you are developing a 4GL program. The message that it returns is probably not helpful to the user of your application.

If an error occurs, the following statements display the error message on the Error line, and terminate program execution:

```
MAIN
  DEFINE
    current_error_status integer,
    my_error_text char(100)

  LET current_error_status = -1110

  DISPLAY "*** Function ERR_QUIT() - Example error -1110 - no Form file found ***" AT
  5,5

  IF current_error_status < 0 THEN
    CALL ERR_QUIT(current_error_status)
  END IF

  CALL fgl_winmessage("ERR_QUIT","Press any key to call ERR_QUIT (close this demo
  application)","info")
END MAIN
```

If you specify the WHENEVER ANY ERROR CONTINUE compiler directive (or equivalently, the **anyerr** command-line flag), **status** is reset after certain additional 4GL statements, as described in "The ANY ERROR Condition" in the 4GL Statements volume.

References

ERR_GET(), ERR_PRINT(), ERRORLOG(), STARTLOG()

errorlog()

The ERRORLOG() function takes a character expression as its argument and copies that argument into the current error log file.



Figure 56 - ERRORLOG() Function

Usage

If you simply invoke the STARTLOG() function, any extra error records that 4GL adds to the error log after each following error have this format:

```
Date: 11/05/2004 Time: 13:57:14
Program error at "contact.4gl", line number 137.
SQL statement error number -236.

Number of columns in INSERT does not match number of VALUES
```

The actual record might be incomplete if, after an error, the operating system does not keep the buffer containing the module name or the line number. You can use the ERRORLOG() function to supplement default error records with additional information. Entries that ERRORLOG() makes in the error log file automatically include the date and time when the error was recorded.

The following description details a typical sequence of events when logging system error messages:

1. Call STARTLOG() to open or create an error log file.
If an error is found, test the value of the global status variable to see if it is less than zero.
If its status is negative, call ERR_GET() to retrieve the error text.
Call ERRORLOG() to make an entry into the error log file.

By default, the WHENEVER ERROR CONTINUE statement is in effect. This default prevents the first SQL error from terminating program execution. It can be over-ridden with another statement or action.

You can use the ERRORLOG() function to identify errors in programs that you are developing and to customize error handling. Even after implementation, some errors, such as those relating to permissions and locking, are sometimes unavoidable. These errors can be trapped and recorded by these logging functions.

You can use error-logging functions with other 4GL features for instrumenting a program, by tracking the way the program is used. This functionality is not only valuable for improving the program but also for recording work habits and detecting attempts to breach security.

The following program fragment calls STARTLOG() in the MAIN program block. Here the ERRORLOG() function has a quoted string argument:

```
CALL STARTLOG("errors.txt")
...

CALL errorlog("Encountered unexpected error")
```

If its argument is not of a character data type (for example, a DECIMAL variable), invoking ERRORLOG() can itself produce an error.

Automatic error logging increases the size of the generated executable.

References

ERR_GET(), ERR_PRINT(), ERR_QUIT(), STARTLOG()

fgl_gethelp()

The fgl_gethelp() function mirrors the functionality of the ERR_GET function. It takes an integer expression as its argument and returns a character string containing the text of the 4GL or SQL error message for the specified argument.



Figure 57 - fgl_gethelp()

Usage

fgl_gethelp () is most useful when you are developing a program. The message that it returns is probably not helpful to the user of your 4GL application. The argument of fgl_gethelp () is typically the value of the **status** variable, which is affected by both SQL and 4GL errors, or else a member of the global **SQLCA.SQLCODE** record. The LET statement in the following code sample assigns the text of a 4GL error message to **my_error_text**, a CHAR variable:

```
MAIN
  DEFINE
    current_error_status integer,
    my_error_text char(100)

  DISPLAY "**** Function fgl_gethelp ( ) - Example error -1110 - no Form file found
  ****" AT 5,5

  LET current_error_status = -1110

  IF current_error_status < 0 THEN
    LET my_error_text = fgl_gethelp(current_error_status)
    ERROR my_error_text
  END IF

  CALL fgl_winmessage("Exit","Press any key to close this demo application","info")
END MAIN
```

The value of status can be assigned by a wide range of different statement variables. In order to avoid the status being changed unexpectedly it is always advisable to copy it as soon as possible so that the value you want to maintain does not get changed.

References

ERR_GET(), ERR_PRINT(), ERR_QUIT(), ERRORLOG()

showhelp()

The showhelp() function takes an integer expression as its argument and displays a runtime help message, corresponding to its specified SMALLINT argument, from the current help file.

Usage

The argument of showhelp() identifies the number of a message in the current help file that was specified in the most recently executed HELP FILE clause of the OPTIONS statement. (See "The HELP FILE Option" in the 4GL Statements volume for details of how to specify the current help file.)

The Help Menu

showhelp() opens the Help window and displays the first (or only) page of the help message text below a ring menu of help options. This menu is called the **Help** menu.

If the help message is too long to fit on one page, the **Screen** option of the **Help** menu can display the next page of the message. The **Resume** option closes the Help window and returns focus to the 4GL screen.

The Help File Displayed by showhelp()

To create a help file, you must use a text editor to create an ASCII file of help messages, each identified by a message number. The message number must be a literal integer in the range from -2,147,483,647 to +2,147,483,647 and must be prefixed by a period (.) as the first character on the line containing the number. No sign is required, but message numbers must be unique within the file. Just as in other literal integers, no decimal points, commas, or other separators are allowed. The NEWLINE character (or a NEWLINE RETURN pair) must terminate each message number.

The help message follows the message number on the next line. It can include any printable ASCII characters, except that a line cannot begin with a period. The text of the help message should contain information useful to the user in the context where SHOWHELP() is called. The message is terminated by the next message number or by the end of the file.

You must then use the **msgc** utility to create a runtime version of the help file that users can view. See the description of the **msgc** utility in the relevant volume of this Reference Guide. Here is a simple example of an ASCII help file for use with showhelp():

Code Sample: showhelp_function.msg

```
.100
This is my dummy help message 100

.101
This is my dummy help message 101

.102
This is my dummy help message 102
```

This message file can be displayed using the 4GL demonstration program shown below.

Sample Code: showhelp_function.4gl

```
MAIN
    DEFINE i integer
    OPTIONS
    HELP FILE "function_showhelp.erm"

    FOR i = 100 to 103
        call showhelp(i)
    END FOR

END MAIN
```

In interactive statements like CONSTRUCT, INPUT, INPUT ARRAY, PROMPT, and the COMMAND clause of a MENU statement, the effect of showhelp() resembles that of the Help key. The Help key, however, displays only the message specified in the current HELP clause. The following example uses INFIELD() with showhelp() to display field-dependent help messages:

```
INPUT ARRAY cont_arr FROM sc_cont.*
ON KEY(CONTROL-B)
    CASE
        WHEN INFIELD(cont_id)
            CALL showhelp(301)
        WHEN INFIELD(cont_comp)
            CALL showhelp(302)
        WHEN INFIELD(ind_type)
            CALL showhelp(303)
        ...
    END CASE
END INPUT
```

Reference

INFIELD()

startlog()

The startlog() function opens an error log file. It uses the syntax CALL startlog and takes a *filename* or other *variable* as its argument. That *filename*, can include a path name and file extension, is a string that specifies the error log file. The *variable* can be a CHAR or VARCHAR datatype that includes the *filename* details.

In Q4GL the startlog() function has two new arguments, *path* and *facility*. The *path* argument defines the path in which the logfile is to be written.

The path can include \$-style environment variable substitutions

If the filename begins with '!', then it will be appended to.

Also, certain other values are legal:

- filename,filename - the second filename specifies an input file
- stderr: - the standard error device
- null: - thrown away
- handle:nnn,[nnn] - a file handle (the optional,nnn specifies an input handle)

N.B: facility [default = "user"]

This indicates what facility the log information is being set for. Current facilities include:

- user
- debug (don't override this if you want to use the debugger)
- database - the database subsystem
- (an empty string) - the default logging path.
- database.<database-driver> - logging information only for the specified database
- Also new facilities can be defined as needed and accessed using the "systemlog" function

Usage

The following is a typical sequence to implement error logging:

1. Call `startlog()` in the MAIN program block to open or create an error log file.
When an error is encountered, use a LET statement with `ERR_GET(status)` to retrieve the error text and to assign this value to a program variable.

Use `errorlog()` to make an entry into the error log file.

The last two steps are not needed if you are satisfied with the error records that are automatically produced after `startlog()` has been invoked. After `startlog()` has been invoked, a record of every subsequent error that occurs during the execution of your program is written to 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 that 4GL appends 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.
```

You can also write your own messages in the error log file by using the `errorlog()` function.

With other 4GL features, the `startlog()`, `err_get()`, 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.

Unless you specify another option, `WHENEVER ERROR CONTINUE` is the default error-handling action when a runtime error condition is detected. The `WHENEVER ERROR CONTINUE` compiler directive can prevent the first SQL error from terminating program execution.

Specifying the Error Log File

If the argument of `STARTLOG()` is not the name of an existing file, the function will create a file of that name. If the file already exists, `STARTLOG()` opens it and positions the file pointer so that subsequent error messages can be added to it. 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.

Code Sample: `startlog_function.4gl`

```
DATABASE cms

MAIN
  DEFINE x TEXT, inp_char CHAR(1)
  CALL startlog("errors.txt")

  CALL errorlog ("Error 1 in my log")
  CALL errorlog ("Error 2 in my log")
  CALL errorlog ("Error 3 in my log")
  CALL errorlog (err_get (-999))

  LOCATE x IN FILE "errors.txt"
  DISPLAY x

  PROMPT "Press Enter to close the application" \
FOR inp_char
END MAIN
```

In this example, text written to the error log file merely shows that control of program execution has passed to the **`start_menu()`** function rather than indicating that any error has been issued.

For portable programs, the filename should be a variable rather than a literal string. As in other filename specifications, any literal backslash (\) that is required as a pathname separator must be entered as two backslashes.

References

`ERRORLOG()`, `ERR_GET()`, `ERR_PRINT()`, `ERR_QUIT()`

4GL Program

arg_val()

The ARG_VAL() function returns a specified argument from the command line that invoked the current 4GL application program. It can also return the name of the current 4GL program.



Figure 58 - ARG_VAL()

ARG_VAL() takes an *ordinal* as its argument. This should be a positive integer expression that is no higher than the number of arguments in the program. (You can use the NUM_ARGS() function to determine how many arguments follow the program name on the command line.)

Usage

This function provides a mechanism for passing values to the 4GL program through the command line that invokes the program. You can design a 4GL program that expects or allows arguments to appear in the command line, after the program name.

You can use the ARG_VAL() function to pass values to MAIN, which cannot have its own formal arguments. ARG_VAL() can be called from any part of the program, not just the MAIN statement.

You can use ARG_VAL() to retrieve individual arguments during program execution.

If $1 \leq \text{ordinal} = n$, ARG_VAL(n) returns the n^{th} command-line argument, as a character string. The value of *ordinal* must be between zero and the value returned by NUM_ARGS(), the number of command-line arguments.

Using ARG_VAL() with NUM_ARGS()

The built-in ARG_VAL() and NUM_ARGS() functions can pass data to a compiled 4GL program from the command line that invoked the program.

For example, suppose that the 4GL program called **program_arguments** can accept one or more account names as command-line arguments (in order to process account reports).

```
program_arguments St MegaSoft01 Querix02 Provia03
```

In either case, statements in the following program fragment use the ARG_VAL() function to store in an array of CHAR variables all the names that the user who invoked **program_arguments** entered as command-line arguments:

```
#Demo application for NUM_ARGS()
MAIN
  DEFINE inp_char CHAR(1),
    args ARRAY[8] OF CHAR(10),
    i, line_disp SMALLINT,
    str_disp char(50)

  FOR i = 1 TO NUM_ARGS()
    LET args[i] = ARG_VAL(i)
  END FOR

  ## just some example/dummy processing

  LET str_disp = "Total Number of passed arguments: " || NUM_ARGS()
  DISPLAY str_disp at 2, 5

  FOR i = 1 TO NUM_ARGS()

    LET line_disp = i + 5
    LET str_disp = "Argument " || i || " was " || args[i]
    DISPLAY str_disp at line_disp , 5

  END FOR

  PROMPT "Press any key to close this demo NUM_ARGS() application" FOR inp_char
END MAIN
```

After **program_arguments** is invoked by these command-line arguments, the NUM_ARGS() function returns the value 3. Executing the LET statements in the FOR loop assigns the following values to elements of the **args** array.

Variable	Value
args(1)	Megasoft01
args(2)	Querix01
args(3)	Provia01

Reference

NUM_ARGS()

num_args()

The NUM_ARGS() function takes no arguments. It returns an integer that corresponds to the number of command-line arguments that followed the name of your 4GL program when the user invoked it.

Usage

The built-in NUM_ARGS() function, and that of ARG_VAL(), can pass data to a compiled 4GL program from the command line that invoked the program.

For example, suppose that the 4GL program called **program_arguments** can accept one or more account names as command-line arguments (such as, to process account reports).

```
program_arguments St MegaSoft01 Querix02 Provvia03
```

In either case, statements in the following program fragment use the ARG_VAL() function to store in an array of CHAR variables all the names that the user who invoked **program_arguments** entered as command-line arguments:

Sample Code: num_args_function.4gl

```
MAIN
  DEFINE inp_char CHAR(1),
         args ARRAY[8] OF CHAR(10),
         i, line_disp SMALLINT,
         str_disp char(50)

  FOR i = 1 TO NUM_ARGS()
    LET args[i] = ARG_VAL(i)
  END FOR

  ## sample processing code follows:

  LET str_disp = "Total Number of passed arguments: " || \
  NUM_ARGS()
  DISPLAY str_disp at 2, 5

  FOR i = 1 TO NUM_ARGS()

    LET line_disp = i + 5
    LET str_disp = "Argument " || i || " was " || \
    args[i]
    DISPLAY str_disp at line_disp , 5

  END FOR

  PROMPT "Press any key to close this demo NUM_ARGS() application" FOR inp_char
END MAIN
```


After **program_arguments** is invoked by these command-line arguments, the NUM_ARGS() function returns the value 3. Executing the LET statements in the FOR loop assigns the following values to elements of the **args** array.

Variable	Value
args(1)	Megasoft01
args(2)	Querix01
args(3)	Provia01

Reference

ARG_VAL()

base.APPLICATION Class

The Application class is designed to allow access to the application environment and internal information, such as program name, execution directory, command line arguments, FGLPROFILE details, information about the program workflow.

The class supports a set of methods which can be applied to the currently run program. It does not support the creation of class objects, this they need not be declared. The class methods are called with the "base.Application." prefix preceding them:

```
base.Application.Method()
```

The Application class does not need to be defined explicitly. The application methods are applied to the application which is currently running.

Command line arguments

Application class supports two methods dealing with the application command line performance. These methods are used to retrieve information about the arguments, passed to the application. Their functions are identical to the functionality of the `arg_val()` and `num_args()` functions.

GetArgumentCount()

The `getArgumentCount()` method can be used to return the total number of arguments passed to the application. The returned value is an integer. The method does not need any arguments. It's result is the same as the result of the `num_args()` build-in function.

```
DEFINE i INT
LET i = base.Application.getArgumentCount()
```

If no arguments were passed to the application, the method will return a NULL value.

GetArgument()

The `getArgument()` method returns the value of the specified argument. The method needs an integer specifying the argument position:

```
DEFINE i, ii INT,
      arg DYNAMIC ARRAY OF STRING
LET i = base.Application.getArgumentCount()
FOR ii = 1 TO i
LET arg[ii] = base.Application.getArgument(ii)
END FOR
```

The first argument of those passed to the application will have the position 1. The `GetArgument()` method returns the application name if the integer passed to it is zero.

Application Information

There are methods that can be used to return the application name and the directory of the executable file. These are `getProgramName()` and `getProgramDir()` methods.

GetProgramName()

The `GetProgramName()` method is used to retrieve a character string containing the name of the current application. The method does not need any arguments:

```
DEFINE name STRING
LET name = base.Application.getProgramName()
```

GetProgramDir()

The `GetProgramName()` method is used to retrieve a character string containing the directory path to the application executable file. This would be the location of the application server on your machine and the subfolders, if they are present. This method doesn't need any arguments.

```
DEFINE path STRING
LET path = base.Application.getProgramDir()
```

Remember, that the location of the application server and the executable file depends on the system used.

GetFglDir()

The `GetFglDir()` method is used to retrieve the Lycia installation path specified in the `LYCIA_DIR` environment variable.

```
DEFINE lyciadir STRING
LET lyciadir = base.Application.getProgramDir()
```

GetResourceEntry()

This method refers to the `fglprofile` file and retrieves the value of the parameter whose name was sent as the method argument. It accepts a single argument of a character data type containing a name of a `fglprofile` parameter and returns a string containing the parameter value. The example below will return either `TRUE` or `FALSE`, if this option was set in the `fglprofile` and if the option is not set or is commented, it will return `NULL`.

```
LET str = base.Application.GetResourceEntry("gui.window.systemmenu.edittheme")
```

GetStackTrace()

The `GetStackTrace()` method is used to keep track of the functions which are invoked by the application at runtime. The method, when called, displays a formatted list of the invoked functions and the `.4gl` files in which they are located.

The method does not need any arguments.

```
base.Application.getStackTrace()
```

It is convenient to invoke this method from WHENEVER ERROR or WHENEVER ANY ERROR statements to find out instantly which function caused the error:

```
MAIN
...
WHENEVER ANY ERROR CALL track_func()
...
END MAIN

FUNCTION track_func()
    DISPLAY base.Application.getStackTrace()
END FUNCTION
```

If an error occurs, the program control goes to the track_func() function which invokes the getStackTrace() method. The output in this case may be similar to the following:

```
track_func() at runner.4gl:50
get_items_list() at items.4gl:321
display_items() at items.4gl:450
search_customer() at customers.4gl:763
```

ui.INTERFACE Class Used for MDI Mode

This class can be used for creating MDI containers. It uses class methods, thus no object of this class is required. To manage MDI containers a set of methods must be set, setting a single method will not be sufficient to create an MDI container and will be ignored by the program. The general syntax of the methods invocation is:

```
CALL ui.Interface.Method()
```

setName()

The setName() method is used to set the name for a MDI container or a child. It accepts a single parameter of a character data type that is the name. The name given to an MDI container in this way can later be used by the child applications to identify the parent container for them using the setContainer() method. The name given to a child application will be useful for counting the running application instances.

getName()

The getName() method is used to retrieve the name of the application, which was previously set by the setName() method.

setText()

The setText() method is used to specify the main title for the application. This title will be displayed in the application main window. If used for the MDI container application, it will set the title for the main MDI window. It accepts a single parameter of a character data type.

getText ()

The getText () retrieves the image previously set by setText() method.

setImage()

The setImage() method is used to specify the icon for the application title bar. This icon will also be seen in taskbars when the child applications are minimized. If applied to the main MDI window, it will appear

at the left corner of the title bar of the MDI container. It accepts the filename with the optional path as a parameter. The file must be located on the application server.

getImage()

The GetImage() retrieves the image previously set by SetImage() method.

setType()

The SetType() method is used to configure MDI and is used to specify the type of the application.

The method can pass one of the following values:

- Normal - the application will be run as an independent application outside the MDI container.
- Container - the application will be run as an MDI container.
- Child - the application will be run as a child application within the specified MDI container. If this option is set, the application cannot be run independently and will throw an error if you try to.

getType()

The GetType() method is used to retrieve the application type set by the SetType() method.

setSize()

The SetSize() method is used to specify the initial size of an MDI application parent container window. The method needs two arguments to be passed: window height and window width, which can be represented by integers as well as by string values:

```
CALL ui.Interface.SetSize(15, 60)
```

By default, the method arguments specify the size in character grid cells. However, you can specify the size in pixels by adding the px unit:

```
CALL ui.Interface.setSize("700px", "700px")
```

The SetSize() method can also be used instead of fgl_setsize() function to change window size in SDI mode. If it is not called for the MDI container, it has a standard size.

setContainer()

The SetContainer() method is used in a child application (defined by the SetType() method as "child"). It specifies the name of the MDI container for the child application to use. It accepts a single parameter which should correspond to the name of the MDI container set by the setName() method in the main MDI application.

getContainer()

The GetContainer() method is used to retrieve the name of the application parent container set by the setContainer method previously.

getChildCount()

The GetChildCount() method is used to get the number of child applications which belong to the current parent MDI container. This method accepts no parameters. It does not count all the child applications linked to the given MDI container, it counts only the currently running child applications. This method must be called from within the program serving as the MDI container. It returns an integer specifying the number of all child applications currently running. It returns 0, if no child applications are running.

getChildInstances()

This method counts the number of instances of the same child application currently running within the current MDI container. It accepts a single parameter that is the name of a child application set by the setName() method. It counts only the child applications with the given name, any other child applications are ignored, and returns an integer number. it returns 0, if no applications with such name are currently running.

loadToolbar()

The LoadToolbar() method is used to load a toolbar. The syntax of its invocation is as follows:

```
CALL ui.Interface.LoadToolbar (file_name)
```

where:

- *file_name* = a STRING variable standing for the file name containing the toolbar to be loaded definition and the path which is optional (the file should be located on an application server)


Usage

Here is an example of the loadToolBar() method calling:

```
#loads the toolbar defined in mytoolbar_file  
CALL ui.Interface.LoadToolBar("mytoolbar_file")
```

It is advisable not to specify the file extension. It is automatically added by the runtime system when the file is determined: initially the system searches for the file with the name specified with the .tb2 extension, if it does not exist the .fm2 file is being searched for. In case the file with the name specified does not exist, the corresponding runtime error occurs.

However, it is still possible to specify the file containing the toolbar definition as mytoolbar_file.fm2. If there is no such a file, an error will occur at runtime.

	<p>The third-party files containing the toolbar definition are converted into the .tb2 format. For more details regarding file conversion, please, refer to the qxcompat description, the Developer Guide.</p>
---	--

loadTopMenu()

The LoadTopMenu() method is used to load a top menu. The syntax of its invocation is as follows:

```
CALL ui.Interface.LoadTopMenu (file_name)
```

where:

- *file_name* = a STRING variable standing for the file name containing the top menu to be loaded definition and the path which is optional (the file should be located on an application server)

Usage


Here is an example of the loadTopMenu() method calling:

```
#loads the top menu defined in my_topmenu_file  
CALL ui.Interface.LoadTopMenu("my_topmenu_file")
```

If there is a top menu already specified for the target form, it will be replaced by the one loaded.

It is advisable not to specify the file extension. It is automatically added by the runtime system when the file is determined: initially the system searches for the file with the name specified with the .tm2 extension, if it does not exist the .fm2 file is being searched for. In case the file with the name specified does not exist, the corresponding runtime error occurs.

However, it is still possible to specify the file containing the top menu definition as my_topmenu_file.fm2. If there is no such a file, an error will occur at runtime.

	<p>The third-party files containing the top menu definition are converted into the <code>.tm2</code> format. For more details regarding file conversion, please, refer to the <code>qxcompat</code> description, the Developer Guide.</p>
---	---

loadStartMenu()

The `LoadStartMenu()` method is used to load a start menu. This method is an alternative to an application launcher menu creation.

While the application launcher menu functions create the menu within the 4GL source, the `LoadStartMenu()` method, in its turn, loads a file storing the start menu options. Though such an approach offers less flexibility, it allows the source code reduction.

The syntax of its invocation is as follows:

```
CALL ui.Interface.LoadStartMenu (file_name)
```

where:

- *file_name* = a STRING variable standing for the file name containing the start menu to be loaded definition and the path which is optional (the file should be located on an application server)


Usage

Here is an example of the `loadStartMenu()` method calling:

```
#loads the start menu defined in my_startmenu_file
CALL ui.Interface.LoadStartMenu("my_startmenu_file")
```

It is advisable not to specify the file extension. It is automatically added by the runtime system when the file is determined: initially the system searches for the file with the name specified with the `.sm2` extension, if it does not exist the `.fm2` file is being searched for. In case the file with the name specified does not exist, the corresponding runtime error occurs.

However, it is still possible to specify the file containing the start menu definition as `my_startmenu_file.fm2` If there is no such a file, an error will occur at runtime.

	<p>The third-party files containing the start menu definition are converted into the <code>.sm2</code> format. For more details regarding file conversion, please, refer to the <code>qxcompat</code> description, the Developer Guide.</p>
---	---

Using the MDI mode with ui.Interface Class

To use an MDI mode you need to have the following:

- the main program which will serve as the MDI container
- the child programs(1 or more)
- the start menu file which specifies the child program calls

You can use the following steps to create an MDI container with one child program:

1. Create the main program that will serve as a container. It must contain the following methods:
 - a. setName() method must be used to name the MDI container.
 - b. setType() must be used to specify that this program will be an MDI container, its parameter must be "container".
 - c. setStartMenu() method specifies the commands used to call the child programs.
 - d. the rest of the methods are optional. The example below also sets the size and the title of the MDI container

```
MAIN
CALL ui.Interface.setName("cont")
CALL ui.Interface.setType("container")
CALL ui.Interface.setText("MDI Container")
CALL ui.Interface.setSize("600px","1000px")
CALL ui.Interface.loadStartMenu("startmenu") -- see step 3
MENU "Main"
    COMMAND "count applications"
    DISPLAY "Total applications running: ", ui.Interface.GetChildCount()
    DISPLAY "Applications with name ""child"": ",
        ui.Interface.GetChildInstances("child")
    COMMAND "Exit" EXIT MENU
END MENU
END MAIN
```

2. Create a child program (e.g. called "child"). The child program must contain the following ui.Interface methods, otherwise it will still be called and will operate normally, but it will be opened outside the MDI container.
 - a. setName() method is used to identify the child program.
 - b. setType() method is used to specify the program as a child program and should be set to "child". If this method is absent, the program will still be opened, but it will be opened outside the MDI container.

- c. `setContainer()` method is necessary to link the child program to a specific MDI container. If this method is absent or specifies a wrong container name, the program will be opened outside the MDI container, when called.



Note: The child application will throw a runtime error, if the `setContainer()` method is set and you try to launch this application independently like a separate program and not from within the MDI container.

- d. Other methods such as `setText()` are optional.

```
MAIN
DEFINE a CHAR
CALL ui.Interface.setName("tree")
CALL ui.Interface.setType("child")
CALL ui.Interface.setText("tree")
CALL ui.Interface.setContainer("cont") -- the container name set in step 1
PROMPT "Press any key..." FROM CHAR a
END MAIN
```

3. Create a start menu file with the reference to the child program (e.g. call it "startmenu"). For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<StartMenu text="MDI">
  <StartMenuGroup text="Call Programs">
    <StartMenuCommand text="child" exec="child.exe"/>
  </StartMenuGroup>
</StartMenu>
```

4. Compile and deploy the child program. The child program must be compiled and deployed to the application server before the main program is run. Child applications do not get deployed automatically.
5. Compile and run the main program. Use the menu option Call Programs - Child to invoke the child application.

4GL Language

ascii

The ASCII operator converts an integer operand into its corresponding ASCII character, using the syntax: *ASCII number*. Where *number* is an integer expression that returns a positive number that is within the range of ASCII values.

Usage

You can use the ASCII operator to evaluate an integer to a single character; and it is particularly useful for displaying CONTROL characters.

The following DISPLAY statement rings the terminal bell (ASCII value of 7):

```
MAIN

DEFINE
    bell_tone CHAR(1),
    inp_char CHAR(1)

LET bell_tone = ASCII 7

DISPLAY bell_tone  -- Make the bell tone
#Note - This is just an example
#we strongly recommend to use the function fgl_bell() for a bell_tone


DISPLAY "This bell tone example works only with the DISPLAY ... NOT with Display
AT" at 4,5
DISPLAY "The bell tone does not work in a GUI environment" at 5,5
DISPLAY "To get a bell tone in GUI and character mode" at 6,5
DISPLAY "you should use the functin fgl_bell()" at 7,5

DISPLAY "The Control character: " at 10, 5
DISPLAY bell_tone at 10, 30  -- Display the control character

PROMPT "Press any key to close this demo application" FOR inp_char
END MAIN
```

The next REPORT program block fragments show how to implement special printer or terminal functions. They assume that, when the printer receives the sequence of ASCII characters 9, 11, and 1, it will start printing in red, and when it receives 9, 11, and 0, it will revert to black printing. The values used in the example are hypothetical; refer to the documentation for your printer or terminal for information about which values to use.

```
FORMAT
FIRST PAGE HEADER
  LET red_on = ASCII 9, ASCII 11, ASCII 1
  LET red_off = ASCII 9, ASCII 11, ASCII 0
ON EVERY ROW
...
PRINT red_on,
"Your bill is overdue.", red_off
```

	<p>4GL cannot distinguish between printable and non-printable ASCII characters.</p> <p>Be sure to account for non-printable characters when using the COLUMN operator to format the screen or a page of report output.</p> <p>Different printers and on-screen rendering applications will handle spaces with control characters in different ways. It may only be possible to determine the appropriate spacing by trial and error.</p>
---	--

The ASCII Operator in PRINT Statements

To print a null character in a report, call the ASCII operator with 0 in a PRINT statement. For example, the following statement prints the null character:

```
PRINT ASCII 0
```

ASCII 0 only displays the null character within the PRINT statement. If you specify ASCII 0 in other contexts, it returns a blank space.

References

`fgl_keyval()`, `fgl_lastkey()`

column

COLUMN specifies the position in the current line of a report where output of the next value in a PRINT statement begins, or the position on the 4GL screen for the next value in a DISPLAY statement.

The COLUMN statement takes a left-offset argument, which can be either an integer expression in a report, or a literal integer in a DISPLAY statement. That integer describes where the next output character should appear, measured in characters from the left-hand margin.

Usage

The PRINT statement and the DISPLAY statement will, when no form is open, display 4GL variables whose widths are determined by their declared data types, unless the CLIPPED or USING keywords are used.

Data Type	Default Display Width (Measured in Characters)
CHAR	The length described in the data type declaration
DATE	10
DATETIME	Between 2 and 25, implied by the data type declaration
DECIMAL	$(2 + m)$, where m is the precision in the data type declaration
FLOAT	14
INTEGER	11
INTERVAL	Between 3 and 25, implied in the data type declaration
MONEY	$(3 + m)$, where m is the precision in the data type declaration
SMALLFLOAT	14
SMALLINT	5
VARCHAR	The maximum length described in the data type declaration

In a REPORT program block or in A DISPLAY statement that outputs data to the 4GL screen, you can use the COLUMN operator to control precisely the location of items within a line. The COLUMN operator is often a requirement for the output of tabular information, and it is convenient for many other uses.

The *left-offset* value specifies a character position offset from the left margin of the 4GL screen or the currently executing 4GL report. In a report, this value cannot be greater than the arithmetic difference (*right margin - left margin*) for explicit or default values in the OUTPUT section of the REPORT definition (for more information, see the "OUTPUT Section" of the Forms & Reports volume). The syntax of 4GL

expressions that return whole numbers are described in the "Integer Expressions" section of the Datatypes volume.

If the printing position in the current line is already beyond the specified *leftoffset*, the COLUMN operator has no effect.

COLUMN in DISPLAY Statements


4GL calculates the left-offset from the first character position of the 4GL screen. For example, in the following statements the display strings are printed at increasing indents.

```
MAIN
  DISPLAY "1234567890123456789012345678901234567890"

  DISPLAY COLUMN 10, "I start at column 10"
  DISPLAY COLUMN 20, "I start at column 20"
  DISPLAY COLUMN 30, "I start at column 30"

  SLEEP 3
END MAIN
```

Output from each DISPLAY statement begins on a new line.

	<p>You cannot use COLUMN to send output to a screen form. Any DISPLAY statement that includes the COLUMN operator cannot also include the AT, TO, BY NAME, or ATTRIBUTE clause. When you include the COLUMN operator in a DISPLAY statement, you must specify a literal integer as the left-offset, rather than an integer expression.</p>
---	--

COLUMN in PRINT Statements

When you use the PRINT statement in the FORMAT section of a report, by default, items are printed one following the other, separated by blank spaces. The COLUMN operator can override this default positioning. 4GL calculates the left-offset from the left margin. If no left margin is specified in the OUTPUT section or in START REPORT, the left-offset is counted from the left margin of the page.

If the following PRINT statements (with COLUMN and SPACE specifications) were part of a report that sent output to the 4GL screen, the output would resemble that of the DISPLAY statements in the previous example:

```
MAIN
  START REPORT repl TO SCREEN
  OUTPUT TO REPORT repl()
  FINISH REPORT repl
END MAIN
```

```
REPORT repl()  
  OUTPUT LEFT MARGIN 0  
  
  FORMAT  
    ON EVERY ROW  
      PRINT "1234567890123456789012345678901234567890"  
      PRINT "Start at column 1"  
      PRINT COLUMN 10, "Start at column 10"  
      PRINT COLUMN 20, "Start at column 20"  
      PRINT COLUMN 30, "Start at column 30"  
      PRINT COLUMN 40, "Start at column 40"  
  END REPORT
```

References

ASCII, CLIPPED, SPACE, USING

fgl_copyblob()

fgl_copyblob(destination, source)

The fgl_copyblob() function performs a value copy of one blob to another. In the 4GL language, all blob assignments are performed by reference which means that it is not possible to copy the contents of one blob to another. For example

```
LOCATE x IN FILE 'a.txt'

LOCATE y IN FILE 'b.txt'

LET b = a
```

In this example, the variable y becomes a reference to the file 'a.txt' as a result of the assignment operation. To be able to copy the contents of 'x' into 'y', we can use the function fgl_copyblob(). For example:

```
LOCATE x IN FILE 'a.txt'

LOCATE y IN FILE 'b.txt'

fgl_copyblob(y, x)

FREE x

FREE y
```

Usage

The function takes two arguments, both of which must be of type BYTE or TEXT. The first argument specifies the destination for the copy operation, and the second specifies the source of the copy operation.

fgl_random()

Function fgl_random() generates a random number between (and including) two minimum and maximum numbers passed to it.

Usage

This function takes two parameters – min and max. These values are the start and end values of the range between which you would like to find a random number, inclusive of min and max. The value returned is of the FLOAT data type, to receive an integer random number assign the value to an INTEGER variable.

For example, to generate a random number in the range 1 to 10:

```
LET value = fgl_random(1, 10)
```

lineno

The LINENO statement returns the number of the line within the page that is currently printing. (This operator can appear only in the FORMAT section of a REPORT program block.)

Usage

This operator returns the value of the line number of the report line that is currently printing. 4GL computes the line number by calculating the number of lines from the top of the current page, including the TOP MARGIN.

For example, the following program fragment examines the value of LINENO. If this value is less than 9, a PRINT statement formats and displays it, beginning in the 10th character position after the left margin.

```
MAIN
START OUTPUT repl TO SCREEN
    OUTPUT TO REPORT repl ()
    OUTPUT TO REPORT repl ()
    OUTPUT TO REPORT repl ()
    OUTPUT TO REPORT repl ()
    OUTPUT TO REPORT repl ()
    OUTPUT TO REPORT repl ()
    OUTPUT TO REPORT repl ()
END MAIN

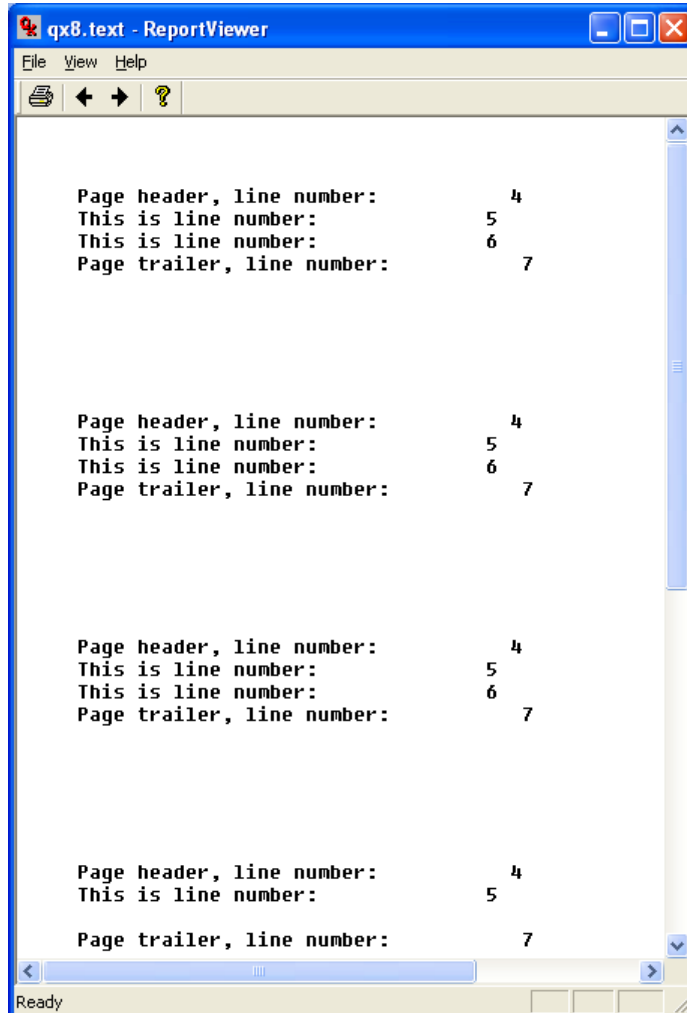
REPORT repl ()
    OUTPUT PAGE LENGTH 10

    FORMAT
        PAGE HEADER
            PRINT "Page header, line number: " \
COLUMN 25, LINENO
            ON EVERY ROW
                PRINT "This is line number : " \
COLUMN 25, LINENO
        PAGE TRAILER
            PRINT "Page trailer, line number " \
COLUMN 25, LINENO
END REPORT
```

You can specify LINENO in the PAGE HEADER, PAGE TRAILER, and other report control blocks to find the print position on the current page of a report.

4GL is not able to evaluate the LINENO operator outside the FORMAT section of a REPORT program block. If this value has to be referenced within other program blocks of your application, the value that LINENO returns must be assigned to a variable that is not local to the report.

The output of the demonstration application code shown on the previous page, is shown here:



```
Page header, line number:      4
This is line number:          5
This is line number:          6
Page trailer, line number:     7

Page header, line number:      4
This is line number:          5
This is line number:          6
Page trailer, line number:     7

Page header, line number:      4
This is line number:          5
This is line number:          6
Page trailer, line number:     7

Page header, line number:      4
This is line number:          5
Page trailer, line number:     7
```

Figure 59 – Demonstration Output Code

Reference

PAGENO

pageno

The PAGENO statement returns a positive whole number, corresponding to the number of the page of report output that 4GL is currently printing. (PAGENO is valid only in the FORMAT section of a REPORT program block.)

Usage

This operator returns a positive integer whose value is the number of the page of output that includes the current print position in the currently executing report.

For example, the following program prints the page number on the top and bottom of each page.

Code Sample: pageno_report_function.4gl

```
MAIN
  START REPORT repl TO SCREEN
  OUTPUT TO REPORT repl ()
  OUTPUT TO REPORT repl ()
  OUTPUT TO REPORT repl ()
  FINISH REPORT repl
END MAIN

REPORT repl()
  OUTPUT PAGE LENGTH 10

  FORMAT
    PAGE HEADER
      PRINT "This is start of page number: ", PAGENO
    ON EVERY ROW
      PRINT "This is the page content"
      SKIP TO TOP OF PAGE
    PAGE TRAILER
      PRINT "This is end of page number: ", PAGENO
  END REPORT
```

You can include the PAGENO operator in PAGE HEADER and PAGE TRAILER control blocks and in other control blocks of a report definition to identify the page numbers of output from a report.

4GL cannot evaluate the PAGENO operator outside the FORMAT section of a REPORT program block. If some other program block of your 4GL application needs to reference the value that PAGENO returns, the report must assign that value to a program variable whose scope of reference is not local to the report.

Reference

LINENO

sizeof()

The SIZEOF() statement is used to get the size of an array or char variable. It will return either the number of elements allowed in an array, or the maximum size of a char variable.

Usage

In Querix4GL it is possible to get the size (dimension) of an array or char variable. The SIZEOF() operator takes a variable name as its argument and returns the number of elements in that variable, providing it is of type array or char. If the operator is called with any other type of variable then a zero will be returned.

An example of this operator in action follows:

```
DEFINE x ARRAY[10] OF CHAR(22)
```

The above has defined an array called 'x' that contains 10 elements, each of which is a 22 character char type.

```
DISPLAY SIZEOF(x)
```

Refers to the actual array and will return its size, i.e., the maximum number of elements within it, which in this case is 10.

```
DISPLAY SIZEOF(x[1])
```

The above will return 22 as this is referring to the actual element at position '1' in the array 'x'. As has been defined – each element is a 22 character char.

Now, say we have a multi-dimensional array defined as below:

```
DEFINE x ARRAY[10, 20] OF CHAR(22)
```

It is possible to discover the size of each dimension using the sizeof() function as follows:

```
DISPLAY SIZEOF(x)
```

This will return 10 as this is the size of the first dimension.

```
DISPLAY SIZEOF(x[2])
```

This will return 20 as it is referring to the size of dimension 1 (with the first dimension considered as dimension 0). Every subsequent dimension is referred to in the same way, so the next dimension in a three dimensional array would have 2 as an index and so on.

```
DISPLAY SIZEOF(x[1, 2])
```

The above will return 22 as it referring to the size of the element at position 2 in dimension 1.

Web Services Functions

This chapter will demonstrate all of the web service functions that have been introduced for Lycia. It includes the traditional functions which were previously used in Hydra to manipulate the web services and are still supported in Lycia for the compatibility.

It also includes the new methods which are used with the WEBSERVICE object data type. The new methods reduce the amount of code needed and make the webservices more flexible and easy to use. WEBSERVICE variables are declared in the same way as variables of simple data types. They are used as described [above](#).

fgl_wsdl_call()

fgl_wsdl_call(<web_service_Meta_data_file_basename>)

First thing to do is LOAD the WS Meta file in order to be able to use the service. The service cannot be used unless this LOAD statement is called. The code to do this follows:

```
CALL fgl_wsdl_call("WS_Meta_data_file")
```

The meta-data file should be the base name of the file and not it's full path location. Loading the WS Meta file provides all the necessary information to be able to use the remote service. It provides the service name, which can have one or more port names, each of which can have one or more operations. For each operation, the input and output data structure is also specified, with mappings of the XML to 4GL data types for each primitive.

fgl_ws_call()

fgl_ws_call("service name", "port name", "operation", [parameters])

This is the simplest method of using an operation offered by a published Web Service. In order to use this function properly, the 4GL developer must know beforehand the expected input and output. This is because there is no error information returned should the service fail to execute. It will always return the expected return values, if these return values are missing or incorrect then they will be replaced by NULL values. The function requires the service name, the port name, the specific operation name and, optionally, any input parameters. This information is required in order to pinpoint a unique operation.

An example function call:

```
CALL fgl_ws_call("Web_Service_Simple", "Simple", "add", 2, 3)
```


fgl_ws_create()

fgl_ws_create("handle", "service name", "port name", "operation")

This function creates an instance of the Web Service Client. This instance is then referred to in the 4GL code by its handle and can be called any number of times while that handle is still open. It can be executed, have its input parameters changed, have its output values checked, and have its error information checked any number of times whilst still open. This function will return a TRUE value if the instance was created successfully and a FALSE value if it fails to create an instance.

An example function call:

```
CALL fgl_ws_create("myAdd", "Web_Service_Simple", "Simple", "add")
```

The handle can be an arbitrary value, but must remain consistent throughout the following 4GL Web service call.

fgl_ws_setparameters()

fgl_ws_setparameters("handle", [parameters])

Use this function once the instance has been created to alter the input parameters required by the operation. These parameters must be in the correct order as defined in the FGL Meta file. It is advised to keep to the same type too, although on conversion back to XML, type conversion will take every care to ensure correct types. If conversion fails, then NULL will be returned in its place; NULL will also be returned in the case of too few return parameters.

An example function call:

```
CALL fgl_ws_setparameters("myAdd", 4, 6, 8)
```

fgl_ws_getdata()

fgl_ws_getdata("handle")

This function will retrieve the output data that had been returned on the last occasion that the instance had been executed. The return values for this function are the 4GL representations of the data returned by the Web Service call. If the instance has not been executed, or if the instance returns no value, then the function will return 0.

An example function call:

```
CALL fgl_ws_getdata("myAdd")
```

fgl_ws_error()

fgl_ws_error("handle", ["type"])

Use this function to retrieve any error information that the remote service may have returned if an error has occurred on execution. The optional type refers to the portion of the error message to receive; one of "code", "reason", or "detail". If no type is specified, "reason" is the default. If an incorrect handle is specified, or there has been no error, this function will return NULL.

An example function call:

```
CALL fgl_ws_error("myAdd", "code")
```

fgl_ws_execute()

fgl_ws_execute("handle")

This function will execute the specified instance of the Web Service client. Any input data required should be set before this instance is executed as any unset parameters will be supplemented with NULL data. There are two types of service that can be executed – 'fire and forget', and 'send and receive'. For a 'send and receive' service, this function returns TRUE on success and FALSE on failure. A failure can either be an invalid handle, or an error returned from the remote service. If the service is a 'fire and forget', no error message will be received and the only failure is an incorrect handle specified.

An example function call:

```
CALL fgl_ws_execute("myAdd")
```

fgl_ws_close()

fgl_ws_close("handle")

This function closes the named instance of a Web Service client. Once closed, the handle is no longer valid. This function will return TRUE on success and FALSE on failure, i.e., if an invalid handle is specified.

An example function call:

```
CALL fgl_ws_close("myAdd")
```

fgl_ws_set_option()

fgl_ws_set_option("handle", "option_name", option_value)

This function is used to set the option of 4GL web service client. This function should be called after `fgl_ws_create` function, but before `fgl_ws_execute` function.

Currently only one option have been defined to switch on/off the namespace of each input parameter xml element in the SOAP request. For this option, `option_name` will be `"parameter_namespace"`, `option_values` can be 0, 1 or other integer. 0 means to switch off the namespaces for these input parameter xml elements. 1 or other integer means to switch on the namespaces for these input parameter xml elements. If you do not set this option, Querix 4GL web service framework will switch on the option by default.

If the web service to be called is .DOT NET web service, `parameter_namespace` option must be switched on. Otherwise you will get unexpected return values. If the web service to be called is SUN J2EE web service, whether the `parameter_namespace` option should be switched on or not depends on whether the namespaces for the input parameters of these J2EE web service operations have been explicitly specify or not using annotation. If they have been explicitly specified, the `parameter_namespace` option should be switched on. Otherwise the `parameter_namespace` option should be switched off.

execute()

variable.Execute([parameters])

This is a method used with a WEBSERVICE variable to execute a webservice operation. It can be called for a WEBSERVICE variable which has already been used with the LoadMeta() method and contains the webservice operation selected by the SelectOperation() method. It accepts optional arguments, which represent the parameters passed to the stub from the calling routine. If no parameters are passed, they are replaced by NULL values. It can return some values, if the webservice operation returns something.

An example method call:

```
CALL ws_var.Execute(input_record.*) RETURNING output_record.*
```


loadMeta()

`variable.LoadMeta("meta_data_file.xml")`

This method can be used with a variable of the WEBSERVICE data type which has just been declared, it does not require other methods to be executed beforehand. It loads the WSDL meta data definition file of a webservice which the programmer wants to use and takes the name of this file (with the .xml extension) as an argument.

An example method call:

```
CALL ws_var.LoadMeta("ws_client.xml")
```

selectOperation()

variable.LoadMeta("service name", "port name", operation name")

This method is used together with a variable of the WEBSERVICE data type to select a web service operation to be invoked. You need to execute the LoadMeta() method for this variable before you would be able to use this method. It accepts three arguments: the name of the web service currently in use, the port name (the LyciaStudio offers two ports SOAP 1.1 and SOAP 1.2, your webservice client can use both of them or any particular one), and the name of the web operation you want to execute.

An example method call:

```
CALL ws_var.SelectOperation("LengthUnit", "LengthUnitSoap",  
                             "ChangeLengthUnit")
```