Lycia Development Suite

Lycia II Forms Guide
Version 6 – March 2012
Revision number 1.40

Querix

Lycia Forms Guide

March 2012

Part Number: 007-050-140-012

Querix4GL Reference – Forms & Reports

Part Number: 007-050-140-012

Published by:

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

Publication history:

April 2005: First Draft

Previously the draft of Forms and Reports volume

June 2005: Updated for 'Hydra4GL' 4.2

January 2008: Updated for ‘Hydra4GL’ 4.3

February 2009: Updated for ‘Hydra4GL’ 4.4

November 2010 Updated for Lycia

September 2011 Updated for Lycia II

Last Updated:

March 2012

Documentation written by:

Anthony Davey/ Sean Sunderland/ Gemma Davis/Valeriy Pantyukhin

Notices:

Table of Contents

ABOUT THIS GUIDE **1**

CONVENTIONS USED IN THIS BOOK 1

Typefaces and Icons *1*

FORM DRIVERS, FIELDS & NAVIGATION **3**

THE REQUIRED 4GL CODE FOR FORMS – FORM DRIVER 4

Display a Form *4*

Form Position *5*

Displaying and retrieving data from Forms *5*

FORM FIELDS 7

Appearance of Fields *7*

Size and Position and identifier tags of Fields *8*

Field identifiers (variable names) *9*

Field Attributes *9*

FIELD NAVIGATION 9

Navigation within a field, Field Width and Input Field Width *9*

Navigation among Form Fields *10*

STRUCTURE OF A FORM SPECIFICATION FILE **13**

DATABASE SECTION 16

Usage *16*

SCREEN SECTION 18

Size properties *18*

The Screen Layout *25*

Display Fields *26*

Static Labels and lines in Forms *29*

TABLES SECTION 32

Usage *32*

Table Aliases *32*

ATTRIBUTES SECTION 34

Usage *35*

Fields Linked to Database Columns *36*

Usage *36*

BYTE Data (BLOB) *39*

FORMONLY Fields *40*

The Data Type Specification *41*

Multiple-Segment Fields (Wordwrap) *42*

Subscripted Fields *42*

Field Attributes *42*

Field Attribute Syntax *44*

INSTRUCTIONS SECTION 45

Screen Records ... 46
Screen Arrays ... 49
GRID ARRAY ... 52
Screen Array with graphical widgets ... 54
Field Delimiters ... 58
Default Attributes ... 60
Precedence of Field Attribute Specifications ... 62
Default Attributes in an ANSI-Compliant Database ... 63
KEYS SECTION ... 64
The button label ... 64
The button icon ... 64
The ORDER keyword ... 64
The STATIC keyword ... 65
ACTIONS SECTION ... 66
The STATIC keyword ... 66
VISUAL FORM OBJECTS ... 67
STATIC GRAPHICAL LINES (GRAPHICAL CHARACTERS) ... 69
STATIC LABEL ... 72
DYNAMIC LABEL WIDGET ... 73
TEXT FIELD BOX (NORMAL FIELD) ... 76
CHECK BOXES WIDGET ... 78
RADIO BUTTONS WIDGET ... 82
FUNCTION BUTTON FIELD WIDGET ... 87
COMBO BOX FIELD WIDGET ... 90
CALENDAR WIDGET ... 94
FILE & URL BROWSER WIDGET ... 97
BUTTON WIDGET ... 103
IMAGE BUTTON ... 108
IMAGE WIDGET ... 112
HOTLINK WIDGET ... 116
FORM FIELD ATTRIBUTES ... 120
ACTION/ ACTIONS ... 120
Action Events ... 120
Toolbar Buttons ... 122
AUTONEXT ... 126
CENTER ... 129
CENTURY ... 132

CLASS 137
COLOR 139
Specifying Logical Conditions with the WHERE Option *142*
Boolean Expressions in 4GL Form Specification Files *144*
Specifying Ranges of Values and Set Membership *145*
Data Type Compatibility *146*
Data Type Conversion in 4GL Boolean Expressions *146*
The Display Modes *147*
CONFIG 148
COMMENTS 149
The Position of the Comment Line *150*
DEFAULT 153
USAGE 153
Literal Values *153*
Built-In 4GL Operators and Functions as Values *154*
DISPLAY LIKE 157
DOWNSHIFT 158
FORMAT 160
Formatting Number Values *160*
Formatting DATE Values *161*
INCLUDE 164
Ranges of Values *165*
FORMONLY Fields *166*
INVISIBLE 168
KEY / KEYS 171
Toolbar Buttons *171*
LEFT 175
NOENTRY 178
NOPROMPT 181
NOSCROLL 185
PASSWORD 186
PICTURE 188
Editing keys during data entry in character mode *189*
PROGRAM 191
Default Editors *192*
The Command String *192*
REQUIRED 194
REVERSE 196
RIGHT 198

SCROLL 201
UPSHIFT 203
VALIDATE LIKE 205
VERIFY 206
WORDWRAP 208
Data Display with WORDWRAP *209*
Text Client Data Entry and Editing with WORDWRAP *209*
Multi Line Editor with WORDWRAP (text mode) *209*
Displaying Program Variables with WORDWRAP *209*
The COMPRESS and UNCOMPRESS Options *210*
WORDWRAP Editing Keys *211*
Non-WORDWRAP Displays *212*

CREATING AND COMPILING A FORM 215

Compiling a Form in LyciaStudio *215*
Compiling a Form at the Command Line *215*
USING PERFORM FORMS IN LYCIA 217

About This Guide

Conventions used in this book

Typefaces and Icons

Constant-width text is used for code examples and fragments, or command line functions.

Constant-width bold is used either for replaceable items or emphasis.

Text or sample code which is shown in *italics*, is used to represent a value which will change according to programming needs, user input or some other factor.

	This icon indicates a suggestion, a general note, or a warning.

Significant code fragments are generally written in a monospace font like this:

```
database cms
main
            display “Hello World”
end main
```

Code examples shown in this document may be used within any Querix applications – no special permission is required.

CHAPTER 1

Form Drivers, Fields & Navigation

A *screen form* is a visual display that can support input and output tasks in the Lycia application. The screen form is defined in a *form specification file* (filename must have the file extension .per). This source file describes the logical format of the screen form and how to display data values in the form at runtime. It must be compiled using the form compiler and can be changed/re-compiled without the need for re-compiling the application logic (Separated Presentation Layer).

The same compiled form can be re-used by different 4GL programs, windows and functions.

This chapter describes form drivers, which control the display of a form, and form fields, including their behaviour and how to navigate among them.

The classical IBM-Informix 4GL and Dynamic 4GL (BDS) Forms are fully compatible with the Querix development tools. Any forms produced for these development tools can be fully re-used, maintained and further enhanced.

Due to the course of the 4GL modernization, Querix has introduced a large range of graphical extensions to its 4GL derived language Querix4GL. Many of these graphical Querix4GL widgets, which you can use in the LyciaStudio Graphical Form Editor have been made compatible with Dynamic 4GL form extensions and allow for additional/optional configuration properties. Using the graphical form editor in LyciaStudio, widgets can be added, positioned and configured in a graphical environment. Where needed, the properties can be viewed and modified in the corresponding properties view. It should be noted that extended form fields (widgets) are only suited for a graphical client environment. If a text mode client is used, these fields look and operate like any other standard text field.

The examples of forms in this guide are parts of the demonstration applications ‘forms’ and ‘guidemo’. You can download them for Lycia from the CVS repository. For information on how to import projects into LyciaStudio, please refer to the ‘Lycia Getting Started’ guide.

The required 4GL code for Forms – Form Driver

To work with a compiled screen form, the application requires a form driver, a logical set of 4GL statements that control the display of the form, bind form fields to 4GL variables, and respond to actions by the user in fields.

The form driver can include 4GL screen interaction and data manipulation statements to enter, retrieve, modify, or delete data in the database. The emphasis of this chapter, however, is on how to create the form specification file, rather than how to design and implement the form driver.

Display a Form

A form can be displayed directly or with a window association.

Display a Form independently

To display a form at any time, the form needs to be opened, displayed and removed (from memory) by using the three separate statements/functions.

Lycia supports both the classic Informix 4GL / BDS form handling statements and its own dynamic form and window management functions. You cannot mix them however, so it would not be possible to open a form using the dynamic method but display it with the static open form statement.

Open Form

```
OPEN FORM <static_form_identifier> FROM <form_file>
CALL fgl_form_open(<form name string>) FROM <form file>
```

Both have the same effect but the function fgl_form_open() has the advantage of using string variables to identify/manage forms. You cannot use a variable for the static form identifier (open form statement).

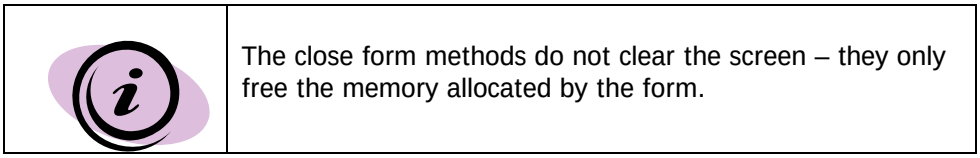
Display Form

```
DISPLAY FORM <static_form_identifier>
CALL fgl_form_display(<form name string>)
```

Close Form

```
CLOSE FORM <static_form_identifier>
CALL fgl_form_close(<form name string>)
```

For more information please refer to the ‘Built-in-functions’ guide.



Display a form associated with a window

When opening a window, a form file name can be associated with that window by specifying the additional information in the corresponding statements/functions.

```
OPEN WINDOW <window identifier> AT <row>,<column> WITH FORM <form file name>
CALL fgl_window_open(<window name>,<row n>,<col n>,<form file>,<border bool.>)
```

Using these statements/functions, the form is opened with the window and the window is sized correspondingly to the form size.

For more information please refer to the 'Built-in Functions' guide.

Form Position

The vertical (start) position of the form is defined within in the 4GL code. Default location is line 3 but this can be changed whenever required by using the statement:

```
OPTIONS
  FORM LINE <line number or LAST or FIRST>
```

The reason for the default line 3 is that the ring menu occupies line 1 for the menu and line 2 for the menu help text.

Displaying and retrieving data from Forms

Regardless of how you define them, there is no implicit relationship between the values of program variables, form fields, and database columns. Even if, for example, you declare a 4GL variable:

```
lname LIKE customer.lname
```

the changes that you make to the variable do not imply any change in the column value. Functional relationships among these entities must be specified in the logic of your form driver, typically through screen interaction statements of 4GL, and through data manipulation statements of SQL. After the user presses the Accept key (which is in the classic 4GL world Escape but can also be changed to another key, e.g. Enter) to terminate an INPUT ARRAY statement, for example, the form driver can use the INSERT statement to modify the database.

Similarly, a 4GL form is only a template. QFORM (the form compiler) reads the system catalogue at compile time to obtain the names and data types of any columns that are referenced in the form specification file. After compilation, however, the form loses its connection to the database. It can no longer distinguish the name of a table or view from the name of a screen record.

It is up to you, as a programmer, to determine what data a form displays and what to do with data values that the user enters into the fields of a form. You must indicate the binding explicitly in any 4GL statement that connects 4GL variables to screen forms or to database columns. The following statements, for example, take input from a 4GL form and insert the entered value from the form into the database. (Here the '@' sign in the INSERT statement tells 4GL that the first lname is the SQL identifier of a database column.)

```
INPUT lname FROM customer.lname
INSERT INTO customer (@lname) VALUES (lname)
```

You can use interactive 4GL statements such as OPEN FORM, OPEN WINDOW, INPUT, DISPLAY FORM, CLEAR FORM, and CONSTRUCT in the form driver to support data entry or data display through the 4GL form. Some statements support temporary binding when a program variable and a screen field have identical names. (See the individual 4GL statement descriptions in the chapter on 4GL Statements for the appropriate syntax.) For example, the following statement could replace the previous INPUT statement:

```
INPUT BY NAME lname
```

It is a good practice to have field variables identical to the database field names.

Accepting/Closing an Input

The user can close a field input at any time by pressing the 'Accept Key'. The Accept key is by default 'Escape', but can be re-defined using the OPTIONS statement to match, for example, the Return key.

```
OPTIONS
ACCEPT KEY RETURN
```

The above example would change the current/default accept key to the 'Enter' key press.

Example Program Location:

Project: forms
Program: fm_options_input_wrap

Form Fields

In a form, a field (sometimes called a screen field or form field) is an area where the user of the application can view, enter, and edit data, depending on its description in the form specification and the statements in the form driver. A field can also be a special graphical widget such as a calendar, browser, bmp or button widget. This section discusses the appearance and behaviour of form fields in Lycia.

Appearance of Fields

Text Mode

In standard 4GL the screen form contains *display fields* bounded by *delimiters* such as square brackets or Pipes.

Colour

The foreground colour of the field text contents can be specified within the form definition or as an ATTRIBUTE during an input or display statement.

The colour of Static text labels (e.g. field labels) which are defined within the form can only be defined on a window basis by using the optional ATTRIBUTE specification during an OPEN WINDOW statement. To overcome this limitation, dynamic labels can be used which are controlled like normal fields.

Unlike the GUI clients, 'Text Mode' clients support only foreground colour (if you ignore the REVERSE field attribute) and there is no difference in colour appearance if the object is active (enabled) or disabled.

GUI Clients

The display of fields (text boxes) in GUI clients depends on the chosen client system, client operating system and some optional configuration settings. Fields can be displayed with or without a field border.

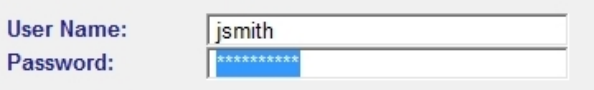
Colour

The field contents' foreground and background colours are dependent on whether the field is active/enabled (in input state), or inactive (disabled display state).

The developer can configure colour style sets which include information for foreground, background, active and inactive states.

Fonts

GUI clients also support any font of the local client operating system including proportional fonts. Using proportional fonts, the user interface not only improves its aesthetic appeal; it also increases the amount of the useable space. Proportional fonts allow usually for about 50% more information compared to non-proportional fonts used with the text mode clients.



Whichever format the form is displayed in, the currently active form field will contain a visible cursor. The *current field* is where keystrokes that the user types are displayed.

Field Context Menu / Clipboard

GUI clients also support the right mouse button context menu. The Clipboard functionality to exchange information between fields and other applications is available by default. The classic Ctrl-C/X/V keys are also operational. The programmer can also add additional entries in the form definition and in the 4GL code to raise key and action events.

Size and Position and identifier tags of Fields

Just like the classic 4GL, the unit for object size and position are rows and columns. The size of a field is measured by the quantity of columns between the corresponding field delimiter within the SCREEN section of the form definition. Each field must have a unique screen field identifier tag.

Example:

The field with the field tag ‘f1’ has a physical width of ‘20 columns’ and its positioned on the form is ‘row 2’, ‘column 1’.

```
SCREEN
{
 12345678902234567890
[f1                  ]
}
```

For details about how to size and position fields in a form, see “Display Fields”.

Field identifiers (variable names)

A screen field tag must be associated with a field identifier (variable name). The 4GL code has no understanding of field tags and will always access fields by using their field identifiers. Field identifiers always consist of the table name and the field name. If the field has no direct relation with a database table, the table name reference ‘formonly’ will be used. Each field tag – field identifier association is terminated by a semicolon ‘;’.

Example:

We associate the field tag f1 with the field identifier customer.cust_id.

```
f1 = customer.cust_id;
```

Field Attributes

The developer has access to a large pool of form field attributes to manipulate the presentation and behaviour of the field contents. These field attributes can be assigned to fields independently and are comma separated.

Example:

We assign the field attributes ‘colour=red’ and convert the field contents to upper case.

```
f1=   customer.cust_id,
      color=red,
      upshift;
```

For information on assigning display and validation attributes to fields, see “Field Attribute Syntax”.

Field Navigation

Navigation within a field, Field Width and Input Field Width

Text mode

Using text mode clients, the user cannot see more characters than the actual visual field width (in columns) even if the underlying value could hold longer strings. The user can scroll within the text field by using the left/right cursor keys. There are compile and field attributes which control whether the field input length is limited to the physical field width (columns) or if the cursor can scroll in the range-length of the underlying field variable.

GUI Clients

The field width is by default controlled by the length of the underlying variable but can be limited to the field width (in columns) by using the optional field attribute ‘noScroll’.

More detailed information on this topic is available in the chapter ‘Field Attributes’, section ‘noPrompt’ and ‘Scroll’.

Navigation among Form Fields

The order in which the cursor moves from field to field on a screen form is determined by the order in which you list fields in the INPUT statement.

```
INPUT a,b WITHOUT DEFAULTS FROM f0,f1
```

At any time before pressing the ACCEPT key or the key used to navigate to the next field in the last field, the user can use the following keys to navigate forward and backward through the fields:

Action	Text Client	GUI Client
Next Field	Tab Shift Tab Cursor/Arrow DOWN ENTER key – if the Enter key is not defined as the Accept key Arrow key RIGHT (when the cursor has reached the last character position in a field.	Tab Cursor/Arrow DOWN ENTER key – if the Enter key is not defined as the Accept key
Previous Field	Cursor/Arrow UP Arrow Key LEFT (when the cursor is on the first character position in a field.	Cursor/Arrow UP Shift Tab

Overwriting the default field order using NEXT FIELD

In CONSTRUCT, INPUT, and INPUT ARRAY statements, the NEXT FIELD clause can override the default order in which fields are accessed.

AUTONEXT Attribute in the Form Attributes Section

The AUTONEXT field attribute can move the cursor automatically to the next field when the user has typed enough characters to fill a field/variable (depending on the client). The user can indicate that data entry is complete by pressing the Accept key in any field.

```
ATTRIBUTES

f0 = formonly.f0, autonext;
f1 = formonly.f1, password;
```

AutoNext - difference between text and GUI clients

Using text clients, the client navigates to the next field when the cursor has reached the last character of the visual field. In a GUI client, the AutoNext field navigation takes place when the cursor has reached the last character/element of the underlying variable.

Detailed information is available in the chapter field attributes, topic 'AutoNext'.

Circular field navigation using INPUT WRAP

Data can also be entered by pressing a 'next field' navigation key when the cursor is located in the last field, but only if INPUT WRAP was not specified in the OPTIONS statement.

If INPUT WRAP was specified, navigating to the next field when the cursor is already located in the last field moves the cursor back to the first field (circular field input navigation).

In order to restore the default field order, specify OPTIONS CURSOR NO WRAP.

```
OPTIONS
INPUT NO WRAP
INPUT WRAP
```

Example Program Location:

Project: forms
Program: fm_options_input_wrap

Field Navigation by visual field location or input variables field order

The FIELD ORDER setting in the OPTIONS statement determines where the arrow keys move the cursor. If FIELD ORDER CONSTRAINED is specified, pressing the Up Arrow key moves the cursor to the previous field, and pressing the Down Arrow key moves the cursor to the next field. If FIELD ORDER UNCONSTRAINED is specified, pressing the Up Arrow key moves the cursor to the field above the current cursor position and pressing the Down Arrow key moves the cursor to the field below the current cursor position.

```
OPTIONS
FIELD ORDER CONSTRAINED
FIELD ORDER UNCONSTRAINED
```

Demo application:

Project: forms
Program: fm_options_field_order

Disabled & NOENTRY Form Fields

When a screen interaction statement like INPUT or CONSTRUCT doesn't include a form field, or the form field is specified as a NOENTRY field in a form file, it is disabled during execution of that statement. The user cannot then move the cursor to that field. If the user attempts to enter the NOENTRY field by using the TAB or arrow keys, the cursor moves to the next field in the default order without processing any BEFORE and AFTER clauses of the corresponding disabled fields.

```
ATTRIBUTES

f0 = formonly.f0, noentry;
```

Demo application:

Project: forms
Program: fm_misc_ex_attrib_noentry

CHAPTER 2

Structure of a Form Specification File

A 4GL form specification file is a source file (with file extension **.per**) that you can create with a text editor or within LyciaStudio. This file comprises three required sections (DATABASE, SCREEN, and ATTRIBUTES), and may also include four optional sections (TABLES, INSTRUCTIONS, KEYS and ACTIONS). If present, these seven sections must appear in the following order:

DATABASE section. Each form specification file must begin with a DATABASE section identifying the database (if any) on which the form is based. This can be any database that the current database server can access, including a remote database.

SCREEN section. The SCREEN section must appear next, showing the dimensions and the exact layout of the logical elements of the form. You must specify the position of one or more screen fields for data entry or display as well as any additional text or ornamental characters.

TABLES section. If it is present, the TABLES section must follow the SCREEN section. This section lists every table or view that is referenced in the ATTRIBUTES section. If a table requires as a qualifier, the name of an owner or of a database, the TABLES section must also declare an alias for the table.

ATTRIBUTES section. The ATTRIBUTES section describes each field on the form and assigns names to fields. Field descriptions can optionally include field attributes to specify, for example, the appearance, acceptable input values, on-screen comments, and default values for each field.

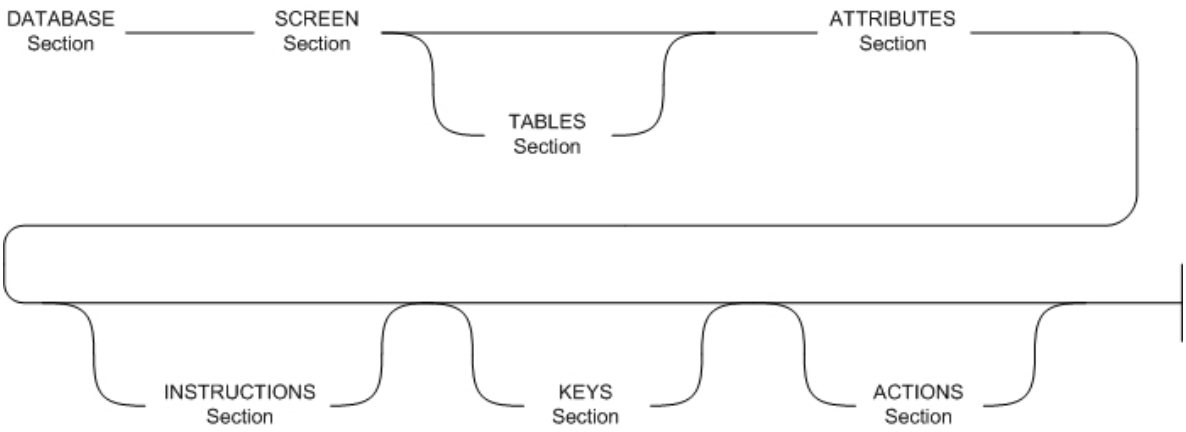
INSTRUCTIONS section. The INSTRUCTIONS section is optional. It can specify screen arrays and non-default screen records and field delimiters.

KEYS section. The KEYS section is optional. It can specify toolbar and context menu items based on key events.

ACTIONS section. The ACTIONS section is optional. It can specify toolbar and context menu items based on action events.

Each section must begin with the keyword which is its name. After you create a form specification file, you must compile it. The form driver of your 4GL application can then use 4GL variables to transfer information between the database and the fields of the screen form.

This is the syntax of a ' form specification:



The next seven sections of this chapter identify the keywords and terms that are listed in this diagram and describe their syntax in detail.

The following example illustrates the overall structure of a typical form specification:

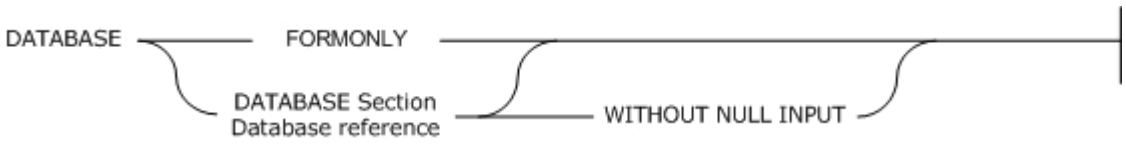
```
DATABASE cms

SCREEN {

   Querix Demo CMS V.0.4 - Login

   \gp-------------------------q   \g
   \g|                         |   \g
   \g|  \gUser\g \gID:  [f1        ] \g|   \g
   \g|  \gPassword: [f2        ] \g|   \g
   \g|                         |   \g
   \gb-------------------------d   \g
      User\g \gID            Admin
      Password           Admin

            [f11     ][f12     ]
}

TABLES
  users

ATTRIBUTES
  f1=users.name, comments="Enter your User ID", upshift;
  f2=users.password, comments="Enter your Password", password, upshift;
  f11=formonly.f11, config="Return OK", widget="button";
  f12=formonly.f12, config="Break Cancel", widget="button";
```

```
INSTRUCTIONS
  DELIMITERS "[]"

KEYS
  F2="OK" ORDER 1
  F3="Cancel" ORDER 2
  F12="Exit" ("icon16/exit01.ico") ORDER 99
  HELP="Help" ("icon16/help01.ico") ORDER 101

ACTIONS
  act_edit="Edit" ("icon16/record_edit01.ico") ORDER 5
```

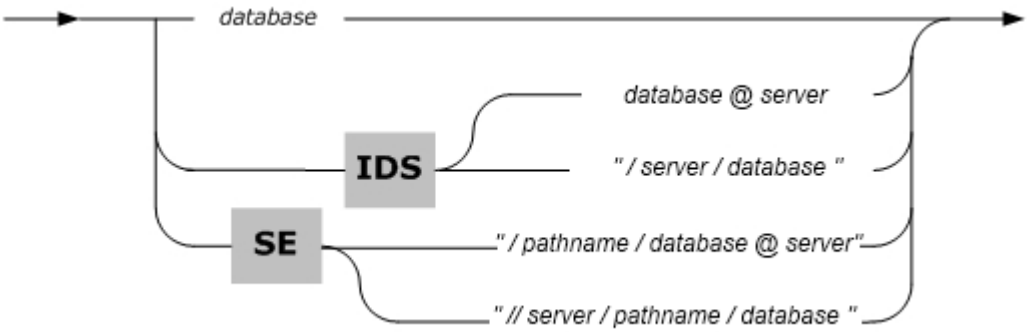
In this example, the screen form has been designed to display columns from a table in the **cms** demonstration database and includes all five of the required and optional sections that are described in the pages that follow.

DATABASE Section

The DATABASE section identifies the database, if any, that contains tables or views whose columns are referenced in the form specification file.



The following diagram shows the syntax of the DATABASE Section Database Reference.



Element	Description
database	This is the SQL identifier of a database.
pathname	This is the path of the parent directory of the **.dbs** directory (for Informix-SE databases only).
server	This is the name of the host system on which the database is held.

Usage

The DATABASE section is required, even if the screen form does not reference any database columns or tables. You can specify only one database.

When compiling forms, Lycia uses the schema of tables from the specified database to define the data types of fields in the form and obtains default values and attributes from the syscolval and syscolatt tables in the default database.

The FORMONLY Option

You can create a form that is not related to any database. To do so, specify FORMONLY after the DATABASE keyword, and omit the TABLES section. Also specify FORMONLY as the only table name in the ATTRIBUTES section when you declare the name of each field.

The following example of a DATABASE section specifies that the screen form is not associated with any database:

```
DATABASE FORMONLY
```

Compilation errors can result if FORMONLY appears in the DATABASE section of a form that also specifies features that depend on information from the system catalogue or from the **syscolval** and **syscolatt** tables of a database.

You can declare fields as FORMONLY in the ATTRIBUTES section, however, even if the DATABASE section specifies a database.

Features of 4GL forms which depend on a database:

- The TABLES section
- Any field associated with a database column in the ATTRIBUTES section
- Any FORMONLY field declared LIKE a column in the ATTRIBUTES section
- DISPLAY LIKE or VALIDATE LIKE attributes in the ATTRIBUTES section

The WITHOUT NULL INPUT Option

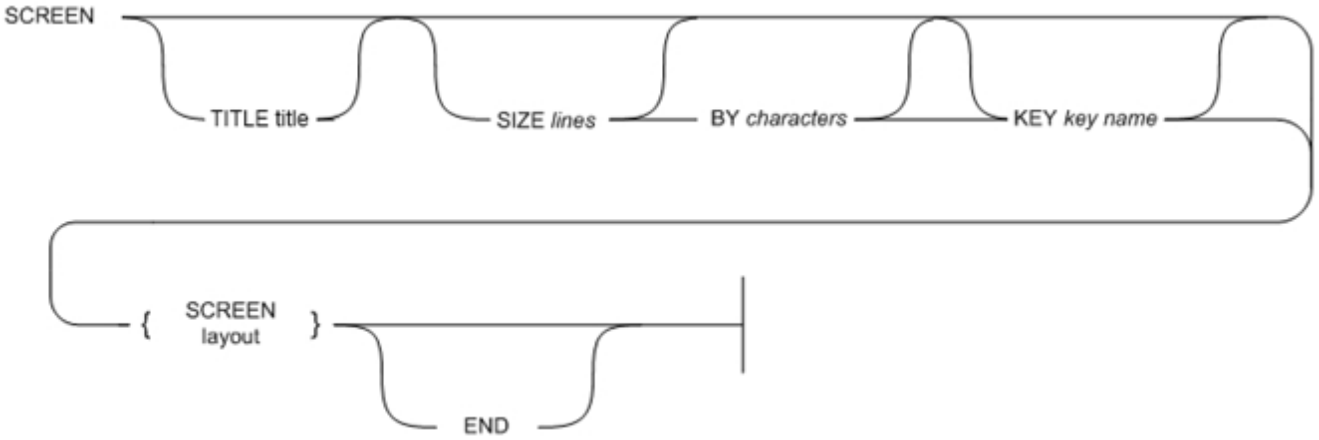
The WITHOUT NULL INPUT keywords indicate that the database name does not support null values. Use this option only if you have elected to create and work with a database that does not support null values.

For fields that have no other defaults, the WITHOUT NULL INPUT option causes the form to display zeros as default values for number and INTERVAL fields and blanks for character fields. DATE values default to 12/31/1899. The default value for DATETIME fields is 1899-12-31 23:59:59.99999.

The use of WITHOUT NULL INPUT is discouraged. This specification was useful in 1981 for Informix 1.00 databases, which did not yet support the construct of null values. Current Informix databases can prevent null input by specifying a not-null constraint on individual columns through the CREATE TABLE or ALTER TABLE statements. If your application requires non-null values in a field corresponding to a database column, you can specify in the ATTRIBUTES section that the field is FORMONLY and NOT NULL, as described in “Field Attribute Syntax”.

SCREEN Section

The SCREEN section of the form specification file specifies the vertical and horizontal dimensions of the physical screen and the position and dimensions of the visual objects that will appear on the screen form.



Size properties

The additional SIZE definitions are optional and are only supported for legacy compatibility reasons. If a screen section is too large to be displayed on the screen, the application will divide the single screen sections into multiple screens during an input. The SIZE keyword is used to specify the maximum size of the screen section.

	The additional SIZE definitions are not recommended for use. They are only supported for legacy reasons; use multiple screens (Tabs) instead. For multiple screens, please refer to the section 'Tabs – Multiple Screens'

Element	Description
Lines	This is a literal integer that specifies how many lines of characters the form can display. The default is 24.
characters	This is a literal integer that specifies how many characters a line can display. The default is the number of characters specified in the longest line of the screen layout.
key name	This is the name of the key press associated with the screen. Pressing the key will bring that screen into focus.
screen name	This is the name of the screen.

Usage

The SCREEN keyword is required. As in other sections of a form specification, the keyword END is optional.

A single pair of braces ({ }), immediately preceded and immediately followed by NEWLINE characters, must enclose the screen layout. You cannot use comment indicators in the SCREEN section. Any contents within the screen section will be displayed.

The SIZE Option

If you omit the SIZE keyword, *lines* defaults to the actual number of lines used + 4.

Specify lines as the total height of the form. Four lines are reserved for the system, so by default, no more than (lines - 4) lines of the form can display data. (But the OPEN WINDOW...ATTRIBUTE (FORM LINE FIRST, COMMENT LINE OFF) statement can reduce this overhead.)

The portion of the SCREEN section between the braces is called the screen layout. This portion shows the geometric arrangement of the logical screen. If the SIZE clause or command line specifies dimensions that are too small for the screen layout, QFORM issues a compile-time warning, but it produces the compiled form that your form specification file described.

Demo Application

Project: forms
Program: fm_screen_size

Tabs - Multiple Screen Sections

A form can have one or more screen sections. Each section must have its own SCREEN section, they all must be located in one form file, they all are references by one ATTRIBUTES section, thus the field tags should be unique. If you add the TITLE keyword followed by a quoted character string which will be used as the tab title, your tabs will have names. Multiple screen sections require a unique screen (Tab) name and optional key event.

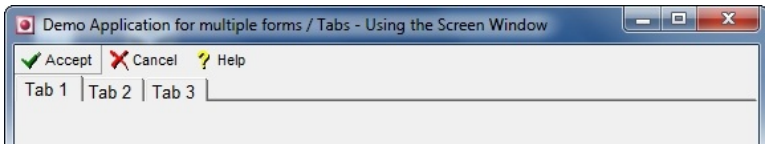
To be able to edit fields located on different screens your INPUT statement must span all of the fields involved.

Text Mode Client:

There is no visual aid to see the existence of multiple screens or to navigate between them. The client navigates between the different screen sections only during an input. Depending on the field, where the cursor is located, the corresponding screen section will be displayed. You can for example control the screen display by using the NEXT FIELD <field name> statement within your 4GL code.

GUI Mode Client:

GUI clients will display the existence of multiple screens by rendering a ‘Tab Selection Bar’.



All screen names are used for the ‘Tab-Label’. When the user clicks on them, the corresponding key event will be launched. Detailed information will follow in the section ‘Tabs-Navigation’.

Example:

In the next example, we see the form code for a multiple screen (tabs) form with three screen definitions (tabs).

The first screen is named ‘Tab 1’ and it launches the key press ‘F13’ when the user clicks the corresponding tab by using the mouse.

The first screen is named ‘Tab 2’ and it launches the key press ‘F14’ when the user clicks the corresponding tab by using the mouse.

The first screen is named ‘Tab 3’ and it launches the key press ‘F15’ when the user clicks the corresponding tab by using the mouse.

Example:

```
SCREEN KEY F13 TITLE "Tab 1"
{

=======top screen 1============

 Hello\g \gTab\g \g1

 [f0        ] [f1        ]

=======bottom screen 1=========

}

SCREEN KEY F14 TITLE "Tab 2"
{

=======top screen 2============
```

```
 Hello\g \gTab\g \g2

 [f2        ] [f3        ]

=======bottom screen 2=========

}

SCREEN KEY F15 TITLE "Tab 3"
{

=======top screen 3============

 Hello\g \gTab\g \g3

 [f4        ] [f5        ]

=======bottom screen 3=========

}
```

Switching Between Tabs (Navigation)

There is the option to have the focus switch to the first field of the chosen tab. This can be done in two ways:

Tabs Navigation using ON KEY() and Next FIELD

Firstly, it is possible to send a keypress when clicking on the required tab header that will invoke a function in the 4GL that shifts focus to the first field in that tab; that tab coming into view as a result of it field being in focus. The following example shows the new way of invoking a function call in the tabs definition:

```
SCREEN KEY F12 TITLE “My Tab”{
...
}
```

Then in the program, you can make a call to F1 that will set a field in the desired tab as the current field, which will bring that tab into view:

```
ON KEY(F12)
  ...
  NEXT FIELD field1
```

Tab Navigation control using fglprofile

There is also an fglprofile setting that will attempt to change the focus to the first field in a selected tab. Depending on the program logic, this may not work, for example if the program is still waiting for the previous field to send information. To utilise this functionality, add the following to your fglprofile:

```
gui.foldertab.input.sendnextfield
```

There is also the following fglprofile setting that will perform an action specified by a key when a tab is selected:

```
gui.key.foldertab.<n>.selection = key
```

Where n is the key name.

Example Application for multiple screens

Demo Application

Project: forms
Program: fm_screen_tabs1, fm_screen_tabs2, fm_screen_tabs3, fm_screen_tabs4

4GL Code:

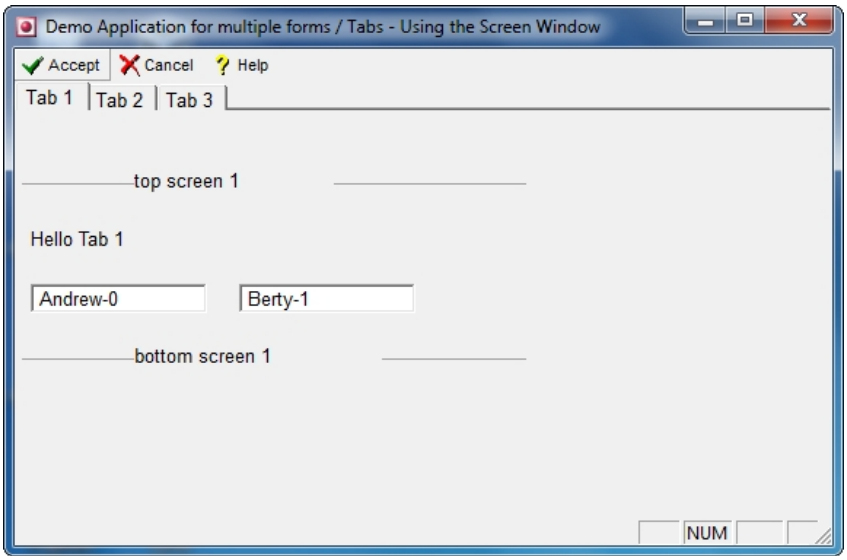
```
##############################################################################
# input_data()  - Input data
##############################################################################
FUNCTION input_data()

  # When all fields are completed in that particular tabs-form, it will
  # switch to the next tab form with the next field in the input statement
  INPUT BY NAME f0,f1,f2,f3,f4,f5 WITHOUT DEFAULTS HELP 100
    ON KEY(F13)
      NEXT FIELD f0
    ON KEY (F14)
      NEXT FIELD f2
    ON KEY (F15)
      NEXT FIELD f4

  END INPUT

END FUNCTION
```

Form (.per) Code:

```
DATABASE formonly

SCREEN KEY F13 TITLE "Tab 1"
{

=======top screen 1============

 Hello\g \gTab\g \g1
```

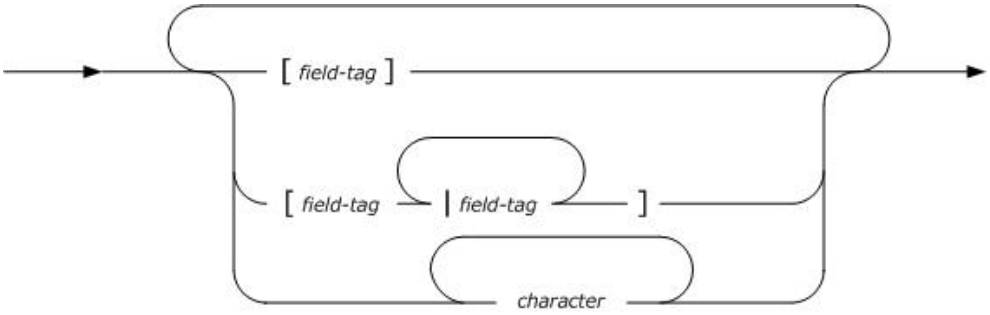
```
 [f0        ] [f1        ]

=======bottom screen 1=========

}

SCREEN KEY F14 TITLE "Tab 2"
{

=======top screen 2============

 Hello\g \gTab\g \g2

 [f2        ] [f3        ]

=======bottom screen 2=========

}

SCREEN KEY F15 TITLE "Tab 3"
{

=======top screen 3============

 Hello\g \gTab\g \g3

 [f4        ] [f5        ]

=======bottom screen 3=========

}

ATTRIBUTES
f0=formonly.f0;
f1=formonly.f1;
f2=formonly.f2;
f3=formonly.f3;
f4=formonly.f4;
f5=formonly.f5;

INSTRUCTIONS
DELIMITERS "[]"
```

Sample Screen:



The Screen Layout

The screen layout of the SCREEN section must be enclosed between a pair of braces, each in the first character position of an otherwise empty line. The screen layout consists of display fields and (optionally) text characters.



Element	Description
field-tag	This is a 4GL identifier of no more than 50 characters within each field. The length of the field tag cannot be greater than the field width.
character	This is a printable character (static label) of text or a graphical line that appears on the form. You cannot format or dynamically change static form labels directly from 4GL. GUI clients support a scripting language to modify the contents and appearance of static form labels. We strongly recommend the use of dynamic labels (not static labels).

If you want to separate fields by using a pipe symbol [aaa|bbb], it is required that you specify a pair of pipe symbols as your field delimiter.

Display Fields

Every 4GL form which is used to input or display field data must include at least one normal field or field widget where data can appear. Use brackets ([ ]) to delimit fields in the screen layout. Between delimiters, each field must have an identifying field tag.

```
SCREEN
{
[f1    ]  [f2    ]
}
```

See “Field Tags” for more details.

Field Delimiters

Each field must be indicated by left and right delimiters to show the length of the field and its position within the screen layout. Both delimiters must appear on the same line. Usually you use left and right brackets to delimit fields. However, to make two fields appear directly next to each other, you can use the pipe symbol (|) to indicate the end of the first field and the beginning of the second field. This however also requires the ‘|’ pipe delimiter definition in the INSTRUCTIONS section of the form

```
SCREEN
{
[f1    ]  [f2    ]
[f3  |f4    |f5 ]
}

INSTRUCTIONS
DELIMITERS "||"
```

Demo application:

Project: forms
Program: fm_instructions_delimiter

For complete information on using a pipe symbol to delimit fields, see “Field Delimiters” in the INSTRUCTIONS section.

Field Length

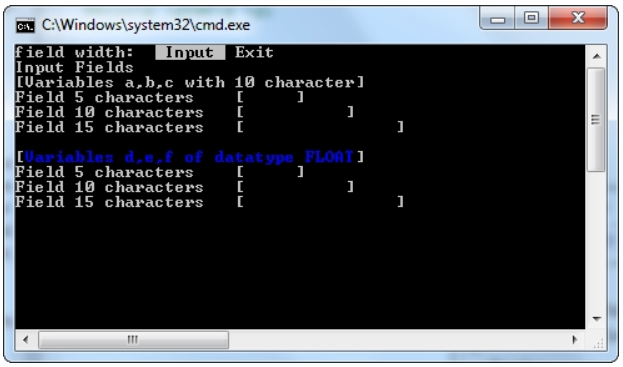
Text mode

If the form will be compiled with the Informix compatibility option for field length (ls), you normally should set the width of each display field in the SCREEN section to be equal to the width of the program variable or the database column to which it corresponds.

Using text mode clients, the user cannot see more characters than the actual visual field width (in columns) even if the underlying value could hold longer strings and in default Informix mode, you can also not enter more details. There are compile options and field attributes which control whether the field

input length is limited to the physical field width (columns) or whether the cursor can scroll in the range-length of the underlying field variable.

A field to display numeric values should always be large enough to contain the largest anticipated value. When a numeric field is too small to display a data value, Lycia fills the field with asterisk (*) symbols to indicate the overflow.



GUI Clients

The field width is by default controlled by the length of the underlying variable but can be limited to the field width (columns) by using the optional field attribute ‘noScroll’.

A field to display numeric values should always be large enough to contain the largest anticipated value.

Fields to display character data only can be shorter than the data length. Lycia fills the field from the left and truncates from the right any string that is longer than its display field. By using multiple-segment fields, you can display portions of a long character value in successive lines of the form.

For more information, see “Multiple-Segment Fields / Wordwrap field attribute” later in this chapter.

Field Tags

Field tags must follow the rules for 4GL identifiers (as described in “4GL Identifiers”. The first character of a field tag must be a letter or an underscore. Other characters can be any combination of letters, digits, and underscores. Because QFORM is not case sensitive, both **a1** and **Al** represent the same field tag. Field tags cannot be referenced in 4GL statements. The ATTRIBUTES section declares a field name for each field tag.

Each field has only one tag, but fields with the same tag can appear at more than one position in the SCREEN section in three special cases:

- As part of a screen array
- As part of a multiple-segment WORDWRAP field
- As part of a browser widget field

Otherwise, each field tag must be unique within a form.

Because a field tag must fit within the brackets that delimit its field, you can give single-character fields the tags **a** through **z**. This designation implies that a form can include no more than 27 single-character fields in the default locale.

Example:

```
SCREEN
{
                          [dl_header                          ]
p-------------------------------------------------------------------------q
| [dl_c_id    |f_cust_id    ]                [a|dl_cb_rec_email  ]  [bmp_image]|
| [dl_title   |f_cust_title]                 [b|dl_cb_rec_call   ]               |
| [dl_f_name  |f_cust_fname           ]      [c|dl_cb_rec_post   ]               |
| [dl_s_name  |f_cust_lname           ]                                          |
| [dl_countr  |ff_cust_country       ]      [d|dl_rb_ct_private ]               |
| [dl_email   |ff_cust_email         ]        [dl_rb_ct_business]               |
| [dl_dob     |ca_cust_dob ]                                                    |
|                                                                               |
| [f_memo                                                 ]                     |
| [f_memo                                                 ]  [hl_url          ]|
| [f_memo                                                 ]  [hl_cust_email   ]|
| [f_memo                                                 ]                     |
|                                                                               |
| [f_act_da][f_act_activity    ][f_act_tit][f_act_conf][f_act_conl][f_act_st]   |
| [f_act_da][f_act_activity    ][f_act_tit][f_act_conf][f_act_conl][f_act_st]   |
| [f_act_da][f_act_activity    ][f_act_tit][f_act_conf][f_act_conl][f_act_st]   |
| [f_act_da][f_act_activity    ][f_act_tit][f_act_conf][f_act_conl][f_act_st]   |
| [f_act_da][f_act_activity    ][f_act_tit][f_act_conf][f_act_conl][f_act_st]   |
b-------------------------------------------------------------------------------d
[fb_input   |fb_done    ]     [fb_error   |fb_message]     [fb_cancel |fb_help    ]
}
```

```
ATTRIBUTES

f_memo = formonly.cust_memo type CHAR,
         comments = "Memo Notes",
         wordwrap compress;

INSTRUCTIONS

SCREEN GRID sc_rec [5] (
       formonly.act_date,
       formonly.act_activity,
       formonly.act_tit,
       formonly.act_conf,
       formonly.act_conl,
       formonly.act_state
       )
```

Static Labels and lines in Forms

A screen layout can specify character strings (and special line characters) that always appear in the form also known as static labels. These strings can label the form and its fields or format the display.

It is recommended not to use static labels if the form should be used in a GUI environment.

See also “Graphical Characters in Forms”.

Text and fields cannot overlap

Because the 4GL form file is a single layer screen definition, it is not possible to overlay multiple fields or static labels on the same location in the screen section.

The “INSTRUCTIONS Section” describes how repeated field tags are used in forms that define screen arrays.

The backslash (\ ) is not valid as a text character; QFORM attempts to interpret it as the beginning of an escape sequence and does not print it.

In addition, your form might not compile correctly if you attempt to use either braces ({ and }) or the field delimiter ([,], and |) symbols as text characters in the screen layout.

QFORM interprets any hash symbols (#) or double-hyphens (--) in the screen layout as literals, not as comment indicators.

Graphical Characters in Forms

You can include graphical characters in the SCREEN section to place boxes and other rectangular shapes in a screen form. Use the following characters to indicate the borders of one or more boxes on the form.

Symbol	Purpose
p	To mark the upper, left corner of a box.
q	To mark the upper, right corner of a box.
b	To mark the lower, left corner of a box.
d	To mark the lower, right corner of a box.
-	To show segments of horizontal line.
|	To show segments of vertical lines.
+	To show an intersection between a horizontal and vertical line.

The meanings of these seven special characters are derived from the **acsc** specifications in the **terminfo** files. Lycia substitutes the corresponding graphics characters when you display the compiled form.

Once the form has the desired configuration, use **\g** to indicate when to begin graphics mode and when to end graphics mode.

Insert **\g** before the first **p**, **q**, **d**, **b**, hyphen, or pipe symbol that represents a graphical character. To leave graphical mode, insert **\g** after the **p**, **q**, **d**, **b**, hyphen, or pipe symbol.

Do not insert **\g** into the original white space of a screen layout. The backslash should displace the first graphics character in the line and push the remaining characters to the right. The process of indicating graphics distorts the appearance of a screen layout in the SCREEN section, compared to the corresponding display of the screen form.

You can include other graphical characters in a form specification file, but the meaning of a character other than the **p**, **q**, **d**, **b**, hyphen, and pipe symbol is terminal dependent.

To use graphical characters, the system terminfo file must include entries for specific variables.

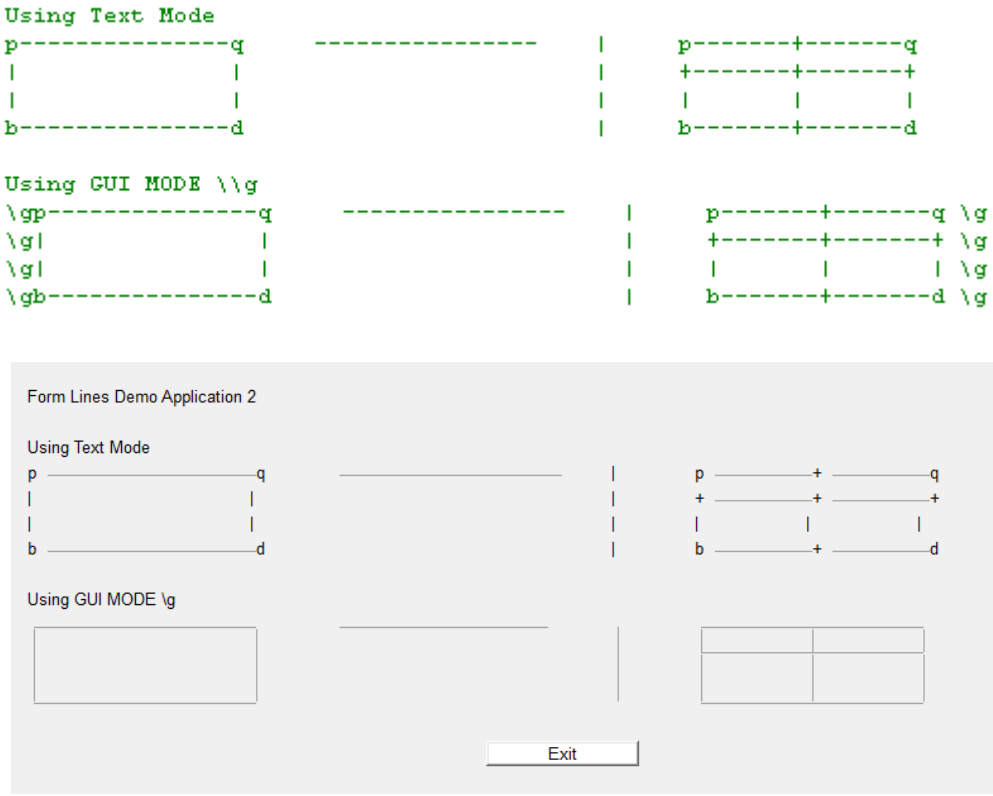
The following table shows what variables need to be set in the **terminfo** file.

terminfo Variable	Description
Smacs	The escape sequence to enter graphical mode.
Rmacs	The escape sequence to leave graphical mode.
Acsc	The concatenated, ordered list of ASCII equivalents for the graphical characters needed to draw the borders of a box.

Rectangles within Forms

You can use the built-in functions fgl_drawline() and fgl_drawbox() to enclose parts of the screen layout within rectangles. Rectangles that you draw with FGL_DRAWBOX( ) are NOT part of the displayed form. Each time that you execute the corresponding DISPLAY FORM or OPEN WINDOW...WITH FORM statement, you must also redraw the rectangle.

Example:



Demo Application

Project: forms
Program: fm_screen_lines1 and fm_screen_lines2

	Avoid any intersection between the rectangle and any field or 4GL reserved line as the rectangle will be broken at the intersection when anything is displayed in the field or in the reserved line.

TABLES Section

The TABLES section lists the database tables that are referenced elsewhere in the form specification file. You must list in this section any table, view, or synonym that includes a column whose name is referenced in the form.

Element	Description
alias	The alias that replaces *table* in the form specification file.
table	The identifier or synonym of a table or view in its database.

Usage

If the DATABASE section specifies FORMONLY, no TABLES section is needed unless you give a field the VALIDATE LIKE or DISPLAY LIKE attribute in the ATTRIBUTES section, or specify a FORMONLY field LIKE a database column.

Every database column referenced in the ATTRIBUTES section must be part of some *table* specified in the TABLES section. The *table* identifier is the name listed in the **tabname** column of the **systables** catalogue or else a synonym.

4GL allows you to specify up to 40 tables, but the actual limit on the number of tables, views, and synonyms that you can reference in a form depends on how your system is configured. The form specification file **orderform.per** in the demonstration application lists four tables:

```
TABLES customer orders items stock
```

The *table* identifier cannot be a temporary table. If the form supports entry or update of data in a view, your 4GL application should test at runtime whether the view is updatable, especially if it is based on other views.

The END keyword is optional.

Table Aliases

The TABLES section must declare an *alias* value for the identifier of any table, view, or synonym that requires a table qualifier. Table qualifiers can specify the owner of a table, or a database (or *database@server* value) that is different from the database in the DATABASE section. In an ANSI-compliant database, for example, you must qualify any table name with the *owner* prefix if the form will

be run by users other than *owner*. You do not need to specify *alias*, unless the form will be used in an ANSI-compliant database by a user who did not create *table*, or if the form references a table, view, or synonym whose name is the same as another in the same database, so that the *owner* prefix is required for an unambiguous reference.

The alias can be the same identifier as *table*. For example, **contact** can be the alias for **cms@ethel:tom.contact**, except to assign an alias in the TABLES section; a form specification file cannot qualify the name of a table. If a qualifier is needed, you must use an alias from the TABLES section to reference the table in other sections of the form specification file.

The same alias must also appear in screen interaction statements of 4GL that reference screen fields linked to columns of a table that has an alias. Statements in 4GL programs or in other sections of the form specification file can reference screen fields as *column*, as *alias.column*, or as *table.column*, but they cannot specify *owner.table.column*. You cannot specify *table.column* as a field name if you define a different alias for *table*.

The following TABLES section specifies aliases for two tables:

```
TABLES
tab1 = tom.activity
tab2 = john.contact
```

ATTRIBUTES Section

The ATTRIBUTES section is used to specify a field description that creates an association between an identifier and a data type for every field (field tag) in the SCREEN section of your form. You can also control the behaviour and appearance of each field by using field attributes to describe how 4GL should display the field, supply a default value, limit the values that can be entered, and set other parameters.

Field attributes are described in “Field Attribute Syntax”.

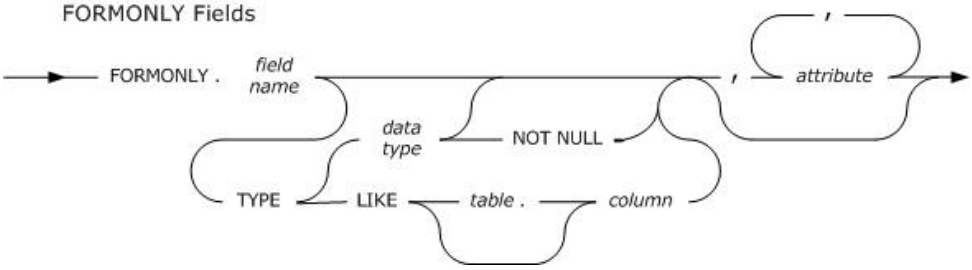
The ATTRIBUTES section has the following syntax.

The Field Description element has two possible forms, dependent on whether the fields are linked to a database column, or are FORMONLY fields.

ATTRIBUTES Section Field Descriptions

Fields Linked to a Database Column

FORMONLY Fields



Element	Description
attribute	This is a string of keywords, identifiers, and symbols that specify a field attribute, among those listed in “Field Attributes”. This can also refer to a Widget.
column	This is the unqualified SQL identifier of a database column.
data type	This is any data type specification (as described in “Data Types of 4GL”) except ARRAY or RECORD.
field name	This is an identifier that you assign to a FORMONLY field (a field that is not associated here with any database column).
field-tag	This is the field tag, as declared in the SCREEN section.
table	This is the name or alias of a table, synonym, or view, as declared in the TABLES section. This variable is not required unless several columns in different tables have the same name or the table is an external table.

Usage

The ATTRIBUTES section must describe every *field-tag* value from the SCREEN section. The order in which you list the field tags determines the order of fields in the default screen records that 4GL creates for each table. (The “INSTRUCTIONS Section” describes screen records.)

The END keyword is optional. It is supported to provide compatibility with form specification files for earlier versions of Informix products.

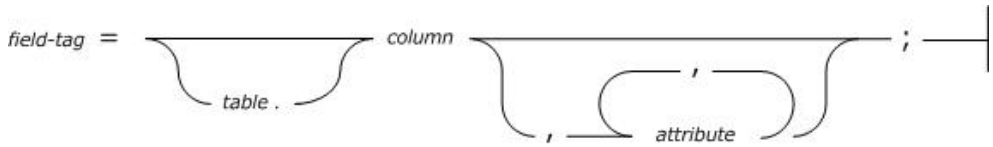
You can specify two kinds of field descriptions: those that associate a field tag with the data type and with the default attributes of a database column and those that link field tags to FORMONLY fields.

Fields Linked to Database Columns

Unless a display field is FORMONLY, its field description must specify the SQL identifier of some database column as the name of the display field. Fields are associated with database columns only during the compilation of the form specification file. During the compilation process, QFORM examines two optional tables, **syscolval** and **syscolatt**, for default values of the attributes that you have associated with any columns of the database. If these tables exist, then they can be examined; if they do not, then this examination will not happen.

(For a description of these tables, see “Default Attributes”.)

After QFORM extracts any default attributes and identifies data types from the system catalogue, the association between fields and database columns is broken, and the form cannot distinguish the name of a table or view from the name of a screen record. The form driver in your 4GL program must mediate between screen fields and database columns with program variables.



Element	Description
attribute	This is a string of keywords, identifiers, and symbols that specify a field attribute, among those listed in “Field Attributes”.
column	This is the unqualified SQL identifier of a database column, which can also be used in 4GL statements that reference the field.
field-tag	This is the field tag, as declared in the SCREEN section.
table	This is the name or alias of a table, or view, as declared in the TABLES section. This element is not allowed to have qualifiers.

Usage

Although you must include an ATTRIBUTES section that assigns at least one name to every *field-tag* value from the SCREEN section, you are not required to specify any field attributes.

You are not required to specify *table* unless the name *column* is not unique within the form specification, or if table is external to the database that the DATABASE section specifies. However, it is recommended that you always specify *table.column* rather than the unqualified *column* name.

If there is any ambiguity, QFORM issues an error during compilation.

Because you can refer to field names collectively through a screen record built upon all the fields linked to the same table, your forms might be easier to work with if you specify *table* for each field. For more information about declaring screen records, see the “INSTRUCTIONS Section” of this chapter.

A screen field can display a portion of a character string if you use subscripts in the *column* specification. Subscripts are a pair of comma-separated integers in brackets ([ ]) that indicate starting and ending character positions within a string value. But if you specify in the ATTRIBUTES section that two fields are linked to the same character column in the database, you *cannot* associate each field with a different substring of the same column.

The ATTRIBUTES section in the following file lists fields linked to columns in the **account and company** table.

```
DATABASE cms

SCREEN
{
                      Account\g \gDetails
p-------------------------------------------------------------q
|Account ID:  [f0         ]Company Name:[f11                 ]|
|Company ID:  [f1         ]Address1:    [f12                 ]|
|Credit Limit:[f2         ]Address2:    [f13                 ]|
|Discount:    [f3         ]City:        [f14                 ]|
|                           State:       [f15           ]      |
|                           Country:     [f16           ]      |
|                           ZIP:         [f17       ]          |
b-------------------------------------------------------------d
                                         [fb1      ][fb2      ]
}

TABLES
account
company

ATTRIBUTES
f0=account.account_id,
             comments="Account ID",
             noentry;
f1=account.comp_id,
             comments="Company ID",
             required;
f2=account.credit_limit,
             comments="Credit Limit",
             default=0.00,
             required;
f11=company.comp_name,
             comments="Company Name";
f12=company.comp_addr1,
             comments="Company Address 1";
f13=company.comp_addr2,
             comments="Company Address 2";
```

```
f14=company.comp_city,
              comments="Company Address 3";
f15=company.comp_zone,
              comments="Zone/State/County";
f16=company.comp_country,
              comments="Country";
f17=company.comp_zip,
              comments="ZIP/Postal Code";
f3=account.discount,
              comments="Discount",
              default=0.00,
              required;
fb1=formonly.fb_ok,
              config="Return OK",
              widget="button",
              comments="Return changes, store data and close window";
fb2=formonly.fb_cancel,
              config="Break Cancel",
              widget="button",
              comments="Ignore changes and close window";

INSTRUCTIONS

DELIMITERS "[]"
```

BYTE Data (BLOB)

In character mode, values from a column of data type BYTE are never displayed in a form; the words <BYTE value> are shown in the corresponding display field to indicate that the user cannot see the BYTE data. In GUI mode, the byte data may be rendered using the in-place viewer configured for the field. The image viewer reads the temporary file name extension and decides if it is to try to view this binary file. GUI clients also have a configuration file (blobtypes.txt) to associate binary files with the corresponding binary editor (e.g., .bmp could be associated to paintbrush). The following excerpt from a form specification file shows a TEXT field **resume** and a BYTE field **photo**. In this example, the BYTE field is short because only the words <BYTE value> are displayed. Similarly, you do not need to include more than one line in a form for a TEXT field. (The PROGRAM attribute that can display TEXT or BYTE values is described in “PROGRAM”.)

```
resume [f003 ]
photo [f004 ]
. . .
attributes
f003 = employee.resume;
f004 = employee.photo;
```

GUI Client configuration file BLOBTYPES.TXT

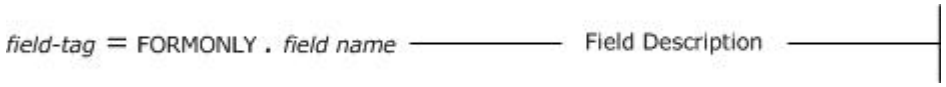
As mentioned already, GUI clients have a configuration file to associate binary files (based on their file extension) to an external editor. This file is a standard text file and can be modified to suit your client’s environment.

Example contents:

```
pdf C:\Program Files\Querix\progs\guidemo.exe,
netxt2 C:\WINNT\system32\MSPAINT.EXE,
#type editor,viewer
bmp mspaint,QuerixActiveX.imageviewer
jpg mspaint,QuerixActiveX.imageviewer
gif mspaint,QuerixActiveX.imageviewer
png mspaint,QuerixActiveX.imageviewer
tif mspaint,QuerixActiveX.imageviewer
tga mspaint,QuerixActiveX.imageviewer
pcx mspaint,QuerixActiveX.imageviewer
avi someapp,MediaPlayer.MediaPlayer.1
mpeg  someapp,MediaPlayer.MediaPlayer.1
txt notepad,QuerixActiveX.txtviewer
vi notepad.exe,QuerixActiveX.txtviewer
#vi notepad.exe,QuerixActiveX.imageviewer
wav someapp,MediaPlayer.MediaPlayer.1
rtf C:\WINNT\system32\dllcache\wordpad.exe,QuerixActiveX.txtviewer
doc C:\WINNT\system32\dllcache\wordpad.exe,QuerixActiveX.rtfviewer
#html notepad,QuerixActiveX.txtviewer
```

FORMONLY Fields

FORMONLY fields are not associated with columns of any database table or view. They can be used to enter or display the values of program variables. If the DATABASE section specifies FORMONLY, this is the only kind of field description that you can specify in the ATTRIBUTES section.



Element	Description
Field name	This is an identifier that you assign to a FORMONLY field. This identifier can also be used in 4GL statements that reference the field.
field-tag	This is the field tag, as declared in the SCREEN section.

Usage

Like other 4GL identifiers, *field name* cannot begin with a number. It can have up to 50 characters, including letters, numbers, and underscore (_) symbols.

If you specify one or more FORMONLY fields, 4GL behaves as if they formed a database table named **formonly**, with the field names as column names. The following fields are examples of FORMONLY fields:

```
DATABASE formonly

SCREEN
{
\gp-------------------------q\g
\g| \g[f0                   ]  \g|\g
\g| \g[f0                   ]  \g|\g
\g| \g[f0                   ]  \g|\g
\g| \g[f0                   ]  \g|\g
\gb-------------------------d\g
         [f1     ] [f2    ]
}


ATTRIBUTES
f0=formonly.f0;
f1=formonly.f1,config="Return OK",widget="button";
f2=formonly.f2,config="Break Cancel",widget="button";

INSTRUCTIONS
SCREEN RECORD sc_rec[4]  (
             formonly.f0
             )

        DELIMITERS "[]"
```

The Data Type Specification

The optional 4GL data type specification uses a restricted subset of the data type declaration syntax that the DEFINE, ALTER TABLE, and CREATE TABLE statements support. The data type *cannot* be declared here as a RECORD or as an ARRAY even if 4GL uses the field to display values from a program record or a program array; screen arrays are declared in another section of the form specification file. (It also cannot be SERIAL because SERIAL is an SQL data type, and only 4GL data types are allowed here.)

If you do not specify any data type, QFORM treats the field as type CHAR by default.

Do not assign a length to CHAR, DECIMAL, and MONEY fields because field length is determined by the display width in the SCREEN section.

For example, the demonstration application uses the following FORMONLY field to store the running total price for the order as items are entered:

```
f21=formonly.tax_total type MONEY;
```

You are required to specify a data type only if you also specify an INCLUDE or DEFAULT attribute for this field. 4GL performs any necessary data type conversion for the corresponding program variable during input or display.

LIKE

An alternative to defining the datatype statically in the form would be to use the keyword LIKE, which instructs the compiler to retrieve the datatype at compile time from the database. Again, 4GL evaluates the LIKE clause at compile time, not at runtime. If the database schema changes, you might need to recompile a program that uses the LIKE clause to describe a FORMONLY field in a form specification file.

BYTE and TEXT

Like a field linked to a database column, a FORMONLY field cannot display a BYTE value directly. The form displays the string <BYTE value> to indicate that the user cannot see the BYTE value. Similarly, you need not allocate more than one line on a form for a FORMONLY field of data type TEXT. You can assign the PROGRAM attribute to a FORMONLY field to display TEXT or BYTE values from 4GL variables.

The NOT NULL Keywords

The NOT NULL keywords specify that if you reference this screen field in an INPUT statement, the user must enter a non-null value in the field (these keywords are more restrictive than the REQUIRED attribute, which permits the user to enter a null value). This keyword is only valid for ‘formonly’ fields. An example follows:

```
f0=formonly.account_name NOT NULL;
```

For more information, see “REQUIRED”.

If the DATABASE section has the (deprecated) WITHOUT NULL INPUT clause, the NOT NULL keywords instruct 4GL to use zero (for number or INTERVAL data types) or blank spaces (for character data types) as the default value for this field in INPUT statements. The default DATE value is 12/31/1899. The default value for DATETIME fields is 1899-12-31 23:59:59.99999.

Multiple-Segment Fields (Wordwrap)

If you need the form to support entry or display of long character strings, you can specify *multiple-segment* fields that occupy several lines of the form. To create a multiple-segment field, repeat the same field tag in different fields of the layout in the SCREEN section, typically on successive lines. For more information, refer to the chapter 'Field Attributes'.

Subscripted Fields

We strongly advice against the use of subscripted fields. This feature only exists for legacy compatibility support.

QFORM can create a default form specification that references a database column of a character data type whose declared length in bytes is greater than (characters - 22), where *characters* is the width of the form. For such columns, QFORM generates two or more fields in the default specification (each with a different field tag but with the same field name) whose total length is the declared length of the corresponding database column.

In the ATTRIBUTES section of the default specification, the name of each such field is immediately followed (in brackets) by an ordered pair of comma-separated literal integers that identify the first and last of the bytes in the database column that the field displays. For example:

```
c3 = contact.description[1,64]
c4 = contact.description[65,128];
```

These are called *subscripted fields*, and they are generated for backward compatibility with the PERFORM forms compiler. Statements of 4GL that enter or display data in screen forms (CONSTRUCT, INPUT, and INPUT ARRAY) are difficult to use with subscripted 4GL fields beyond the first field.

Use a text editor to change such fields to segmented WORDWRAP fields.

Demo Application

Project: forms
Program: fm_field_subscript

Field Attributes

The following table summarizes the field attributes accepted by QFORM and their effects.

Attribute	Effect
AUTONEXT	Causes the cursor to advance automatically to the next field
CENTER	Aligns the contents of the field centrally

CENTURY	Specifies expansion of 2-digit years in DATE and DATETIME fields
CLASS	Defines the class associated with a widget type.
COLOR	Specifies the colour or intensity of values displayed in a field
COMMENTS	Specifies a message to display on the Comment line
CONFIG	Specifies configuration options associated with widget defined for the form field.
DEFAULT	Assigns a default value to a field during data entry
DISPLAY LIKE	Assigns attributes from **syscolatt** table that the **upscol** utility creates, associating attributes with specific database columns
DOWNSHIFT	Converts to lowercase any uppercase character data
EDITOR	Synonym for the PROGRAM field attribute.
FORMAT	Formats DECIMAL, SMALLFLOAT, FLOAT, or DATE output
INCLUDE	Lists a set of acceptable values during data entry
INVISIBLE	Does not echo characters on the screen during data entry
KEY	Specifies a label to be associated with a hot-key for this field.
LEFT	The field data is left aligned, overriding alignment properties associated with the underlying datatype.
NOENTRY	Prevents the user from entering data in the field
NOPROMPT	[character mode only]. Prevents the edit of a field with the 'SCROLL' attribute from using the prompt line to complete data where the field is full.
NOSCROLL	Limits the user input to the size of the edit field rather than the size of the underlying variable.
PASSWORD	Field data is rendered using asterisk characters, preventing the user from reading the form data.
OPTIONS	Specified miscellaneous options relevant to the class of widget associated with the field.
PICTURE	Imposes a data-entry format on CHAR or VARCHAR fields
PROGRAM	Invokes an external program to display TEXT or BYTE values
REQUIRED	Requires the user to supply some value during data entry
REVERSE	Causes values in the field to be displayed in reverse video
RIGHT	The field data is right aligned, overriding alignment properties associated with the underlying datatype.
SCROLL	Allows the user to input data in an edit field to the size of the underlying variable rather than the length of the edit field.
UPSHIFT	Converts to uppercase any lowercase character data
VALIDATE LIKE	Validates data entry with the **syscolval** table that the **upscol** utility creates, associating default values with specific database columns
VERIFY	Requires that data be entered twice when the database is modified
WIDGET	Specifies the widget type associated with this form field.
WORDWRAP	Invokes a multiple-line editor in multiple-segment fields

Each of these attributes is described in the chapter ‘Field Attributes’

Field Attribute Syntax

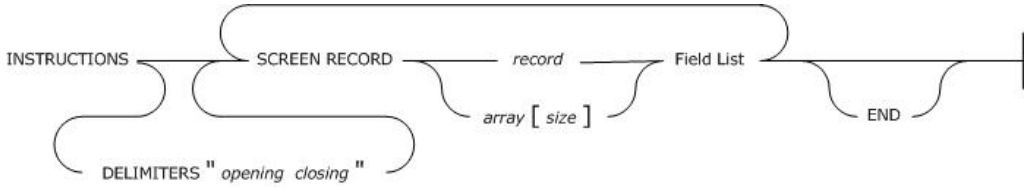
Syntax for assigning field attributes is described in the chapter Field Attributes. As the syntax diagram for ATTRIBUTES indicates, fields that have more than one attribute must separate successive attribute specifications with a comma (,), and must terminate the list of attributes with a semicolon (;), even if the attribute list is empty.

If a form links a view to a screen field that permits data entry or data editing, it is the responsibility of the programmer to test at runtime whether the view is updatable, especially if the view is based on another view.

INSTRUCTIONS Section

The INSTRUCTIONS section is an optional section of a form specification file. This section can declare non-default screen records and screen arrays.

The INSTRUCTIONS section appears after the last field description (or after the optional END keyword) of the ATTRIBUTES section.



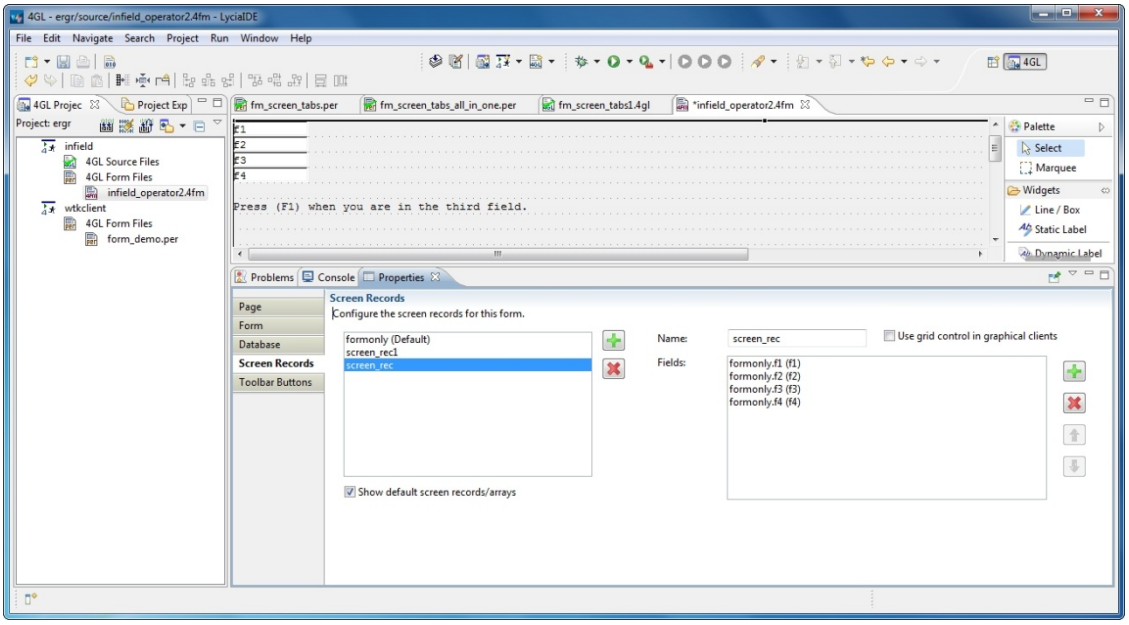
Element	Description
array	This is the 4GL identifier declared here for the screen array. (It is also the name of the screen record of each line of the array.)
closing	This is the closing field delimiter.
opening	This is the opening field delimiter.
record	This is the 4GL identifier declared here for the screen record.
size	This is a literal integer, between square brackets ([]), that specifies the number of screen records in the array.

The END keyword is optional and provides compatibility with earlier Informix products.

Screen Records

A *screen record* is a group of fields that screen interaction statements of the 4GL program can reference as a single object. By establishing a correspondence between a set of screen fields (the screen record) and a set of 4GL variables (typically a program record), you can pass values between the program and the fields of the screen record. In many applications, it is convenient to define a screen record that corresponds to a *row* of a database table.

Screen records within forms can be easily created and managed with LyciaStudio. Please refer to the appropriate guides for detailed information.



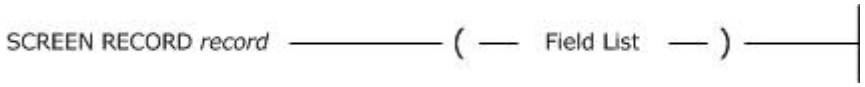
Default Screen Records

4GL recognizes *default screen records* that consist of all the screen fields linked to the same database table within a given form. QFORM automatically creates a *default record* for each table that is used to reference a field in the ATTRIBUTES section. The components of the default record correspond to the set of display fields that are linked to columns in that table.

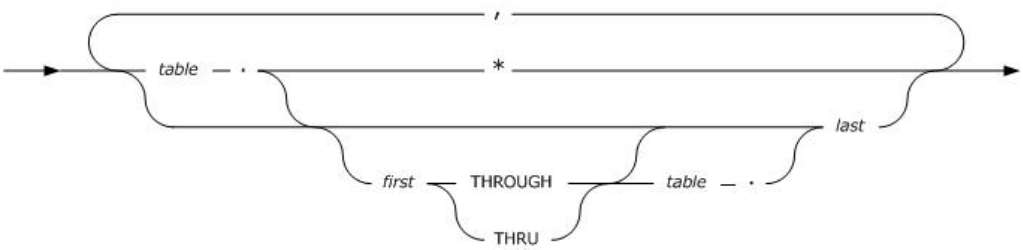
The name of the default screen record is the table name (or the alias, if you declared an alias for that table in the TABLES section). For example, all the fields linked to columns of the **customer** table constitute a default screen record whose name is **customer**. If a form includes one or more FORMONLY fields, those fields constitute a default screen record called **formonly**.

Non-default Screen Records

The INSTRUCTIONS section of a form specification file can declare non-default screen records. You use the SCREENRECORD keywords of the INSTRUCTIONS section to declare a name for the screen record and to specify a list of fields that are members of the screen record. A record declaration has this syntax.



The Field List has this syntax:



Element	Description
first	This is a field name, declared in the ATTRIBUTES section.
last	This is a field name declared after *first*.
record	This is a 4GL identifier declared for the screen record.
table	This is the name, alias or synonym of a table, or the FORMONLY keyword.

The field name is the SQL identifier of a database column linked to the field unless you specify FORMONLY as the table reference.

The *record* name of a non-default screen record can have up to 50 characters, and it must comply with the rules for 4GL identifiers.

Like the name of a screen field, the identifier of a screen record must be unique within the form, and it has a scope that is restricted to when its form is open. Statements can reference *record* only when the screen form that includes it is being displayed. QFORM returns an error if *record* is the same as the name or alias of a table in the TABLES section.

The List of Member Fields

The fields within a screen record are called members of the record. The list of member fields must be enclosed within a pair of parentheses. Use commas to separate elements of the list of field names.

You must specify the *table* qualifier if the field name is not unique among the fields in the ATTRIBUTES section or if table is a required. Otherwise, *table* is optional but including it might make the form specification file easier to read.

A screen/grid record can include screen fields whose identifiers have different *table* specifications (including the FORMONLY keyword). You can use the notation *table.** to include default screen records in the list of fields:

```
SCREEN RECORD sc_inv_lines[3]  (
             invoice_line.*,
             stock_item.*,
             formonly.line_net_total,
             formonly.line_tax_total,
             formonly.line_total
                   )
```

Here the asterisks (*) represent all of the fields in the form that the ATTRIBUTES section associated with columns in the **invoice_line** and **stock_item** tables. These fields do not necessarily correspond to all of the columns in these tables unless the form includes fields that are linked to all of the columns.

You can use the keyword THRU to specify consecutive fields, in the order of their listing in the ATTRIBUTES section from *field name1* to *field name2*, inclusive. (The keyword THROUGH is a synonym for THRU.) For example, the following instruction creates a screen record called **contact** from fields linked to some columns of the **contact** table. This record can simplify 4GL statements to update customer address and telephone data.

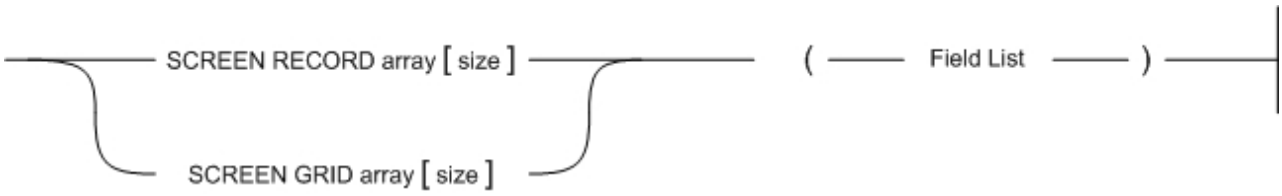
```
SCREEN RECORD contact
(contact.cont_addr1 THRU contact.cont_zip)
```

The order of fields in the portion of a screen record specified by the *table.** or THRU notation is the order of the field names within the ATTRIBUTES section.

Screen Arrays

A *screen array* is usually a repetitive array of fields in the screen layout, each containing identical groups of screen fields. Each *row* of a screen array is a screen record. Each *column* of a screen array consists of fields with the same field tag in the SCREEN section of the form specification file.

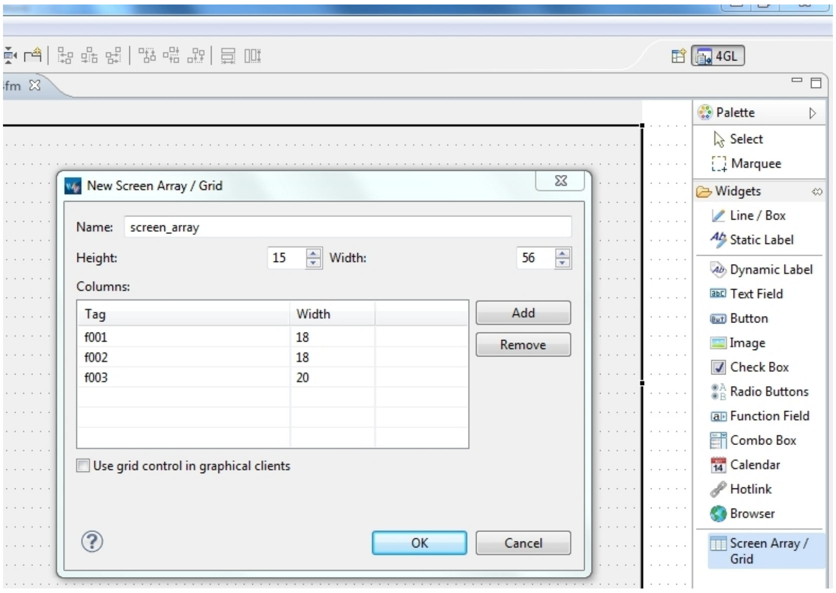
You must declare screen arrays in the INSTRUCTIONS section and use syntax like the syntax described for a screen record in the previous section, but with an additional parameter to specify the *number of screen records* in the array.



Element	Description
Record	Classic 4GL representation of a screen record array
GRID	Modern GUI representation of a screen record array
array	This is the 4GL identifier declared here for the screen array. (It is also the name of the screen record of each line of the array.)
size	This is a literal integer, between square brackets ([]), that specifies the number of screen records in the array.

The *size* value should be the number of lines in the logical form where the set of fields that comprise each screen record is repeated within the screen array.

In LyciaStudio screen arrays can be easily created and managed by means of the Graphical Form Editor. Please refer to the appropriate documentation for more detailed information.



For example, a SCREEN section might represent a screen array like this:

```
DATABASE cms

SCREEN
{
                    User List
\gp----------------------------------------------q    \g
\g| \gId#    Name      Password  Type  Contact     \g|   \g
\g+----------------------------------------------+    \g
\g|\g[f000][f001      ][f002     ][a ][f003     ]  \g|   \g
\g|\g[f000][f001      ][f002     ][a ][f003     ]  \g|   \g
\g|\g[f000][f001      ][f002     ][a ][f003     ]  \g|   \g
\g|\g[f000][f001      ][f002     ][a ][f003     ]  \g|   \g
\g|\g[f000][f001      ][f002     ][a ][f003     ]  \g|   \g
\g|\g[f000][f001      ][f002     ][a ][f003     ]  \g|   \g
\g|\g[f000][f001      ][f002     ][a ][f003     ]  \g|   \g
\g|\g[f000][f001      ][f002     ][a ][f003     ]  \g|   \g
\g|\g[f000][f001      ][f002     ][a ][f003     ]  \g|   \g
\g|\g[f000][f001      ][f002     ][a ][f003     ]  \g|   \g
\gb----------------------------------------------d    \g
 [fb4     ][fb5     ][fb6     ]      [fb2      ]
 Press:\g \gF4-Create\g \gF5-Modify\g \gF6\g \gDelete
}
```

This example requires a *size* of **[10]**. Except for the *size* parameter, syntax for specifying the identifier and the field names of a screen array is the same as for a simple screen record. Unlike 4GL program arrays, which can have up to three dimensions, 4GL screen arrays have only one dimension.

To illustrate the declaration of a typical screen array in more detail, consider the following fragment of a form specification file:

```
SCREEN
{
  Choose\g \gInvoice:
 \gp-------------------------------------------------------------------------q \g
 \g| \g[f1       ][f2                  ][f3        ][f4        ][f5         ] \g|
\g
 \g| \g[f1       ][f2                  ][f3        ][f4        ][f5         ] \g|
\g
 \g| \g[f1       ][f2                  ][f3        ][f4        ][f5         ] \g|
\g
 \g| \g[f1       ][f2                  ][f3        ][f4        ][f5         ] \g|
\g
 \g| \g[f1       ][f2                  ][f3        ][f4        ][f5         ] \g|
\g
 \gb-------------------------------------------------------------------------d \g
  [f7        ]  [f0        ]                    [f8        ][f6        ]

       }


TABLES
invoice
company

ATTRIBUTES
f1=invoice.invoice_id;
f2=company.comp_name;
f3=invoice.net_total;
f4=invoice.tax_total;
f5=invoice.inv_total;
f0=formonly.f0,config="F5 Edit",widget="button";
f8=formonly.f1,config="Return OK",widget="button";
f6=formonly.f6,config="F9 Cancel",widget="button";
f7=formonly.f7,config="F8 Print",widget="button";

INSTRUCTIONS
SCREEN RECORD sc_invoice_scroll[5]  (
                  invoice.invoice_id,
                  company.comp_name,
                  invoice.net_total,
                  invoice.tax_total,
                  invoice.inv_total
                  )

    DELIMITERS "[]"
```

The **sc_invoice_scroll** screen array has five rows and four columns and includes fields linked to columns from two database tables. Rows are numbered from 1 to 5. The screen record that follows the display label **Item 3** in the screen layout, for example, can be referenced as **sc_invoice_scroll[3]** in a 4GL statement.

If a screen array contains a default screen record, you can reference its fields in specific lines of the screen array (such as **net_total[5]** for the **f5** field in the last line), as if you had declared an array of records linked to that table.

You can reference *array-name* in the DISPLAY, DISPLAY ARRAY, INPUT, INPUT ARRAY, and SCROLL statements of 4GL but only when the screen form that includes the screen array is the current form.

Screen records and screen arrays can display program records. If the fields in the screen record have the same sequence of data types as the columns in a database table, you can use the screen record to simplify 4GL operations that pass values between program variables and rows of the database.

GRID ARRAY

To replace a screen array with a grid simply replace the word 'RECORD' with 'GRID' in the form definition, .per file.

As an example, we have used the Querix GUI Demo program, changing the Screen Record definition to Screen Grid, as below.

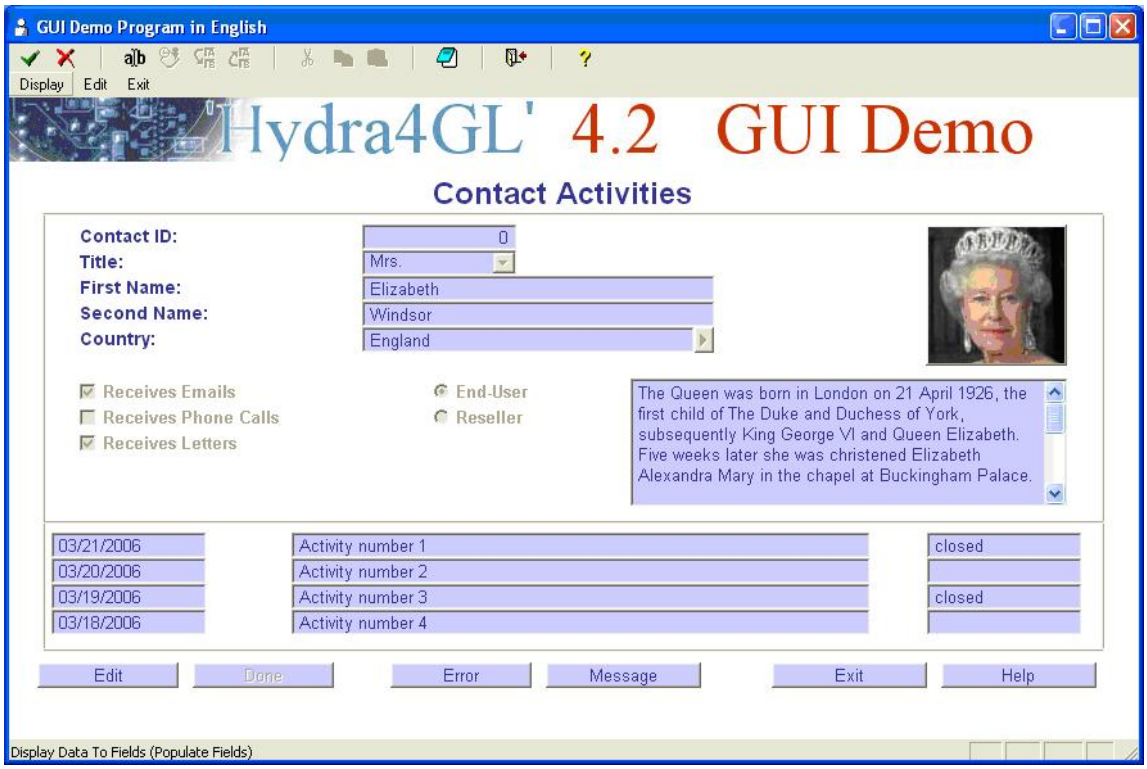
Therefore:

```
SCREEN RECORD sc_rec [3]  (
      formonly.act_date,
      formonly.act_activity,
      formonly.act_state
      )
```

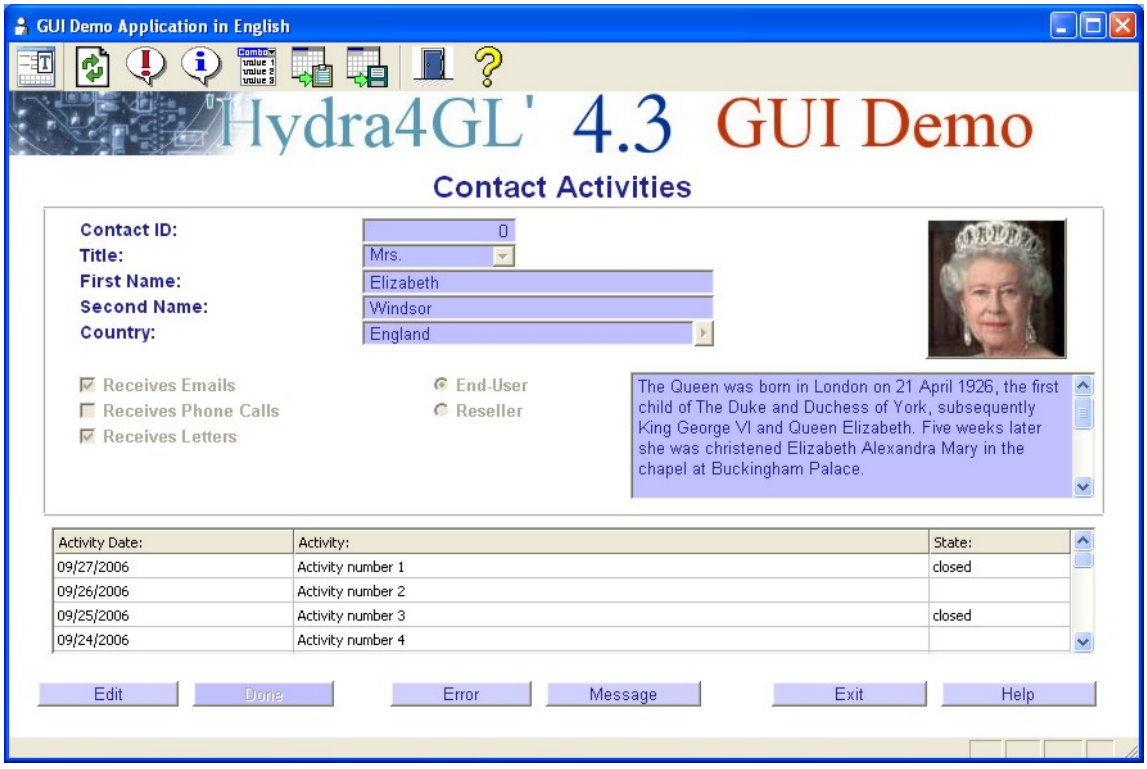
Changes to:

```
SCREEN GRID sc_rec [3]  (
      formonly.act_date,
      formonly.act_activity,
      formonly.act_state
      )
```

Using the guidemo demonstration program we can graphically illustrate the distinction this would make on the user interface. The following figures display the demo program showing the Screen Record syntax and the Screen Grid syntax.



GUI Demo program showing Screen Record



GUI Demo program showing Screen Grid

Grid Column Headers

Grid column headers can be defined in the form definition section. Here, it is possible to define the text header of a grid column, the event associated with clicking the mouse on the header and its alignment.

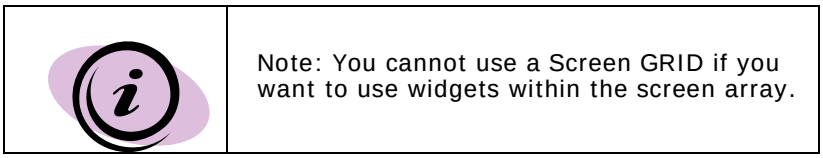
A code example of this in practice might look like the following:

```
ATTRIBUTES
f1=my_table.my_field, OPTIONS="HEADER='Date' KEY='F9' "
```

Screen Array with graphical widgets

Screen arrays are a modern approach of representing and manipulating 'Text' array data in a graphical table. Grid data displays data only in text; graphical widgets/controls such as radio buttons or check boxes are not yet supported.

If you need a screen array with graphical widgets, create a normal screen array and use the corresponding widgets.

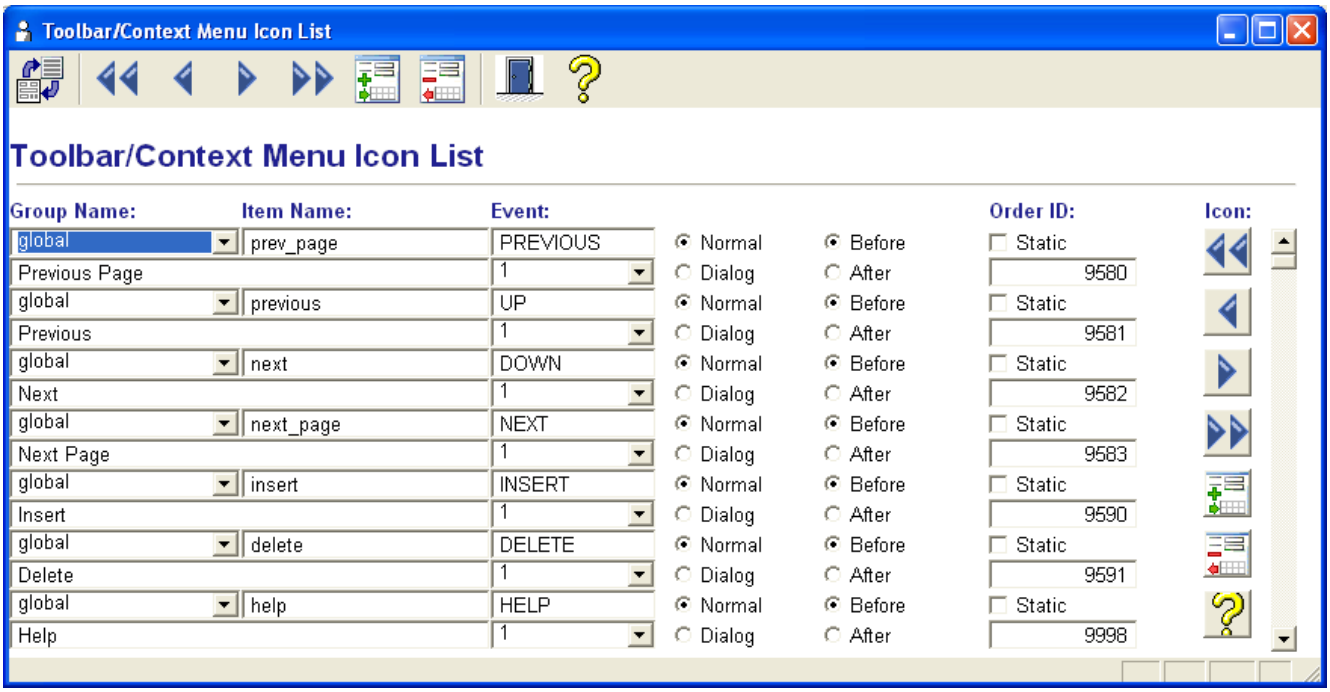


Example – SCREEN RECORD ARRAY with widgets (extracted from GUIDemo)

```
DATABASE formonly

SCREEN
{
[dl_header                                                              ]
\g-------------------------------------------------------------------\g
[dl_grp_nam   |dl_item       |dl_event]                    [dl_order_id][dl_ic]
[m_name     |m_item   |event    ][dialog  |act_time][stat    ]   [img]
[txt_en                     |fa       ]            [ord     ]
[m_name     |m_item   |event    ][dialog  |act_time][stat    ]   [img]
[txt_en                     |fa       ]            [ord     ]
[m_name     |m_item   |event    ][dialog  |act_time][stat    ]   [img]
[txt_en                     |fa       ]            [ord     ]
[m_name     |m_item   |event    ][dialog  |act_time][stat    ]   [img]
[txt_en                     |fa       ]            [ord     ]
[m_name     |m_item   |event    ][dialog  |act_time][stat    ]   [img]
[txt_en                     |fa       ]            [ord     ]
[m_name     |m_item   |event    ][dialog  |act_time][stat    ]   [img]
```

```
[txt_en                      |fa       ]            [ord     ]
[m_name     |m_item   |event    ][dialog  |act_time][stat    ]   [img]
[txt_en                      |fa       ]                  [ord     ]
}

#[dl_grp_nam   |dl_item      |dl_event]
[dl_order_id][dl_ic]
#[m_name       |m_item       |event    |dialog  |action_time]   [stat    ]
[image]
#[txt_en                     |fa       ]                       [ord     ]


ATTRIBUTES
m_name=formonly.m_name,
             widget="combo",
             include=("global",
                   "guidemo",
                   "gd_lang",
                   "gd_contact",
                   "gd_co_edit",
                   "gd_co_ed_gr",
                   "country_sel",
                   "gd_col_main",
                   "gd_col_1",
                   "gd_col_2",
                   "gd_col_3",
                   "control_w",
                   "control_j",
                   "msg_box",
                   "theme",
                   "dde_main",
                   "web_demo",
                   "help_html",
                   "blob_demo",
                   "img_brows");
m_item=formonly.m_item;
act_time=formonly.action_time,
             config="0 Before 1 After",
             widget="radio";
dialog=formonly.dialog,
             config="0 Normal 1 Dialog",
             widget="radio";
event=formonly.event;
txt_en=formonly.txt_en;
ord=formonly.ord;
img=formonly.image,
             config="icon32/exit01.ico F9",
             widget="bmp";
```

```
dl_header=formonly.dl_header,
              config="{Toolbar/Context Menu Icon List}",
              widget="label",
              color=bold Blue;
dl_order_id=formonly.dl_order_id,
              config="{Order ID:}",
              widget="label",
              color=bold Blue;
dl_event=formonly.dl_event,
              config="Event:",
              widget="label",
              color=bold Blue;
dl_ic=formonly.dl_ic,
              config="Icon:",
              widget="label",
              color=bold Blue;
stat=formonly.stat,
              config="1 0 Static",
              widget="check";
dl_grp_nam=formonly.dl_grp_nam,
              config="{Group Name:}",
              widget="label",
              color=bold Blue;
dl_item=formonly.dl_item,
              config="{Item Name:}",
              widget="label",
              color=bold Blue;
fa=formonly.action type integer,
              widget="combo",
              include=(1,2,3,4);

INSTRUCTIONS
SCREEN RECORD tb_list_ar[7] (
              formonly.m_name,
              formonly.m_item,
              formonly.event,
              formonly.stat,
              formonly.action,
              formonly.ord,
         formonly.txt_en,
         formonly.dialog,
         formonly.action_time,
         formonly.image
              )

DELIMITERS "||"
```


Toolbar/Context Menu Icon List
Toolbar/Context Menu Icon List
Group Name:
Item Name:
Event:
Order ID:
Icon:
global
prev_page
PREVIOUS
Normal
Before
Static
Previous Page
1
Dialog
After
9580
global
previous
UP
Normal
Before
Static
Previous
1
Dialog
After
9581
global
next
DOWN
Normal
Before
Static
Next
1
Dialog
After
9582
global
next_page
NEXT
Normal
Before
Static
Next Page
1
Dialog
After
9583
global
insert
INSERT
Normal
Before
Static
Insert
1
Dialog
After
9590
global
delete
DELETE
Normal
Before
Static
Delete
1
Dialog
After
9591
global
help
HELP
Normal
Before
Static
Help
1
Dialog
After
9998

Field Delimiters

You can change the delimiters that 4GL uses for fields from brackets ([ ]) to any other printable character, including blank spaces. The DELIMITERS instruction tells 4GL which symbols to use as field delimiters when it displays the form on the screen. The opening and closing delimiter marks must be enclosed within speech marks (").

The following specifications display < and > as opening and closing delimiters of screen fields:

```
INSTRUCTIONS
DELIMITERS "<>"
END
```

Each delimiter occupies a space, so two fields on the same line are ordinarily separated by at least two spaces. To specify only one space between consecutive screen fields, use the following procedure.

To use only one space between fields

1. In the SCREEN section, substitute a pipe symbol (|) for paired back-to-back ( ][ ) brackets that separate adjacent fields.

2. In the INSTRUCTIONS section, define some symbol as both the beginning and the ending delimiter.

For example, you could specify "| |" or "/ /" or ": :" or " " (blanks).

The following specifications substitute | for][between adjacent fields in the same line of the screen layout and display a colon as both the opening and closing delimiter:

```
SCREEN
{
. . .
Contact Name-[1    |f2   ]
. . .
}
. . .
INSTRUCTIONS
DELIMITERS "::"
```

Here the fields whose tags are **f1** and **f2** will be displayed as:

```
Contact Name-: | :
```

If you substitute blanks for colons as DELIMITERS symbols, field boundaries are not marked (or are only marked if they have attributes that contrast with the surrounding background).

	QFORM requires square brackets ([ ]) in the SCREEN section of a form specification file to define the start and end of any field, regardless of any DELIMITERS instruction.

Example:

Field Delimiter demo application located in the forms project.

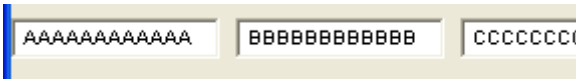
Form Screen Section

```
SCREEN
{
[f0         ][f1         ][f2          ]
}
ATTRIBUTES
f0=formonly.f0;
f1=formonly.f1;
f2=formonly.f2;
INSTRUCTIONS
DELIMITERS "[]"
```

Client Result

Text Client

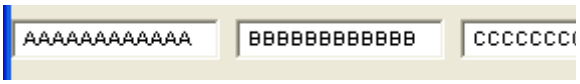
GUI Client




```
SCREEN
{
[f0          ][f1          ][f2          ]
}
ATTRIBUTES
f0=formonly.f0;
f1=formonly.f1;
f2=formonly.f2;
INSTRUCTIONS
DELIMITERS "||"
```

Text Client

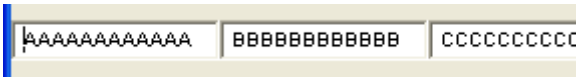
GUI Client




```
SCREEN
{
[f0          |f1          |f2          ]
}
ATTRIBUTES
f0=formonly.f0;
f1=formonly.f1;
f2=formonly.f2;
INSTRUCTIONS
DELIMITERS "||"
```

Text Client

GUI Client



Default Attributes

Field attributes can also be specified in two special tables in the database, **syscolval** and **syscolatt**. These tables are maintained by the **upscol** utility. QFORM searches these tables for default validation and display attribute specifications. It applies these specifications to form fields whose names match the names of the specified database columns or that reference these columns in the DISPLAY LIKE or VALIDATE LIKE attribute specifications.

The schema of the **syscolval** table is as follows.

Column	Data Type	Column	Data Type
tabname	CHAR (18)	attrname	CHAR (10)
colname	CHAR (18)	attrval	CHAR (64)

The schema of the **syscolatt** table is as follows.

Column	Data Type	Column	Data Type
tabname	CHAR (18)	underline	CHAR (1)
colname	CHAR (18)	blink	CHAR (1)
seqno	SERIAL	left	CHAR (1)
color	SMALLINT	def_format	CHAR (64)
inverse	CHAR (1)	condition	CHAR (64)

Here, **tabname** and **colname** are the names of the table and column, respectively, to which the attributes apply. **colname** can be neither a BYTE nor a TEXT column.

Valid values for the **attrname** and **attrval** columns in **syscolval** are shown in the following table.

attrname	attrval
AUTONEXT	YES, or NO (which is the default value)
COMMENTS	Explained elsewhere in this chapter.
DEFAULT	Explained elsewhere in this chapter.
INCLUDE	Explained elsewhere in this chapter.
PICTURE	Explained elsewhere in this chapter.
SHIFT	UP, DOWN,or NO (which is the default value)
VERIFY	YES, or NO (which is the default value)

QFORM adds the attributes from these tables to any attributes that are listed in the form specification file. In case of conflict, attributes from the form specification file take priority. 4GL applies the resulting set of field attributes during execution of INPUT and INPUT ARRAY statements (by using **syscolval**) and during execution of DISPLAY and DISPLAY ARRAY statements (by using **syscolatt**).

The **color** column in **syscolatt** stores an integer that describes colour (for colour terminals) or intensities (for monochrome terminals).

The next table shows the displays specified by each value of **color** and the correspondence between default colour names, number codes, and intensities.

Number	Colour Terminal	Monochrome Terminal
0	WHITE	NORMAL †
1	YELLOW	BOLD
2	MAGENTA	BOLD
3	RED	BOLD †
4	CYAN	DIM
5	GREEN	DIM
6	BLUE	DIM †
7	BLACK	DIM

† If BOLD is specified as the attribute, the field is displayed as RED on a colour screen. If the keyword DIM is specified as the attribute, the field is displayed as BLUE on a colour screen. Colour terminals display NORMAL as WHITE.

The background for colours is BLACK in all cases. The same keywords are also supported by the COLOR attribute for colour and monochrome terminals.

The valid values for **inverse**, **underline**, **blink** and **left** are Y (yes) and N (no). The default for each of these columns is N; that is, normal display (bright characters in a dark field), no underline, steady font, and right-aligned numbers. Which of these attributes can be displayed simultaneously with the colour combinations or with each other is terminal dependent.

The **def_format** column takes the same string that you would enter for the FORMAT attribute in a screen form; do not use quotation marks.

The **condition** column takes string values that are a restricted set of the WHERE clauses of a SELECT statement, except that the WHERE keyword and the column name are omitted. 4GL assumes that the value in the column identified by **tabname** and **colname** is the subject of all comparisons.

Examples of valid entries for the **condition** column follow:

- <= 100
- MATCHES "[A-M]*"
- BETWEEN 101 AND 1000
- IN ("CA", "OR", "WA")
- >= 1001
- NOT LIKE "%analyst%"

The VALIDATE statement compares the members of a program record or variable list to the validation rules in **syscolval**. The INITIALIZE statement can read the default values in **syscolval** for a list of columns and assign these values to a corresponding list of 4GL variables.

Some statements (including CONSTRUCT, DISPLAY, DISPLAY ARRAY, ERROR, INPUT, INPUT ARRAY, MESSAGE, PROMPT, OPEN WINDOW, and OPTIONS) support an ATTRIBUTE clause that can specify colour and intensity attributes.

The default attributes in **syscolatt** can be over-ridden. This is done either by assigning other attributes in the form specification file, or in the ATTRIBUTE clause of one of the following statements:

- CONSTRUCT
- DISPLAY
- DISPLAY ARRAY
- INPUT
- INPUT ARRAY

If the current 4GL statement is one of these and includes an ATTRIBUTE clause, the field displays only the attributes that are specified in that clause. For example, if a column is designated as RED and BLINK in **syscolatt**, or in the form specification file, and your 4GL program executes the following statement, the field has only the BLUE attribute, not blinking BLUE:

```
DISPLAY . . . ATTRIBUTE BLUE
```

If an ATTRIBUTE clause is present in the currently executing statement, there is no implicit carry-over of display attributes from the compiled form (except FORMAT).

Precedence of Field Attribute Specifications

Lycia uses these rules of precedence (highest to lowest) to resolve any conflicts among multiple defined display attribute specifications:

1. The ATTRIBUTE clause of the current 4GL statement

2. The field descriptions in the ATTRIBUTES section of the current form
3. The default attributes specified in the **syscolatt** table of any fields linked to database columns
4. The ATTRIBUTE clause of the most recent OPTIONS statement
5. The ATTRIBUTE clause of the current form in the most recent DISPLAY FORM statement
6. The ATTRIBUTE clause of the current 4GL window in the most recent OPEN WINDOW statement

Default Attributes in an ANSI-Compliant Database

In a database that is not ANSI compliant, the default screen attributes and validation criteria that you specify with the **upscol** utility are stored in two tables, **syscolval** and **syscolatt**. These defaults are available to every user of a form that references the specified column of the database.

In an ANSI-compliant database, however, the separate *owner*.**syscolval** and *owner*.**syscolatt** tables are created for each user of the **upscol** utility. These tables store the default specifications of that individual user. Which set of tables is used by QFORM depends on the nature of the request.

If the TABLES section specifies a table alias for *owner.table*, QFORM uses the **upscol** tables of the owner of *table*. If that user owns no **upscol** tables, no defaults are assigned to fields associated with that table alias. If the TABLES section of the form does not specify a table alias that includes the owner of a database table, the **upscol** tables owned by the user running QFORM are applied to fields associated with that database table unless the user owns no **upscol** tables. In the ATTRIBUTES section, field descriptions of the following forms use **upscol** tables (if they exist) owned by whoever runs QFORM, unless *table* is an alias that specifies a different owner:

```
field-tag = . . . DISPLAY LIKE table.column
field-tag = . . . VALIDATE LIKE table.column
```

If *table* is an alias for *owner.table*, QFORM uses the **upscol** tables of the owner specified by *table*, if they exist. If no **upscol** tables exist, the DISPLAY LIKE and VALIDATE LIKE attributes have no effect. If *owner* is not the correct owner, the compilation fails and an error message is issued.

References

INITIALIZE statement, VALIDATE statement

KEYS Section

The KEYS section is used to specify the key label for a button which would then be permanently attached to the dynamic toolbar. The general syntax of a key-associated button specification is:

```
SCREEN {
..
}
ATTRIBUTES
…
KEYS
Key = “Button label”[(icon)] [ORDER integer] [STATIC]
```

The button label

Button label stands for a character string which will be displayed to the button. The button label is also displayed as a tooltip when the mouse cursor is positioned over the button. If you don’t want the button to have a label, you can include an empty string instead of the label. However, this will lead to the empty tooltip message.

You can switch off the tooltips and button labels using the corresponding script options (see Graphical Client Reference)

The button icon

The *icon* stands for the name of a file containing a picture you want to become a button icon, preceded with its path, if necessary. The default size of the button icon is 16x16. If you use larger icons, they can be truncated or distorted in other ways. If you use smaller pictures, they will be displayed to the top left corners of the places allocated for the icons.

You can change the size of the displayed button images using script icon.Height and icon.Width script options:

It is advisable that you specify icons with equal height and width and of standard sizes like 9x9, 16x16, 32x32. You can make icons larger and with different height and width sizes, but it can lead to unexpected display results. When you specify a new icon size, you should specify the icon images that fit this size.

The ORDER keyword

The ORDER keyword and an integer value following it are used to specify the order in which the buttons will be placed on the toolbar. The program places the buttons on the toolbar one by one from left to right according to the order values of the buttons (in the ascendant order). If one or several buttons have no ORDER parameter, they will appear at the end of the toolbar in the order in which they are specified in the .per file.

Therefore, a key button with a label, icon, and positional order would be defined as follows:

```
KEYS
F1 = “This is a global label for F1” (“F1_icon.jpg”) ORDER 3
```

The STATIC keyword

The STATIC keyword ensures that the toolbar button label is visible (though disabled and grey) even then there is no corresponding key event for this button. In this way the toolbar button will be always visible.

ACTIONS Section

The action label specification works like the key labels in the KEYS section. They can be defined using the global ACTIONS section form using the following syntax, which would then be permanently attached to the Toolbar:

```
SCREEN {
..
}
ATTRIBUTES
…
ACTIONS
Action = “Button label”[(icon)] [ORDER integer] [STATIC]
```

With an icon image and positional order number required, the key label would be defined as follows:

```
ACTIONS
act_print_invoice = “This is a global label for the action act_print_invoice”
(“print.jpg”) ORDER 3
```

If the .per file contains both ACTIONS and KEYS sections, the KEYS section should come first, and the ACTIONS section should follow it.

The STATIC keyword

The STATIC keyword ensures that the toolbar button label is visible (though disabled and grey) even then there is no corresponding key event for this button. In this way the toolbar button will be always visible.

CHAPTER 3

Visual Form Objects

As we have already discussed, a form consists of visual form objects such as static lines, static text labels, fields and graphical widgets. This chapter will discuss these visual form objects in greater detail.

Static Labels

Static form text is simply any text which is defined in the screen section without field delimters [|] . This is the traditional way of defining static text information such as field labels.

Example of a static field label 'Company' in the screen section:

```
Company: [f_comp]
```

Static labels have no additional attributes/properties in the Attribute section. When changing to a GUI environment, we strongly recommend replacing any existing static labels with dynamic label widgets (explained later).

Static Lines

Static lines, identical to static labels, are line drawing symbols which only exist in the screen section. They do not have any additional attributes/properties in the Attribute section of the form definition.

Text Edit Field

The text edit field allows to display/enter data and associate them with a program variable or database table field. This is the most common generic form of a field which already existed in the original Informix 4GL language. Text fields are referenced as field tags in the screen section and configured in the attribute section. There is a vast range of additional properties which can be assigned to a text field. Text fields have no 'class' or 'config' attribute configuration.

Graphical Widgets

Graphical widgets can be seen as advanced 'fields'. The base definition in the screen section and in the attributes section are identical but they have the attribute 'widget' which defines what type of graphical widget it is and one or both additional attributes 'class' and 'config'.

The graphical widgets can be categorised in labels, buttons, fields or a combination of them. It should be noted that graphical widgets are only supported in graphical clients and will be rendered as normal text edit fields in text mode clients.

If your application is required to support both text mode clients and graphical clients, you can either create separate form files and query the client type at runtime to decide which form to use, or use only normal edit text fields and reference them as graphical widgets by using the scripting language. It should be noted that both scenarios involve additional programming work and if you don't have to support text mode clients, we strongly recommend utilising the graphical widgets.

Widget	Widget Name	Button functionality	Displays its field data as text	Allows field data manipulation in a graphically controlled fashion	Displays its field data using an image viewer
Dynamic Label	Label	No	Yes	No	No
Combo Box	combo	No	Yes	Yes	No
Radio Button	Radio	Yes	No	Yes	No
Check Box	Check	Yes	No	Yes	No
Function Field	Field_bmp	Yes	Yes	Yes	No
Button Widget	button	Yes	No	No	No
BMP button	Bmp	Yes	No	No	Yes
Browser Widget	Browser	No	No	No	Yes
Hotlink Widget	Hotlink	Yes	No	No	No
Calendar Widget	Calendar	No	Yes	Yes	No

Static Graphical Lines (graphical characters)

You can include graphical characters in the SCREEN section to place boxes and other rectangular shapes in a screen form. Use the following characters to indicate the borders of one or more boxes on the form.

Symbol	Purpose
P	To mark the upper, left corner of a box.
Q	To mark the upper, right corner of a box.
B	To mark the lower, left corner of a box.
D	To mark the lower, right corner of a box.
-	To show segments of horizontal line.
|	To show segments of vertical lines.
+	To show an intersection between a horizontal and vertical line.

The meanings of these seven special characters are derived from the **acsc** specifications in the **terminfo** files, respectively. 4GL substitutes the corresponding graphics characters when you display the compiled form.

Once the form has the desired configuration, use **\g** to indicate when to begin graphics mode and when to end graphics mode.

Insert **\g** before the first **p**, **q**, **d**, **b**, hyphen, or pipe symbol that represents a graphical character. To leave graphical mode, insert **\g** after the **p**, **q**, **d**, **b**, hyphen, or pipe symbol.

Do not insert **\g** into original white space of a screen layout. The backslash should displace the first graphics character in the line and push the remaining characters to the right. The process of indicating graphics distorts the appearance of a screen layout in the SCREEN section, compared to the corresponding display of the screen form.

You can include other graphical characters in a form specification file, but the meaning of a character other than the **p**, **q**, **d**, **b**, hyphen, and pipe symbol is terminal dependent.

To use graphical characters, the system terminfo file must include entries for specific variables.

The following table shows what variables need to be set in the **terminfo** file.

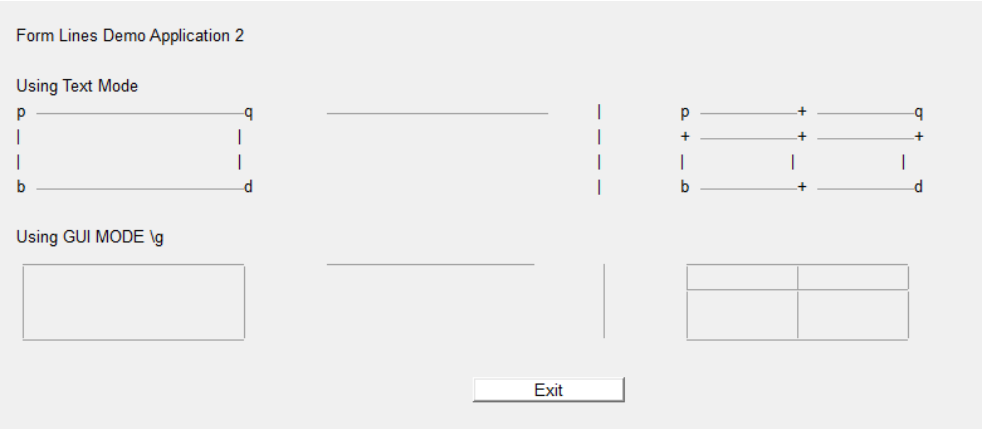
Terminfo Variable	Description
Smacs	The escape sequence to enter graphical mode.
Rmacs	The escape sequence to leave graphical mode.
Acsc	The concatenated, ordered list of ASCII equivalents for the graphical characters needed to draw the borders of a box.

Rectangles within Forms

You can use the built-in functions fgl_drawline() and fgl_drawbox() to enclose parts of the screen layout within rectangles. Rectangles that you draw with FGL_DRAWBOX( ) are NOT part of the displayed form. Each time that you execute the corresponding DISPLAY FORM or OPEN WINDOW...WITH FORM statement, you must also redraw the rectangle.

Example:

```
Using Text Mode
p---------------q     -----------------     |     p-------+-------q
|               |                           |     +-------+-------+
|               |                           |     |       |       |
b---------------d                           |     b-------+-------d

Using GUI MODE \\g
\gp---------------q     -----------------     |     p-------+-------q \g
\g|               |                           |     +-------+-------+ \g
\g|               |                           |     |       |       | \g
\gb---------------d                           |     b-------+-------d \g
```



Avoid any intersection between the rectangle and any field or 4GL reserved line because the rectangle will be broken at the intersection when anything is displayed in the field or in the reserved line.

Drawing graphical lines in the graphical form editor

The graphical form editor (part of LyciaStudio) allows you to draw, position and size graphical lines. Intersection points (line crossing or corners) will automatically produce the required graphical character.

Demo Application

Location:

Project: forms
Program: fm_screen_line1 and fm_screen_line2

Static Label

Static labels are static text strings which are defined in the screen section. As the name already tells us, static label cannot be manipulated at runtime within the 4GL language. Static labels are supported for legacy reasons to allow a smooth migration from classic I4gl or other 4gl clone vendors to Querix.

They are simply defined in the screen section of a form and have no attributes in the attributes section.

The problems with the legacy static text labels are as follows:

a) you cannot manipulate them at runtime
b) you can not define attributes directly (you need to search/match static labels and assign properties using a separate script)
c) a form definition works with non-proportional fonts. This means if, for example, a string with 80 characters were to be defined in a form, it would require the entire line in the forms screen section. If it is rendered using a proportional font at runtime (GUI client), it will only allocate about 60% of the line. This means there is always a lot of wasted/un-usable space in forms using static labels.

We realise that when migrating from another 4GL environment it would be large job to convert all your static labels to dynamic labels. For this reason there are a number of utilities available in the LyciaStudio suite to simplify this work and to allow for a staged migration process.

To merge multiple static labels (single words) to one string, simply enclose the spaces within graphical escape sequences \g \g

Example:

```
My\g \gString
```

Using the graphical form editor, this work is done for you automatically. It also offers tools to merge single words to a string or even to a dynamic widget.

Dynamic Label widget

The dynamic label widget creates a simple form label with a static location. Dynamic labels do not merge with normal (DISPLAY AT) or static form labels. To display data to this widget, you will need to use a normal DISPLAY TO *field* 4GL statement.

Any text (other than normal edit fields and widgets) should be displayed using dynamic labels, not using static labels or DISPLAY AT statements. Dynamic labels overcome many problems that may be experienced when using static labels. For example, you can already define attributes such as color, bold, alignment and underlined. Also, dynamic labels can be referenced like any other field. This means, to manipulate them using the scripting language, you do not need to search, match and reference text displayed in dynamic labels. Finally, there are also no merging issues.

Another advantage is when using modern proportional fonts; the displayed string length can be larger than the dynamic label's field length specified in the form.

Strings that are too large to be displayed in a dynamic label will be clipped. This means the developer will always have the assurance that no object which is displayed behind a dynamic label will ever be overwritten by an overlapping string label.

When changing the layout of a form, any 4GL DISPLAY AT statements must be adjusted accordingly. Again, when using dynamic labels, this problem no longer exists.

By default, text displayed is aligned left and numeric values are aligned right. This can be changed by using the attribute LEFT and RIGHT.

The content of the input field is rendered as the label of the field. To manipulate the font, the script must match the dynamic label like a normal field with *table name* and *field name* resource specifiers.

Attributes Syntax:

```
<tag> =    <field>,
           config=<string initially rendered when the form is displayed>,
           widget="label"
```

Attribute Example

```
ATTRIBUTES
dl_4=formonly.dl_example,
     config="{String initialised by the form}",
```

```
widget="label";
```

Properties

Dynamic labels require their widget attribute be set to ‘label’. The label can be initialised in the form definition by using the config property. Additionally, it will be able to take full advantage of the properties LEFT, RIGHT, UNDERLINE, COLOR and BOLD.

widget

The name of the dynamic label widget is ‘label’

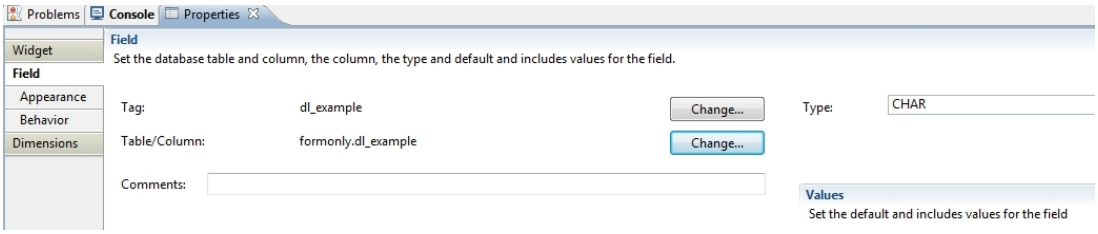
```
widget = “label”
```

config

The config property is simply a string which will be used to initialise the label when the form is displayed.

```
config = “{First Name:}”
```

Example properties using the graphical form editor



How to Create a Dynamic Label - Example

The following code creates a dynamic label with an initial string. Then, another string will be displayed followed by a numeric value.

Location:

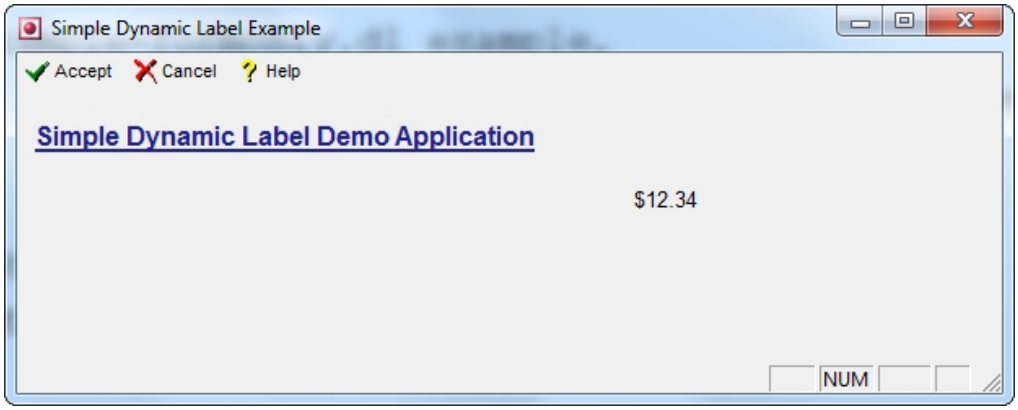
Project: forms
Program: fm_field_widget_dynamic_label1 ...~2 and ...~simple

Example .per Form Code:

```
DATABASE formonly

SCREEN
{
 Simple\g \gDynamic\g \gLabel\g \gDemo\g \gApplication

 [dl_example                          ]
}
```

```
ATTRIBUTES
dl_example=formonly.dl_example,
             config="{String initialised by form}",
             widget="label";

INSTRUCTIONS
DELIMITERS "[]"
```

Example screenshot



Text Field Box (Normal Field)

The edit text box is the most generic field. It is supported by all clients and rendered as a normal field. It can be used to display and enter data in text format and supports all attributes with the exception of the widget dedicated attributes class and widget.

Attributes Syntax

```
<tag> = <field>;
```

Attribute Example

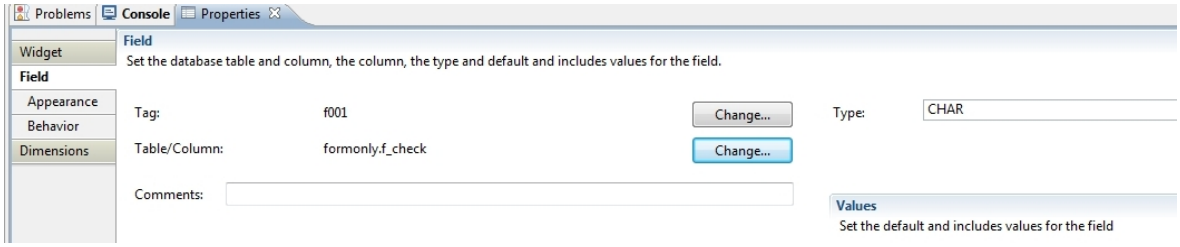
The following Attributes code sample creates a check box.

```
ATTRIBUTES
f_check=formonly.f_check, color=blue;
```

Properties

Different to the field and control widgets, the generic form field has no advanced properties but all field attributes are supported.

Example attributes using the graphical form editor



How to Create a Form Field - Example

The following code sample creates a simple form with three text fields.

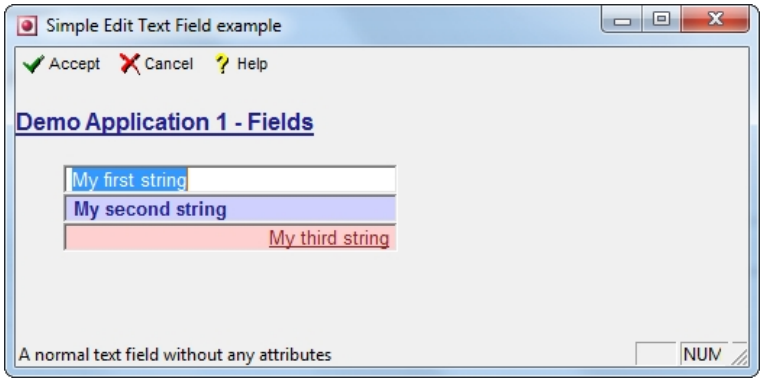
Location:

Project: forms
Program: fm_field_simple, fm_field1 and fm_field2

Example .per Form Code:

```
DATABASE formonly

SCREEN
{
[dl_header                ]

   [field1             ]
   [field2             ]
   [field3             ]
}


ATTRIBUTES
field1=formonly.field1;
field2=formonly.field2,
      color=bold Blue;
field3=formonly.field3,
      right,
      color=Red underline;
dl_header=formonly.dl_header,
      config="{Demo Application 1 - Fields}",
      widget="label",
      color=bold Blue underline;

INSTRUCTIONS

DELIMITERS "[]"
```

Example Screen Shot



Check Boxes widget

Check boxes are used for making binary choices and each check box relates to a single variable. Unlike radio buttons, where they are used in groups, it is possible to select all or any of the options.

The programmer can specify the value that should be written to the underlying variable when checking and un-checking this box. It should also be noted that if the value of the underlying variable is neither of the two values, it will show not ticked but will not modify this value by default. The user still needs to select the box to change the value of the variable.

Alternatively, you can use a check box to trigger key and action events, instead of writing values to variables directly. This may be required if you want to execute some program logic straight after the user has clicked the check box.

In ASCII mode, check boxes display as normal fields.

Attributes Syntax

```
<tag> =    <field>,
           config=<checked value> <un-checked value> <Label Text>,
           widget=”check”;
```

Attribute Example

The following Attributes code sample creates a check box.

```
ATTRIBUTES
f_check=formonly.f_check,
     config="1 0 {Checked is 1 and not checked is 0}",
     widget="check";
```

Properties

widget

The name of the check box field widget is ‘check’.

```
widget = “check”
```

class

If no class attribute is specified (or the attribute class=default), it will behave like a normal data check box.

```
Class="default"
```

If the attribute class=key is specified, it behaves like a button with two key or action events.

```
Class="key"
```

config

For a normal (class=default) check box, the configuration string consists of the checked value, unchecked value and the check box text (the label next to the check box).

```
config="1 0 {Check box label text}"
```

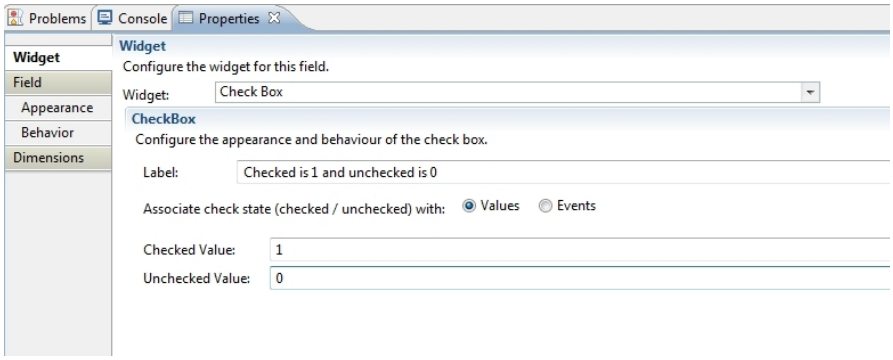
If you do not set a default value, the check box is set to a null string when processing an input statement with defaults.

When clicking the check box, the value in the field is changed, however any AFTER FIELD logic associated with the field will not be triggered unless the user attempts to leave the field either by pressing 'tab' or by selecting another field with the mouse.

For a 'KEY' or 'Action' check box (class=key), the configuration string consists of the event to be fired when the box is checked, the event for non-checked and the check box text (the label next to the check box).

```
config="F1 F2 {Check box label text}"
```

Example properties using the graphical form editor



Invoking a Key Code or Action (class="key")

You can overcome the issue of returned string length and after field statements by invoking a single key code or action event name instead of returning a string. To do this you will need to use the code

```
class=”key”
```

You need to add this to the attribute section of the form file, where the check box is declared.

To define the key presses use the same config syntax as when you use normal field values.

```
config="F1 F2 {Send E-Mail now ?}
```

You can also enable or disable check boxes that use the key class within 4GL programs. The code sample below shows the lines needed to firstly enable and secondly to disable the check box called myButton:

```
DISPLAY “!” TO myButton
DISPLAY “*” TO myButton
```

How to Create a Check Box - Example

The following code sample creates a form with a check box. Pressing the Accept key (or clicking the Accept toolbar icon) will read the check box value and display it to the screen.

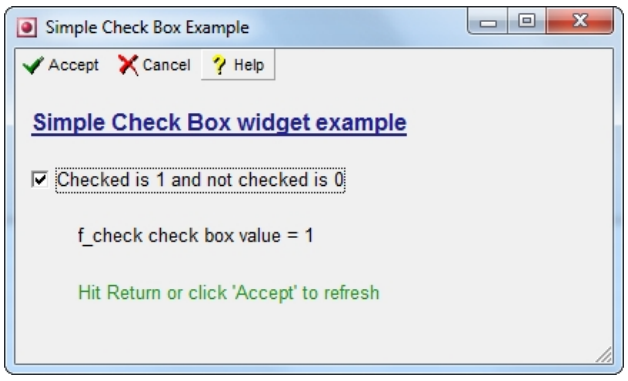
Location:

Project: forms
Program: fm_field_widget_check1 ...~2 , ...~simple and ...~event

Example .per Form Code:

```
DATABASE formonly

SCREEN
{
 Simple\g \gCheck\g \gBox\g \gwidget\g \gexample

 [f_check                          ]
}


ATTRIBUTES
f_check=formonly.f_check,
             config="1 0 {Checked is 1 and not checked is 0}",
             widget="check";

INSTRUCTIONS

DELIMITERS "[]"
```

Example screenshot



Example 2:

```
# Define a checkbox with the label ‘receive mail’.
#If the check box is ticked, the value 1 should be written
#into the record variable contact.rec_mail, otherwise ‘0’
#should be written. The default value should be ‘0’.
f0=          contact.rec_mail,
       config="1 0 {Receive Mail}",
      widget="check",
      default=0;
```

Example 3:

```
# Define a checkbox with the label ‘Next Action’.
#If the check box is ticked, the key press F1 should be
#sent, when unticked, the key press F2 is sent
f0=          formonly.nextAction,
       config="F1 F2{Next Action}",
      widget="check",
      class=”key”;
```

Example 4:

```
# Define a checkbox with the label ‘Next Action’.
#If the check box is ticked, the action event ‘act_print_text’ should be
#sent, when unticked, the action event ‘act_print_html’ is raised.
f0=          formonly.nextAction,
       config="act_print_text act_print_html{Next Action}",
      widget="check",
      class=”key”;
```

Radio Buttons widget

Radio buttons allow you one method of producing a selection list group from which only one option can be chosen. You can have one single or ‘many’ radio button items in one radio button group but all items in the group will write to the same variable. To write to more than one variable, you need to create separate radio button groups.

In ASCII mode, the entire radio button group is shown as one ‘single’ normal field.

If the value returned by the radio button field is a null string or does not match any of the radio button values, all buttons are shown as ‘not’ selected.

Alternatively, you can use a radio button to trigger key events, instead of writing values to variables directly. This may be required if you want to execute some program logic straight after the user has select an item from the radio button group.

Attributes Syntax:

```
<tag> =    <field>,
           config= "<1st option value> {<1st option label>}
                   [<2nd option value> {<2nd option label>}...]",

           widget=”radio”;
```

Attribute Example

The following Attributes code sample creates a radio button group consisting of two radio buttons. The first radio button writes ‘0’ to the variable f_check and the second writes ‘1’.

```
ATTRIBUTES
f_radio=formonly.f_radio,
     config="0 {Radio Button Value 0} 1 {Radio Button Value 1}",
     widget="radio";
```

Properties

Radio buttons use similar options to those of check boxes. You need to set the widget attribute to RADIO. The config option is constructed with a space separated list within double speech marks (“). You will need to list a series of strings; the first item in the string is the value returned if the first radio button is selected, followed by the label for that radio button. The third item in the string is the value returned if the second radio button is selected; the fourth item is its label. This format is repeated for as many radio buttons as you need.

widget

The name of the radio button field widget is ‘radio’

```
widget = “radio”
```

config

For a normal (class=default) radio button group, the configuration string consists of value/label pairs. Each radio button to be displayed needs to have a value followed by the corresponding radio button label.

```
config="0 {1st Radio Button} 1 {2nd Radio Button} 2 {3rd Radio Button}"
```

If you do not set a default value for the INPUT, the check box will set its variable to a null string (none of the radio buttons will be shown selected).

When clicking on any radio button within its radio button group, the value in the field is changed, however any AFTER FIELD logic associated with the field will not be triggered unless the user attempts to leave the field either by pressing ‘tab’ or by selecting another field with the mouse.

For a ‘KEY’ or ‘ACTION’ check box (class=key), the configuration string consists of the event to be fired when the corresponding radio button is selected and its corresponding radio button label text.

```
config="F1 {Goto 1} F2 {Goto 2} F3 {Goto 3} F4 {Goto 4}"
```

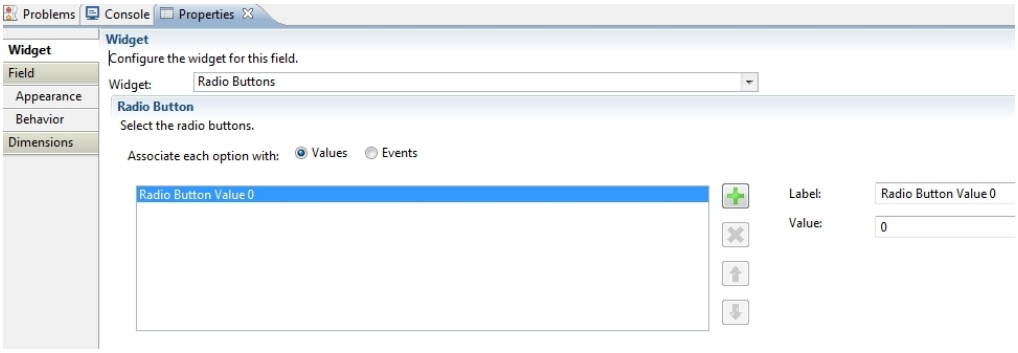
Accelerator keys

With the Radio widget there is also the optional functionality of including an accelerator key for each label. This will allow the user a keyboard shortcut to the chosen option by pressing Alt+Key. The shortcut key has to be a character in the label and will be underlined. To enable this functionality, place an ampersand before the required character in the label definition in the Config section. For example:

```
config=”0 {&Zero} 1 {&One} 2 {&Two} 3 {Thr&ee}”,
widget = “radio”;
```

This would underline Z, O, T and e respectively for each label.

Example properties using the graphical form editor



How to Create Radio Buttons - Example

The following code creates a radio button group consisting of two radio buttons. Pressing the Accept key will read the value of the radio button variable.

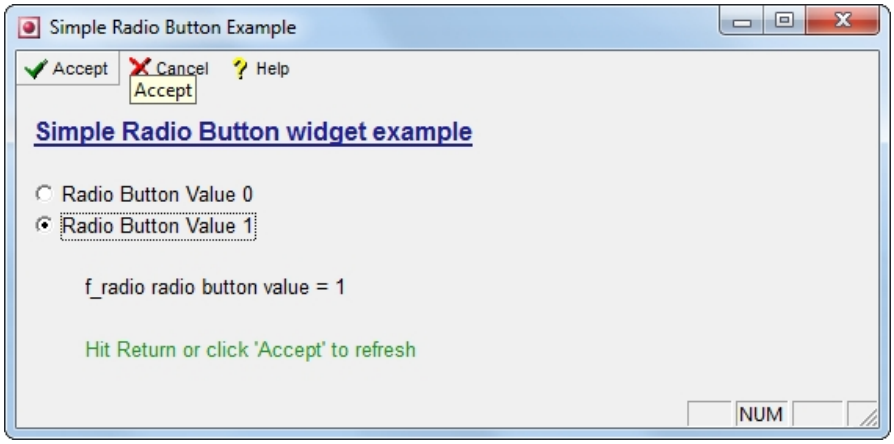
Location:

Project: forms
Program: fm_field_widget_radio1 ...~2 , ...~simple and ...~event

Example .per Form Code:

```
DATABASE formonly

SCREEN
{
 Simple\g \gRadio\g \gButton\g \gwidget\g \gexample

 [f_radio              ]

}
ATTRIBUTES
f_radio=formonly.f_radio,
      config="0 {Radio Button Value 0} 1 {Radio Button Value 1}",
      widget="radio";

INSTRUCTIONS
DELIMITERS "[]"
```

Example screenshot



Invoking a Key Code or Action Event (class = key)

As with check boxes, it is possible to send a single key or action event from a radio button instead of a value.

If you want the program to execute a corresponding program block after a particular check box or radio button is selected, the normal before and after field events are not sufficient. The before field event can't be used because it will be executed when the field gains focus (that is, tabbed into), the after field clause only executes a program block after you leave the field (tabbing out or clicking on a different field). Using the key class model, it sends a corresponding key press which depends on the menu option that the user chooses.

This is done by declaring the class option as "key" in the attribute section of the form file.

You can also select or deselect radio buttons that use the key class within 4GL programs. The code sample below shows the lines needed to firstly enable and secondly to disable the radio button called myButton:

```
DISPLAY "!" TO myButton
DISPLAY "*" TO myButton
```

	If you activate a default class radio button type outside an INPUT statement, the radio box will appear as selected, but you will not be able to use it.

Example 2:

```
# Define a radio button group of three items, ‘Phone now’
# with the value 1, ‘Phone later’
# 0 and ‘Phone never’ -1. The data should be written to
# the record variable contact.next_call of type integer.
f7=contact.next_call type INT,
       config="1 {Phone Now} 0 {Phone Later} -1 {Phone Never}",
       widget="radio",
       default="0";
```

Example 3:

```
# Identical to the previous example, but this time,
# we will send the key presses F1, F2 and F3.
f7=formonly.email,
       config="F1 {Send now} F2 {Send later} F3 {Delete}",
       widget="radio",
       class="key";
```

Example 4:

```
# Again identical to the previous example, but this time,
# we will raise the action events ‘act_sort_by_contact’,
‘act_sort_by_company’,’act_sort_by_date’.
f7=formonly.sort,
       config="act_sort_by_contact {Contact} act_sort_by_company {Company}
act_sort_by_date {Date}",
       widget="radio",
       class="key";
```

Function Button Field widget

The function button field object is a standard field which has a button displayed to its right. This button is a bitmap image that you can specify as a config value in the Attributes section of your form file. The button should be used to trigger a key or action event. A classic example would be a field where the e-mail address of a contact is stored. The button could be used to start the e-mail client to send an e-mail to this e-mail address. Another example would be a field to keep the website of a company. When the user clicks the button, the browser window opens with the corresponding URL.

Attributes Syntax:

```
<tag> =    <field>,
           config=<button image> <key or action to be invoked>,
           widget=”field_bmp”;
```

Attribute Example

The following Attributes code sample creates a function field (field_bmp widget). Next to the field, there will be a button with the image ‘icon10/search01.ico’. When the user clicks on it, the key press F6 will be invoked.

```
ATTRIBUTES
f_field_bmp=formonly.f_field_bmp,
      config="icon10/search01.ico F6",
      widget="field_bmp";
```

Properties

The function button field object is a combination of a normal field and a bmp widget. For this reason, it uses similar options to those of bmp widget; the only difference is that the image is a static property.

widget

The name of the function field field_bmp widget is ‘field_bmp’

```
widget = “field_bmp”
```

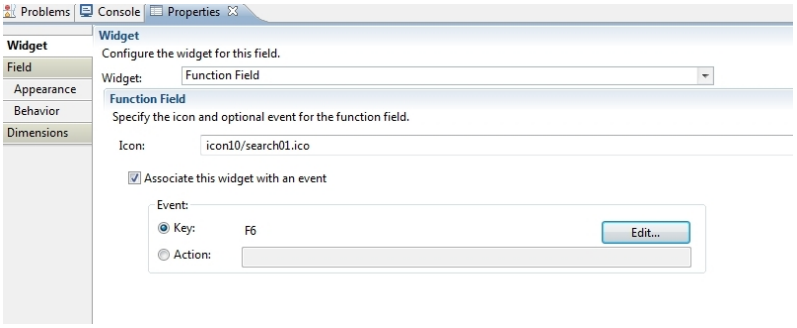
config

The config string keeps the image which is displayed in the button and the key/action event. If you don’t specify an image, the GUI clients will use a default image. The image file may be located on the server, or in a directory local to the client. But we strongly recommend keeping and managing all client resources on the server. Lycia supports all major graphic formats (ico, bmp, jpg, tif, gif and png) but will not scale or stretch the image to keep its quality. It is the responsibility of the developer to choose an image with the correct size. The button will never be deeper than the field and will scale proportionally to the applied font. If you use a default font height of 10, the image should not be larger than 10 pixels in height.

```
config=<image> <key/action>
config="my_image.bmp F12"
```

	When entering data into a function field and after clicking on the button, the field value is not yet written to the variable. To overcome this, use the function fgl_dialog_update_data(). Like any other field or widget, the new data will remain in the field buffer until the function fgl_dialog_update_data() is called, or the user navigates to the next field. AFTER FIELD logic will only be executed if the user attempts to leave the field as in navigating to another field.

Example properties using the graphical form editor



How to Create a Function Field - Example

The following code creates a function button field with a customized icon for the button.

Location:

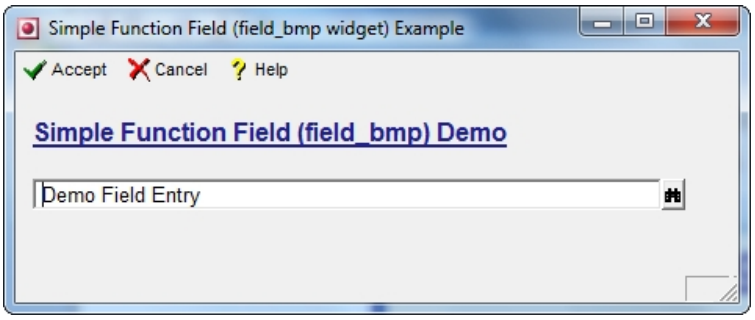
Project: forms
Program: fm_field_widget_field_bmp1 ...~2 and ...~simple

Example .per Form Code:

```
DATABASE formonly

SCREEN
{
 Simple\g \gFunction\g \gField\g \g(field_bmp)\g \gDemo
```

```
 [f_field_bmp                          ]
}
ATTRIBUTES
f_field_bmp=formonly.f_field_bmp,
      config="icon10/search01.ico F6",
      widget="field_bmp";

INSTRUCTIONS
DELIMITERS "[]"
```

Example screenshot



Example 2:

```
# Define a button field which sends the key press F5. The button
# image is a file called zoom.ico which is located in the
# applications sub-directory images and is thereby a server resource.
# The field is linked to the variable cust_country.
f10=          formonly.cust_country,
       config="images/zoom.ico F5",
      widget="field_bmp";
```

Example 3:

```
# Define a button field which sends the action event ‘action_print’.
# The button image is a file called print.ico which is located in the
# applications sub-directory icon10 and is thereby a server resource.
# The field is linked to the variable print_file.
F5=           formonly.print_file,
       config="icon10/print.ico action_print",
      widget="field_bmp";
```

Combo Box Field widget

A combo box field displays itself as a field with a down arrow button at the right hand end. Clicking the down arrow displays a drop down list of values from which users can choose. The list can be populated statically within the form definition using the 'include' attribute. There is also the option to allow the user to enter data into the field which do not exist in the list. At runtime, the combo list values can be changed (removed, inserted, searched, etc...) by using the relevant fgl_list_xxx() functions. The user can select list entries either by using the mouse or by navigating through the list using the keyboard (cursor keys or alpha numerical keys to search for entries).

Attributes Syntax:

```
<tag> =    <field>,
           widget=”combo”,
           class=”combo”, (Optional)
           include=(list item 1, list item2, list item n);
```

Attribute Example

The following Attributes code sample creates a combo box field containing all the days of the week. With the class set to 'combo' the user will also be able to enter their own data into the field.

```
ATTRIBUTES
f_combo=formonly.f_combo,
      widget="combo",
      class="combo",
      include=("Monday",
             "Tuesday",
             "Wednesday",
             "Thursday",
             "Friday",
             "Saturday",
             "Sunday");
```

Properties

Combo List Boxes require setting the widget attribute to 'combo'. The list is produced with the include property which is a comma separated list. The class property (class = “combo”) is optional and is used to allow the user to enter data (freely) which are not in the include list. In this case, the server will no longer validate the field data with the include list.

widget

The name of the combo list field widget is 'combo'

```
widget = “combo”
```

class

For ‘selection only’ combo list boxes you do not need to specify this attribute. If the user should be allowed to select from the list AND enter data freely into the field, set the class to combo.

```
class="combo"
```

include

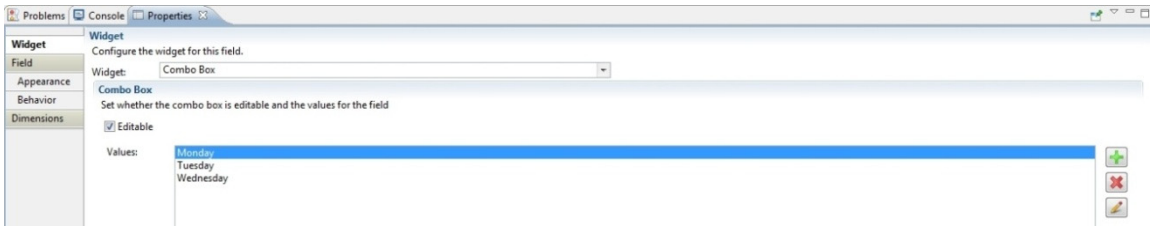
The include property (just like for any other 4GL field) is used to define a list of valid entries. In a graphical environment, these items are displayed in the combo list.

```
include = (“Mr.”,”Mrs.”,”Dr.”)
```

For formonly fields, always specify the data TYPE property for the field; otherwise a conversion error type may occur from the combo list data type to the field data.

When clicking on any item within the list, the value in the field is changed, however any AFTER FIELD logic associated with the field will not be triggered unless the user attempts to leave the field either by pressing ‘tab’ or by selecting another field with the mouse.

Example properties using the graphical form editor



How to Create a Combo List Box - Example

The following code creates a combo list box with a list of week days. Pressing the Accept key will read the value of the radio button variable.

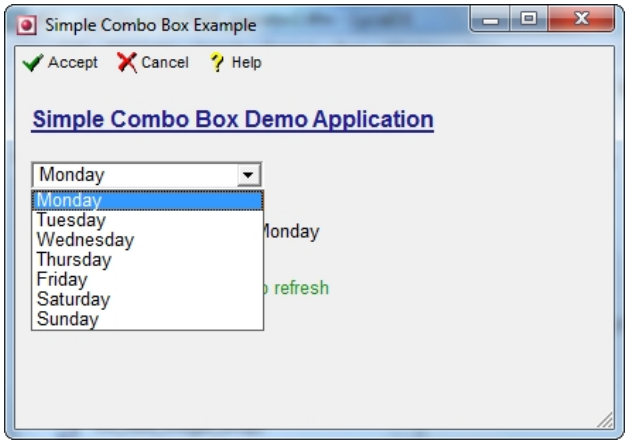
Location:

Project: forms
Program: fm_field_widget_combo1 ...~2 , ...~simple and ...~event

Example .per Form Code:

```
DATABASE formonly

SCREEN
{
 Simple\g \gCombo\g \gBox\g \gDemo\g \gApplication

 [f_combo       ]
}


ATTRIBUTES
f_combo=formonly.f_combo,
      widget="combo",
      include=("Monday",
             "Tuesday",
             "Wednesday",
             "Thursday",
             "Friday",
             "Saturday",
             "Sunday");

INSTRUCTIONS

DELIMITERS "[]"
```

Example screenshot



	Normal fields which have an INCLUDE attribute will by default be rendered as a combo box. This behaviour can be turned off using the script resource nocombo.

Combo List Box Functions

Please refer to the functions guide for a complete list of fgl_list_xxx() functions to manage and manipulate combo box lists at runtime.

Calendar widget

The calendar widget is displayed just like a function button field. Clicking on the button opens a calendar window, allowing the user to select the required date graphically using the mouse.

The selected date is then written back to the field. The date can also be entered manually using the keyboard.

The calendar widget field displays the date in the same format as specified in either the Environment attribute DBDATE or the fglprofile setting system.environment.dbdate = <date format>.

Example:

```
system.environment.dbdate = "dmy4-"
```

Attributes Syntax:

```
<tag> =    <field>,
           widget=”calendar”
```

Attribute Example

The following Attributes code sample creates a date field widget. Unlike the other widgets, the calendar field has no config string.

```
ATTRIBUTES
f0=   formonly.ca_date,
      widget="calendar";
```

Properties

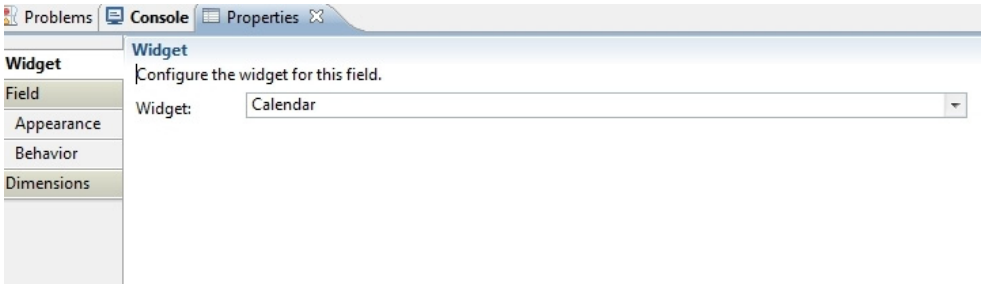
The developer only needs to specify the widget type to create a calendar field (date picker) widget.

widget

The name of the calendar widget is ‘calendar’

```
widget = “calendar”
```

Example properties using the graphical form editor



How to Create a Calendar (Date Picker) widget - Example

The following code creates a date widget.

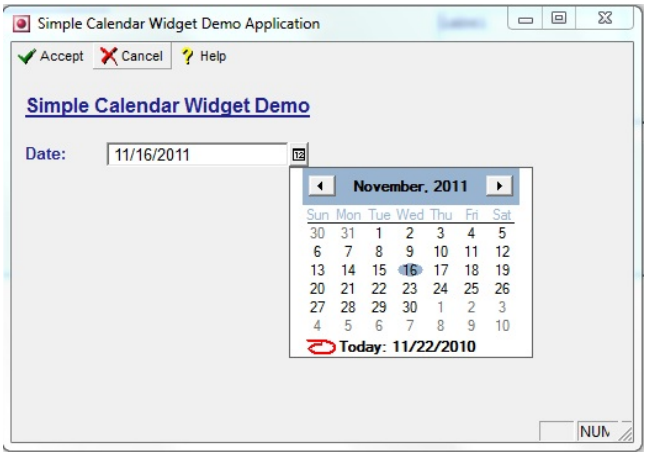
Location:

Project: forms
Program: fm_field_widget_calendar_simple

Example .per Form Code:

```
DATABASE formonly

SCREEN
{
 Simple\g \gCalendar\g \gWidget\g \gDemo

 Date: [f0            ]
}

ATTRIBUTES
f0=formonly.ca_date,
     widget="calendar";
INSTRUCTIONS
DELIMITERS "[]"
```

Example screenshot


Simple Calendar Widget Demo Application
Accept
Cancel
Help
Simple Calendar Widget Demo
Date:
11/16/2011
November, 2011
Sun Mon Tue Wed Thu Fri Sat
30 31 1 2 3 4 5
6 7 8 9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 1 2 3
4 5 6 7 8 9 10
Today: 11/22/2010
NUM

File & URL Browser widget

The file browser widget embeds a fully functional browser into the .per form. LyciaDesktop uses the Internet Explorer browser, and LyciaWeb uses the browser in which it is opened also as the embedded browser.

The file browser will attempt to render the source specified in its field value. You can use the browser to view HTML files, images, or any other files that it supports. The file browser is also useful to merge 4GL applications with Web-based applications.

Because the URL is taken from the field value, you can change it dynamically at any time using 4GL statements such as DISPLAY TO <field>.

The windows client will use an embedded version of the locally installed MS-IE. The LyciaWeb client will use any browser it is opened in. Querix is not responsible for the functionality or behaviour of these browsers.

Attributes Syntax:

```
<tag> =    <field>,
           Config=<url or file system path>
           widget=”browser”
```

Attribute Example

The following Attributes code sample creates a browser panel widget.

```
ATTRIBUTES
br_url=     formonly.br_url,
            config="www.querix.com",
            widget="browser";
```

Properties

The browser widget has two required attributes: the widget type ‘browser’ and the URL to be used for the initial form display.

widget

The name of the browser widget is ‘browser’

```
widget = “browser”
```

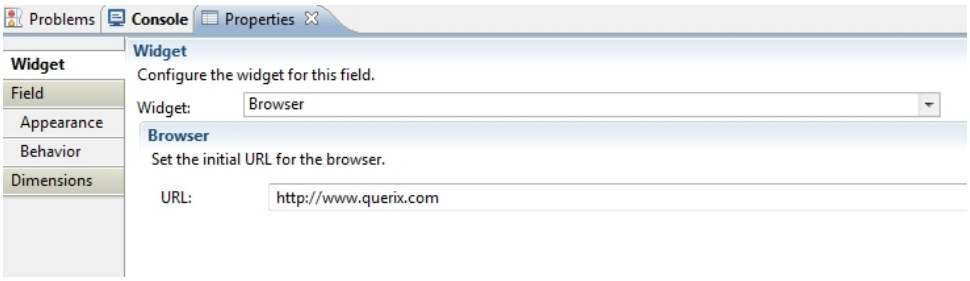
config

```
config=<url or file system path>
```

config examples:

```
config=”www.querix.com”
config=”c:/”
config=”c:/images/map.jpg”
```

Example properties using the graphical form editor



How to Create a Browser Panel widget - Example

The following code creates a browser widget.

Location:

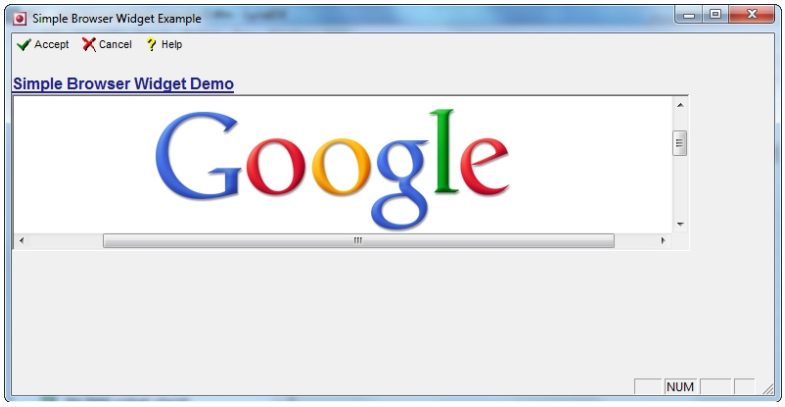
Project: forms
Program: fm_field_widget_browser1 and fm_field_widget_browser_simple

Example .per Form Code:

```
DATABASE formonly

SCREEN
{
Simple\g \gBrowser\g \gWidget\g \gDemo
[br_url                                                    ]
[br_url                                                    ]
[br_url                                                    ]
[br_url                                                    ]
[br_url                                                    ]
[br_url                                                    ]
[br_url                                                    ]
[br_url                                                    ]
```

```
}


ATTRIBUTES
br_url=formonly.br_url,
      config="www.querix.com",
      widget="browser";

INSTRUCTIONS

DELIMITERS "[]"
```

Example screenshot



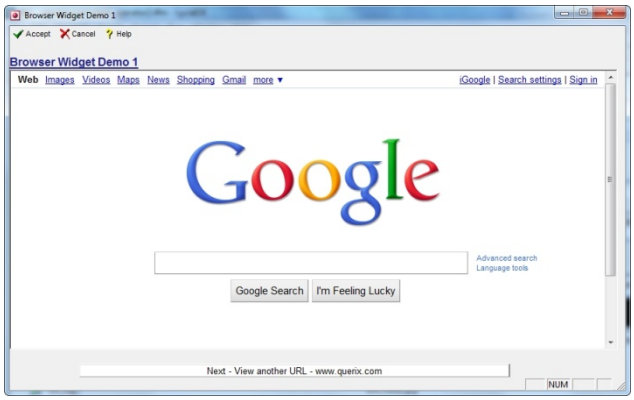
Browser Example 2

Example .per Form Code:

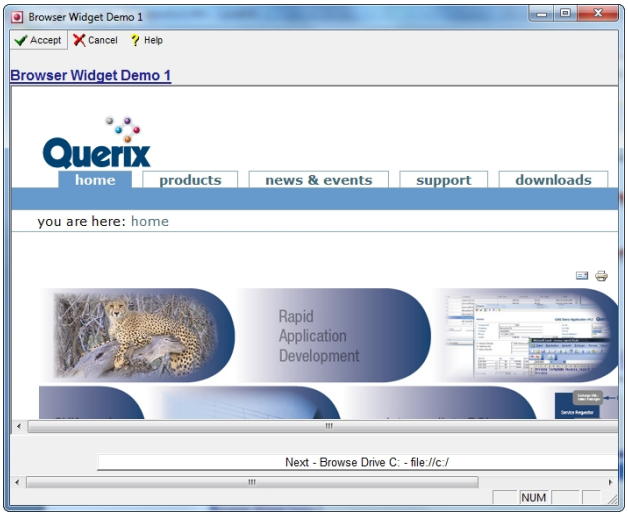
```
DATABASE formonly

SCREEN
{
Browser\g \gWidget\g \gDemo\g \g1
[br_url                                                    ]
[br_url                                                    ]
[br_url                                                    ]
[br_url                                                    ]
[br_url                                                    ]
[br_url                                                    ]
[br_url                                                    ]
[br_url                                                    ]
[br_url                                                    ]
[br_url                                                ]
[br_url                                                ]
[br_url                                                ]
[br_url                                                ]
[br_url                                                ]
```

```
[br_url                                                    ]
[br_url                                                    ]
[br_url                                                    ]
[br_url                                                    ]
[br_url                                                    ]
[br_url                                                    ]

         [bt_next                                 ]

}


ATTRIBUTES

br_url=formonly.br_url,
       config="",
       widget="browser";

bt_next=formonly.bt_next,
       config="Accept {Next - View another URL ?????}  ",
       widget="button";

INSTRUCTIONS

DELIMITERS "[]"
```

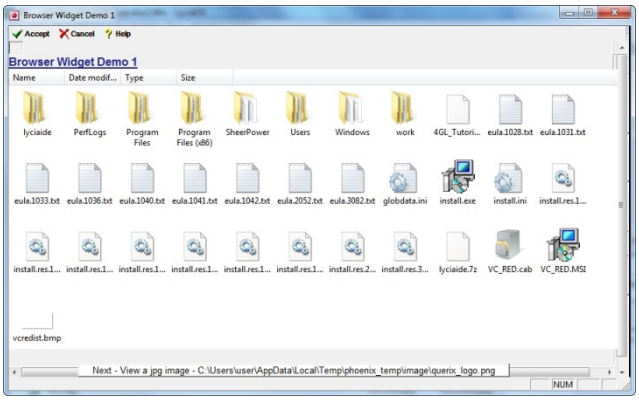
This example will first open the Google web site (www.google.com) in a browser.



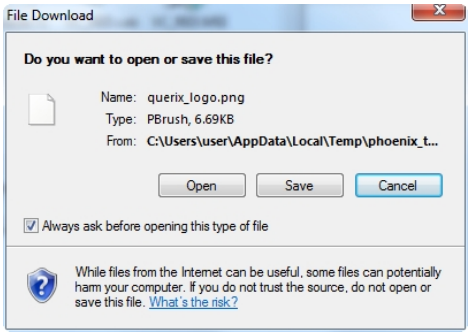
After clicking the 'Next' button, it will render another web site (www.querix.com) by displaying the URL using the statement: `DISPLAY "www.querix.com" TO br_url.`



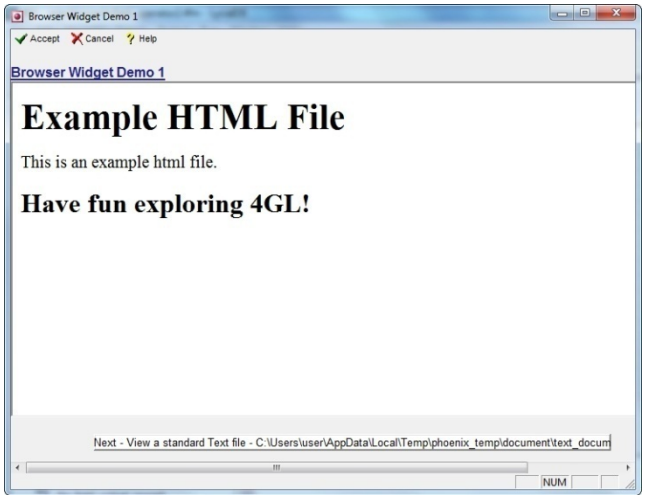
Next, it will display the file browser with the following statement: `DISPLAY "file://c:/" TO br_url`



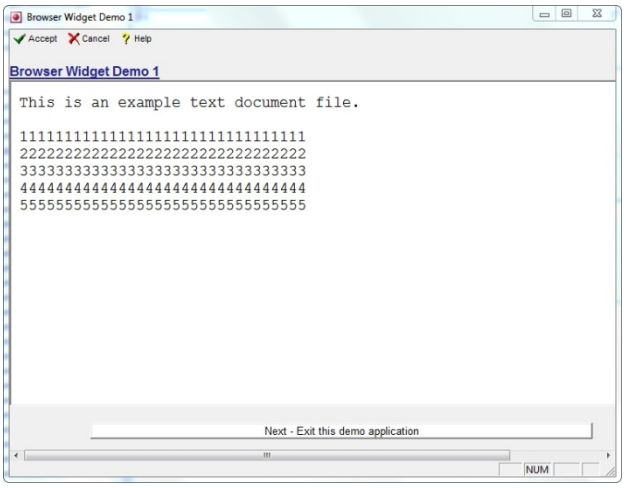
This will be followed by a dialog window prompting you to open or save the querix logo – a .png image.



Pressing the ‘Next’ button, you will be presented with an .html document (downloaded to the local client).



Followed by the contents of a text file (also downloaded to the local client): `DISPLAY "document/text_document.txt" TO br_url`



The final press on the ‘Next’ button will close the application.

Button widget

The button widget creates an input field, which is a button that the user can click. When clicked, the button sends a keypress or action event to the 4GL application. This behaves in exactly the same way as if the user had pressed the key directly.

Any text sent to the field using a DISPLAY TO statement has the effect of changing the label shown on the button.

The form buttons are active by default, if the current dialog includes the corresponding ON KEY/ON ACTION event. If the event is not specified, the button is disabled and greyed.

However, it is possible to manipulate buttons activation with the help of DISPLAY ... TO *button_field* statement passing key characters to the button fields:

- "!" - activate an inactive button.
- "*" - deactivate an active button.
- "?" - return the button to the automatic activation mode.

When a form button is activated or deactivated explicitly with the DISPLAY statement, the effect lasts until the form is closed or another DISPLAY statement is executed. Switching between dialogs with different sets of ON KEY/ON ACTION events specification does not influence the button set up by the DISPLAY statement.

In ASCII mode, buttons display as normal fields.

If the keypress associated with the button is not recognised as a hotkey or an action event, pressing the button may simply echo the key into the active input field.

Attributes Syntax:

```
<tag> =      <field>,
            config=<key or action> <label>,
            widget="button";
```

Attributes Example:

```
fb_1=        formonly.fb_1,
       config="F1 {My button label}",
       widget="button";
```

Properties:

widget

The name of the button widget is ‘button’.

```
widget = “button”
```

config

The configuration string consists of the key press followed by the button text. The values are separated by a space.

```
config="key  {button label}"
```

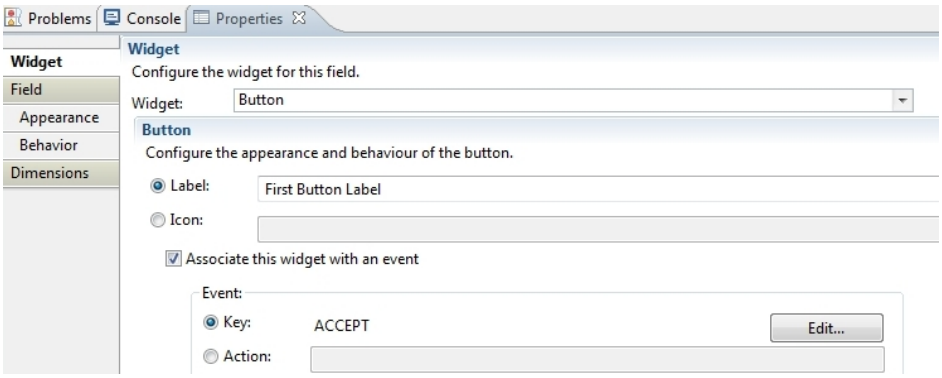
Accelerator keys

With the Button widget there is also the optional functionality of including an accelerator key for each button label. This will allow the user a keyboard shortcut to the chosen option by pressing Alt+Key. The shortcut key has to be a character in the label and will be underlined. To enable this functionality, place an ampersand before the required character in the label definition in the Config section. For example:

```
config=”F1 &First Button”,
widget = “button”;
```

This would underline F in the label “First Button”.

Example properties using the graphical form editor



INPUT on a button widget field

Because the button widget is a user control object, it makes no sense to include it in an input statement.

Dynamically update the button widget at runtime from 4GL

At runtime, you can toggle the state of the button between enabled and disabled and change the button label text.

Disabled and Enabled Buttons

Buttons are by default disabled. To enable them, send an “!” character to the field using the display statement; to disable them, display an asterisk ‘*’ character.

```
#Enable the button f_button_exit
DISPLAY “!” TO f_button_exit

#Disable the button f_button_exit
DISPLAY “*” TO f_button_exit
```

Changing the button label/text

To change the button text, use the display to statement:

```
#Change the label of the button f_button_exit to Exit
DISPLAY “Exit” TO f_button_exit
```

How to Create a Button – Example 1

The following code sample creates a form with a button. Clicking on it triggers an ‘Accept’ key event which will toggle the button labels.

Location:

Project: forms
Program: fm_control_widget_button1 ...~2 , and ...~simple

Example .per Form Code:

```
DATABASE formonly

SCREEN
{
Simple\g \gButton\g \gwidget\g \gexample

[f_button                  ]
}

ATTRIBUTES
f_button=formonly.f_button,
      config="Accept {First Button Label}",
```

```
        widget="button";

    INSTRUCTIONS
    DELIMITERS "[]"
```

Example screenshot

Example 2 :

```
# Define a form button fb_button which triggers the key event
# F1 and has got the label ‘Done’.

fb_done =    formonly.fb_done,
             config="F1 Done",
             widget="button";
```

Example 3:

```
# Define a form button fb which triggers the
# key event ‘Break’ and has the ‘two word’ text
# label ‘My Button’. Configuration attribute
# strings which have got multiple words need
# to be enclosed in curly braces.

Fb =         formonly.fb,
             config="Break {My Button}",
             widget="button";
```

Example 4:

```
# Define a form button fb which triggers the
# action event ‘act_save_data’ and has the ‘two word’ text
# label ‘Save Record’. Configuration attribute
# strings which have got multiple words need
# to be enclosed in curly braces. Additionally, the button label is blue and bold

fb=          formonly.fb,
       config="act_save_data {Save Record}",
```

```
widget="button" ,
color=bold Blue;
```

Image Button

You can also create a button with an image displayed in it instead of a string. The same button widget is used for this purpose in the graphical form editor, but in the test form the name of the image is different. A key or action event can be assigned to this button in the same way as to the button with the inscription which gets triggered if the user clicks the image button. This widget behaves identically to the button widget but instead of displaying a string, it displays an image. The image will be displayed with a 1:1 scale and the button size will depend on the size of the widget.

Any text sent to the field using a DISPLAY TO statement has the effect of changing the image shown on the image button. The bmp image button is displayed disabled by default. It can be enabled at runtime by sending an '!' character (DISPLAY "!" TO <field_name>). To disable it, send an asterisk '*' or an empty string "" to it (DISPLAY "*" TO <field_name>).

The size of the image bmp area will not be changed at runtime. If the bmp widget is initialised with a 100x100 pixel image, and another image with 200x200 pixels is displayed to it, it will still only display 100x100 pixels and the remaining area will be lost. If the image is smaller than the original, it will be displayed centred.

Like all Querix graphical components, the bmp button widget supports bmp, jpg, tif, ico, png and gif format.

In ASCII mode, bmp fields display as normal fields.

If the keypress associated with the button is not recognised as a hotkey, pressing the button may simply echo the key into the active input field. You can prevent this from happening by using Function keys (F1-F1024) or Action Events.

Attributes Syntax

<tag> = <field>, config=<image> <key or action>, widget="bmp";

Attributes Example

The following Attributes code sample creates an image button.

```
ATTRIBUTES
f_bmp=      formonly.f_bmp,
            config="image/querix_logo.png Accept",
```

```
widget="bmp";
```

Properties:

widget

The name of the bmp widget is ‘bmp’.

```
widget = “bmp”
```

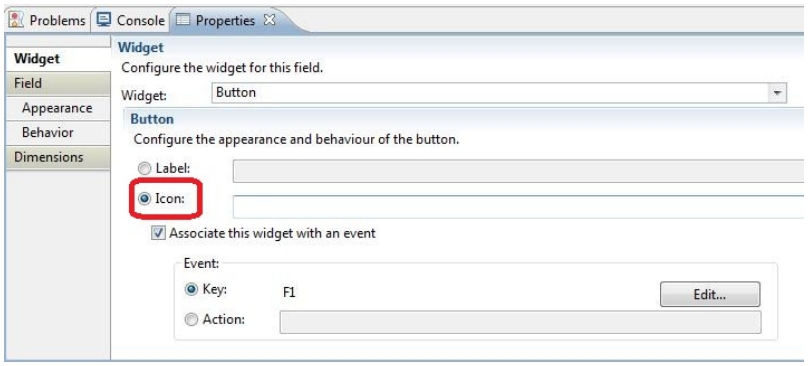
config

The configuration string consists of the image path, followed by the key/action event. The values are separated by a space.

```
config="<image path> <key or action>"
```

To make an ordinary button a bmp image button by means of the graphical form editor, it is necessary to select the ‘icon’ option instead of the ‘label’ option in the properties view, and then choose an appropriate image. The following figure illustrates this.

Example properties using the graphical form editor



INPUT on a bmp widget button field

The bmp widget is a combination of field and user controls. If the bmp widget field is included in the input statement, you query and set the image path and file name. After the image field’s contents have changed, the widget will render/display the newly defined image when possible.

Dynamically update/change the widget at runtime from 4GL

Disabled and Enabled Buttons

Buttons are disabled by default; to enable them, send an “!” character to the field using the display statement. Sending an asterisk ‘*’ character would disable it.

```
#Enable the button f_button_exit
```

```
DISPLAY “!” TO f_button_exit

#Disable the button f_button_exit
DISPLAY “*” TO f_button_exit
```

Changing the image at runtime

```
To change the image at runtime, use the display to statement
#Change the image to the relative path image/ok.ico
DISPLAY “image/ok.ico” TO f_bmp
```

How to Create a Bmp Button - Example

The following code sample creates a form with an image bmp button. Clicking on it triggers an ‘Accept’ key event which will toggle the images.

Location

Project: forms
Program: fm_control_widget_bmp1 ...~2, and ...~simple

Example .per Form Code:

```
Form Code:

DATABASE formonly

SCREEN
{

  Simple\g \gBmp\g \gwidget\g \gexample

 [f_bmp]









}


ATTRIBUTES
f_bmp=formonly.f_bmp,
      config="image/querix_logo.png Accept",
```

```
        widget="bmp";
    INSTRUCTIONS
    DELIMITERS "[]"
```

Example screenshot

In ASCII mode, bmp fields display as normal fields.

Image Widget

The image widget, as the name suggests, creates a static image area. The image path can be initialised in the form but changed at runtime. The image is *non-clickable* and will be displayed with a 1:1 scale.

Any text sent to the field using a DISPLAY TO statement has the effect of changing the image shown on the image button. The image button is displayed disabled by default. It can be enabled at runtime by sending an '!' character (DISPLAY "!" TO <field_name>). To disable it, send an asterisk '*' or an empty string "" to it (DISPLAY "*" TO <field_name>).

Like all Querix graphical components, the image widget supports bmp, jpg, tif, ico, png and gif format.

In ASCII mode, image fields display as normal fields.

Attributes Syntax

<tag> = <field>, config=<image_path>, widget="image";

Attributes Example

The following Attributes code sample creates an image button (non-clickable).

```
ATTRIBUTES
f_image=    formonly.f_image,
            config="image/querix_logo.png",
            widget="image";
```

Properties:

widget

The name of the image widget is 'image'.

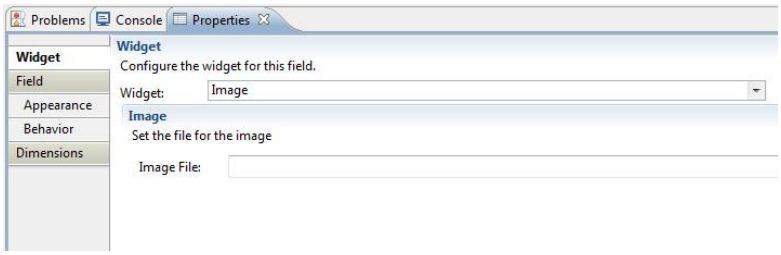
```
widget = "image"
```

config

The configuration string consists of the image path only.

```
config="<image path>"
```

Example properties using the graphical form editor



INPUT on a bmp widget button field

The image widget is a combination of field and user controls. If the image widget field is included in the input statement, you query and set the image path and file name. After the image field's contents have changed, the widget will render/display the newly defined image when possible.

Dynamically update/change the widget at runtime from 4GL

Disabled and Enabled Images

Images are enabled by default; to enable them after having been disabled, send an exclamation "!" character to the field using the display statement. Sending an asterisk '*' character would disable it.

```
#Enable the button f_image_exit
DISPLAY "!" TO f_image_exit

#Disable the button f_image_exit
DISPLAY "*" TO f_image_exit
```

Changing the image at runtime

```
To change the image at runtime, use the display to statement
#Change the image to the relative path image/ok.ico
DISPLAY "image/ok.ico" TO f_image
```

How to Create an image - Example

The following code sample creates a form with an image button.

Location

Project: form
Program: fm_control_widget_image1 ...~2, and ...~simple

Example .per Form Code:

```
Form Code:

DATABASE formonly

SCREEN
{

  Simple\g \gBmp\g \gwidget\g \gexample

 [f_bmp]









 [dl_sub_header1  ]
}


ATTRIBUTES
f_bmp=formonly.f_bmp,
             config="image/querix_logo.png",
             widget="image";
dl_sub_header1=formonly.dl_sub_header1,
             config="{Press Enter to exit}",
             widget="label";

INSTRUCTIONS

DELIMITERS "[]"
```

Example screenshot

In ASCII mode, image fields display as normal fields.

Hotlink widget

The hotlink widget creates a label that, when clicked on, will either perform a key event, an action event, open a web browser with a specified URL (can also be a file path), or open up a file explorer. The widget type as well as the class (type of event) is specified in the Form definition .per file, as are the key and label. Depending on the widget class, clicking on the label will perform the aforementioned actions.

Attributes Syntax

```
<tag> =      <field>,
             Config=<url or file system path> <label>
             widget=”hotlink”;
```

Attribute Example

The following Attributes code sample creates a hotlink widget launching an URL (www.querix.com) at form initialisation stage. We have also assigned the attributes underline and colour blue for the label.

```
   ATTRIBUTES
hl_example=formonly.hl_example,
             config="www.querix.com {Querix Web Site: www.querix.com}",
             widget="hotlink",
             color=Blue underline;
```

Properties

The hotlink widget has two required attributes: The widget type ‘hotlink’ and the config property consisting of the URL or file system path and the label to be displayed at the initial form display. They must be separated by a space and label names which consist of more than one word must be enclosed in curly braces {}.

widget

The name of the hotlink widget is ‘hotlink’

```
widget = “hotlink”
```

class

To launch an external browser, do not specify any class or specify the class as ‘default’.

```
class = “default”
```

To raise a ‘key’ press or an action, set the class to ‘key’’

```
class = “key”
```

config

```
config=”<url or file system path> <label>”
```

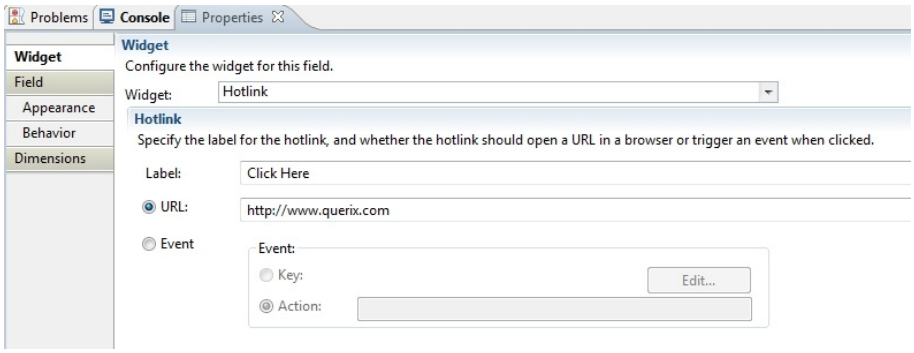
config examples relating to class default (class=”default”):

```
# to launch a local web browser and render the URL www.querix.com
config=”www.querix.com {Querix Home Page}”

# Browse the local file system from the drive c:\ - label is ‘My C Drive’
config=file://c:/ {My C Drive}

# Display the locally located image map.jpg – label ‘My Photo’
config=”c:/images/my_pic.jpg {My Photo}”
```

config examples relating to key and action events (class=”key”):

```
# Raise the key event F12 – label Exit Application
Config=”F12 {Exit Application}

# Raise the action ‘act_store_data’ – label ‘Save Record’
Config=”act_store_data {Save Record}
```

Example properties using the graphical form editor



INPUT on a hotlink widget field

Because the hotlink widget is a control widget object, it makes no sense to include it in an input statement.

Dynamically update/change the widget at runtime from 4GL

You can change the URL/Key or the URL/Key and the label at runtime by using the DISPLAY <> TO statement. If you only send one word to the field, it will replace the URL/key/event. Sending two words to the file will replace the entire config string URL/key/event and Label

Change the URL or Key at runtime (hotlink field is named hl_control)

Syntax:

DISPLAY "<URL, key or action>" TO <field name>

Example 1:

```
DISPLAY "www.google.com" TO hl_control
```

Example 2:

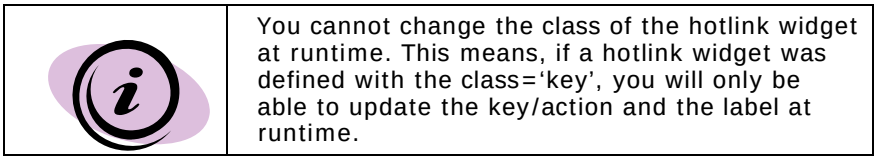
```
DISPLAY "www.google.com {search page}" TO hl_control
```

Change the URL/key and the label

```
DISPLAY "<url, key or action> {Label}"
```

Example:

```
DISPLAY "F12 {Exit Application}
```



You cannot change the class of the hotlink widget at runtime. This means, if a hotlink widget was defined with the class='key', you will only be able to update the key/action and the label at runtime.

How to Create a Hotlink Widget - Example 1

The following code creates a simple hotlink widget demo application which on click, launches the local browser and requests/renders the URL www.querix.com.

Location

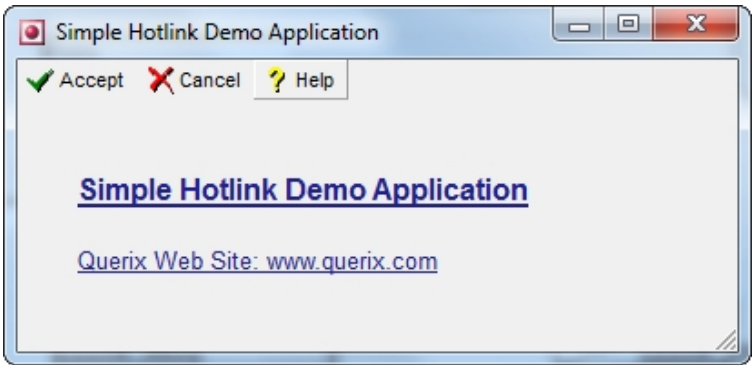
Project: form
Program: fm_control_widget_button1 ...~2 , and ...~simple

Example .per Form Code:

```
DATABASE formonly
```

```
SCREEN
{

   Simple\g \gHotlink\g \gDemo\g \gApplication

   [hl_example                    ]
}


ATTRIBUTES
hl_example=formonly.hl_example,
             config="www.querix.com {Querix Web Site: www.querix.com}",
             widget="hotlink",
             class="default",
             color=Blue underline;

INSTRUCTIONS

DELIMITERS "[]"
```

Example Screenshot

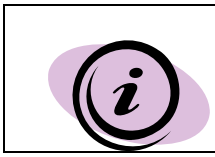


The following properties are common across all classes:

```
DISPLAY “!” TO my_hotlink
DISPLAY “ ” TO my_hotlink
```

Passing ‘!’ to a hotlink will disable that hotlink, rendering it effectively a label only and passing a space to the hotlink will be ignored, leaving the hotlink unchanged.

As well as the functionality above, the hotlink widget supports the form ATTRIBUTES color, underline, bold, and reverse.



If the keypress associated with the hotlink is not recognised as a hotkey, pressing the hotlink may simply echo the key into the active input field.

CHAPTER 4

Form Field Attributes

ACTION/ ACTIONS

Action Events

Querix4GL has advanced functionality that allows for a more useful method of assigning actions to buttons than keys; replacing the current 'key' syntax with 'action' and allowing for sensible, user-defined words instead of key presses.

This functionality ranges across 4GL input blocks, form definitions, form field attributes, and the fglprofile. An example of an 'action' equivalent of the key syntax for each section follows:

4GL Input Block

In a 4GL input block, if you wanted a menu to exit every time the F9 key was pressed, rather than having to remember the F9 key it might be easier to remember a word – 'close' for example. So instead of having:

```
ON KEY (F9)
```

it would be more sensible to write:

```
ON ACTION ("close")
```

Fglprofile

The following is an example of the fglprofile key label definition using the 'action' event to label the buttons instead. So,

```
key.F1.text = "Print this text"
```

Can be written to make more sense in the following way:

```
action.print.text = "Print this text"
```

Form Field Definition

Another example, in the form field definition, where an action label, icon image and positional order are set follows:

```
Print = formonly.Print,
Action Print = “Label text for Print” (“icon.jpg”) ORDER 3
```

This will, as with defining Key labels, set the label text for the print action event to ‘Label text for Print”, will set the picture at “icon.jpg” as the icon image, and will give the button a positional order number of 3.

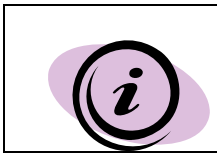
Global Form Definition

The following defines keys in the global Keys Form:

```
Screen
{
…
}
ATTRIBUTES
Tag1 = formonly.Tag1,
      Config = “F9 {Close Button}”
      Widget = “button”

KEYS
F9 = “Global label for F9” (“icon.jpg”) ORDER 3
```

The above can be written to incorporate the action event as follows:

```
Screen
{
…
}
ATTRIBUTES
Tag1 = formonly.Tag1,
      Config = “Close {Close button}”
      Widget = “button”

ACTION
Close = “Global label for Close” (“icon.jpg”) ORDER 3
```



This attribute is used to control the graphical toolbar and can for that reason, only be processed by graphical clients. Text mode clients ignore this attribute.

Toolbar Buttons

Toolbar buttons that have been defined in COMMAND KEY, ON ACTION or ON KEY statements are displayed on the top of a window in the form of a toolbar. The developer has the options to simply assign a key or action event with a text label, with an icon image, or with both. The order position number can also be specified optionally.

The buttons can be displayed permanently (Static True) or 'just in time' (Static False) when they are activated by a 4GL instruction.

The following order of precedence is used when editing hot-key button labels:

1. The key/ action attributes of a .per file
2. The fgl_dialog_setkeylabel()/ fgl_dialog_setactionlabel() functions
3. The KEYS/ ACTIONS section in a .per file
4. The fgl_setkeylabel/ fgl_setactionlabel() functions
5. The fglprofile file

The following examples use the action equivalents for each method.

Defining Hot-Key Labels in fglprofile

You can define a label for each hot-key in the $QUERIXDIR/etc/fglprofile configuration file. The resource you use to do this is:

```
action."action".text = "label"
```

You can set the order of appearance of the hot-keys the right-hand frame of the application window, in fglprofile, with the following resource:

```
action."action_name".order
```

Defining Hot-Key Labels in the Form File

You can also define hot-key labels by editing the ACTIONS section of the form specification file, using syntax like that shown here:

```
ACTIONS
action = "new_action_label"
```

Setting the ACTION Field Attribute

You can use the ACTION field attribute in the form specification file to change the label on a hot-key only when the cursor is in the corresponding field. To do this use syntax like that shown here:

```
ATTRIBUTES
f001 – customer.customer_num,
   ACTION Search = “SEARCH”,
   ACTION Clear = “CLEAR”,
   REVERSE
```

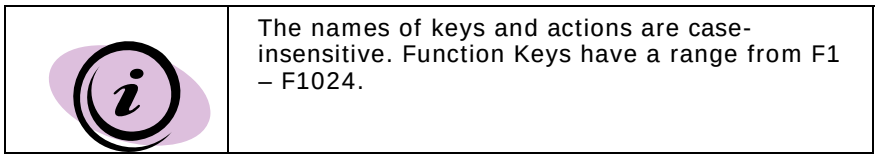
This code sample would cause the labels of the Search and Clear hot-keys to be changed only when the cursor is in the field that corresponds to the f001 tag.

Using 4GL Functions

You can use calls on 4GL functions in source code modules, to change the labels on hot keys. This can be done both in functions that execute within a dialog with the user (such as INPUT, INPUT ARRAY, and CONSTRUCT), and those that are not specific to an active user dialog.

To find the current text of a hot-key label, use the fgl_dialog_getactionlabel() function. To set a hot-key label for a specific user dialog, use the fgl_dialog_setactionlabel(). All of these functions are described in more detail in the Built-In Functions manual.

To find or set a hot-key label on a global, rather than dialog specific, basis, use the fgl_getactionlabel () and fgl_setactionlabel() functions.



The names of keys and actions are case-insensitive. Function Keys have a range from F1 – F1024.

Example Application:

Location:

Project: forms
Program: project fm_attribute_action

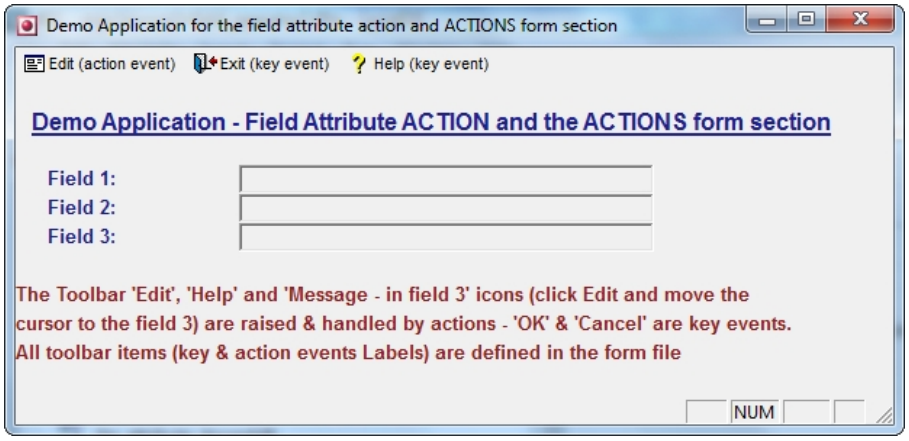
Form .per Source:

```
DATABASE formonly

SCREEN
{
 [dl_header                                          ]

  [dl_field1] [f1                        ]
  [dl_field2] [f2                        ]
  [dl_field3] [f3                        ]
```

```
[dl_info                                                    ]
[dl_info2                                                   ]
[dl_info3                                                   ]
}


ATTRIBUTES

f1=formonly.f1 type VARCHAR,
     KEY F13="Message Field 1 (in field 1 key event)" ORDER 11;
f2=formonly.f2 type VARCHAR,
     KEY F14="Message Field 2 (in field 2 key event)" ("icon16/info01.ico")
ORDER 12;
f3=formonly.f3 type SMALLINT;
dl_field1=formonly.dl_field1,
     config="{Field 1:}",
     widget="label";
dl_field2=formonly.dl_field2,
     config="{Field 2:}",
     widget="label";
dl_field3=formonly.dl_field3,
     config="{Field 3:}",
     widget="label";
dl_header=formonly.dl_header,
     config="{Demo Application - Field Attribute ACTION and the ACTIONS
form section}",
     widget="label",
     color=bold Blue;
dl_info=formonly.dl_info,
     config="{The Toolbar 'Edit', 'Help' and 'Message - in field 3' icons
(click Edit and move the}",
     widget="label",
     color=bold RED;
dl_info2=formonly.dl_info2,
     config="{cursor to the field 3) are raised & handled by actions - 'OK'
& 'Cancel' are key events.}",
     widget="label",
     color=bold RED;
dl_info3=formonly.dl_info3,
     config="{All toolbar items (key & action events Labels) are defined in
the form file}",
     widget="label",
     color=bold RED;

INSTRUCTIONS

DELIMITERS "[]"
```

```
KEYS
F2="OK (key event)" ORDER 1
F3="Cancel (key event)" ORDER 2
F12="Exit (key event)" ("icon16/exit01.ico") ORDER 99


ACTIONS
act_edit="Edit (action event)" ("icon16/record_edit01.ico") ORDER 5
HELP="Help (action event)" ("icon16/help01.ico") ORDER 101
```

Example Screen:



AUTONEXT

The AUTONEXT field attribute can move the cursor automatically to the next field when the user has typed enough characters to fill a field (depending on the client). The user can indicate that data entry is complete by pressing the Accept key in any field.

```
ATTRIBUTES
f0=formonly.f0,
    autonext;
f1=formonly.f1,
   autonext;
```

Difference between text and GUI

Text Mode Client:

The client navigates to the next field when the cursor has reached the last character of the visual field.

GUI Mode Client:

The auto next field navigation takes place when the cursor has reached the last character/element of the underlying variable.

Usage

You specify the order of fields in each INPUT or INPUT ARRAY statement. If the most recent OPTIONS statement specifies INPUT WRAP, the *next* field after the last field is the first field.

AUTONEXT is particularly useful with character fields in which the input data is of a standard length, such as numeric postal codes. It is also useful if a character field has a length of 1 because only one keystroke is required to enter data and move to the next field.

If data values entered in the field do not meet requirements of other field attributes like INCLUDE or PICTURE, the cursor does *not* automatically move to the next field but remains in the current field, with an error message.

The demonstration application uses the **invoice** form to enter payment information for a new invoice. The following excerpt from the ATTRIBUTES section of the **invoice** form uses the AUTONEXT attribute:

```
f35=invoice.pay_desc;
f34=pay_methods.pay_name,widget="combo",autonext,include=("VISA  ","Credit","Debit
");
f33=invoice.invoice_po;
```

When an item is selected in the pay_methods field (thus filling the field), the cursor moves automatically to the beginning of the next screen field (the **invoice.pay_desc** field).

Example application

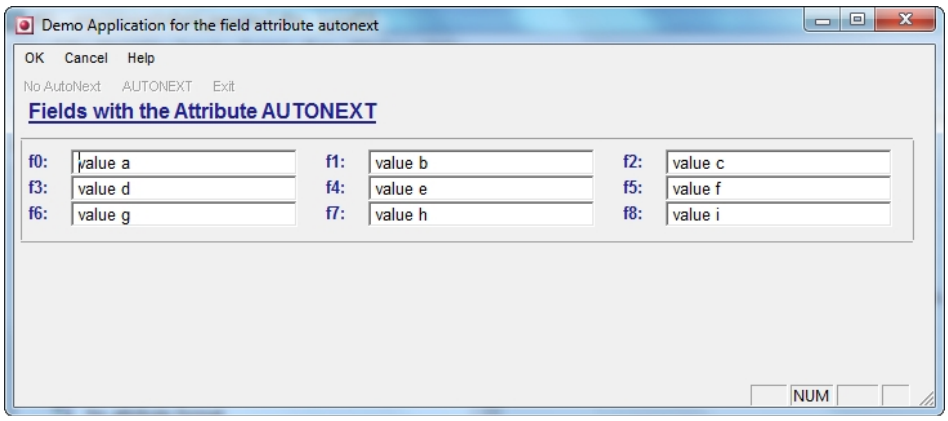
Location:

Project: forms
Program: project fm_attribute_autonext

Form .per Source

```
DATABASE formonly

SCREEN
{
 [dl_header                          ]
\gp-------------------------------------------------------------q\g
\g|\gf0:[f0             ] f1:[f1             ] f2:[f2             ]\g|\g
\g|\gf3:[f3             ] f4:[f4             ] f5:[f5             ]\g|\g
\g|\gf6:[f6             ] f7:[f7             ] f8:[f8             ]\g|\g
\gb-------------------------------------------------------------d\g
}


ATTRIBUTES
f0=formonly.f0,
     autonext;
f1=formonly.f1,
     autonext;
f2=formonly.f2,
     autonext;
f3=formonly.f3,
     autonext;
f4=formonly.f4,
     autonext;
f5=formonly.f5,
     autonext;
f6=formonly.f6,
     autonext;
f7=formonly.f7,
     autonext;
f8=formonly.f8,
     autonext;
dl_header=formonly.dl_header,
     config="{Fields with the Attribute AUTONEXT}",
     widget="label";

INSTRUCTIONS

DELIMITERS "[]"
```

Example Screen



CENTER

The CENTER attribute aligns the field contents centrally regardless of the default formatting characteristics of the data type being used.

For example, by default, 4GL left justifies all CHAR/VARCHAR fields, and right justifies all numeric datatypes. The example program which follows shows the right attribute, left attribute, CENTER attribute and the default alignment of the data types.

Usage:

The contents of the field ‘cust_name’ would be aligned centrally.

```
Cust_name=    customer.cust_name,
              CENTER;
```

Example Program

Location:

Project: forms
Program: project fm_attribute_right, fm_attribute_left and fm_attribute_center

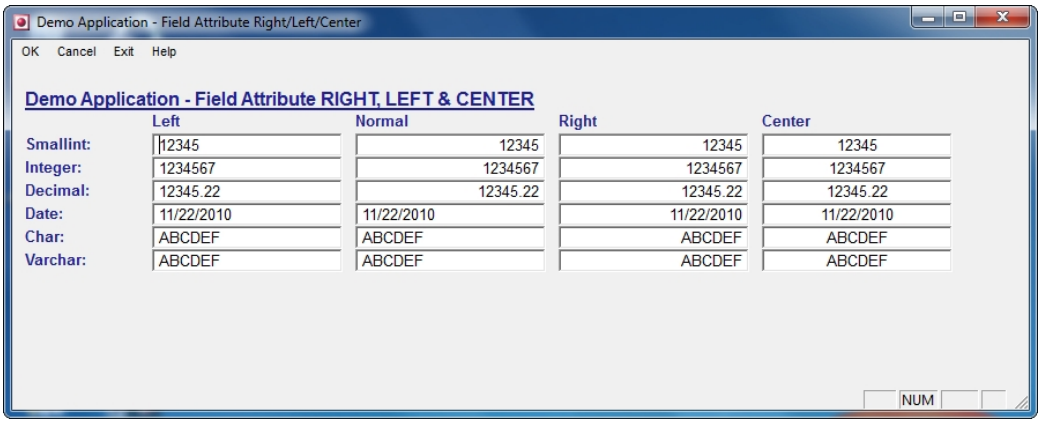
Form .per Source code:

```
DATABASE formonly

SCREEN
{
 [dl_header                                        ]
           [dl_sub_header1][dl_sub_header2][dl_sub_header3][dl_sub_header4]
 Smallint: [f0l           ][f0            ][f0r           ][f0c           ]
 Integer:  [f1l           ][f1            ][f1r           ][f1c           ]
 Decimal:  [f2l           ][f2            ][f2r           ][f2c           ]
 Date:     [f3l           ][f3            ][f3r           ][f3c           ]
 Char:     [f4l           ][f4            ][f4r           ][f4c           ]
 Varchar:  [f5l           ][f5            ][f5r           ][f5c           ]

[dl_guide                                                                 ]
[dl_guide                                                                 ]
}


ATTRIBUTES
f0r=formonly.f0r,
              right;
f1r=formonly.f1r,
```

```
                right;
f2r=formonly.f2r,
                right;
f3r=formonly.f3r,
                right;
f4r=formonly.f4r,
                right;
f5r=formonly.f5r,
                right;
f0l=formonly.f0l,
                left;
f1l=formonly.f1l,
                left;
f2l=formonly.f2l,
                left;
f3l=formonly.f3l,
                left;
f4l=formonly.f4l,
                left;
f5l=formonly.f5l,
                left;
f0c=formonly.f0c,
                center;
f1c=formonly.f1c,
                center;
f2c=formonly.f2c,
                center;
f3c=formonly.f3c,
                center;
f4c=formonly.f4c,
                center;
f5c=formonly.f5c,
                center;
f0=formonly.f0;
f1=formonly.f1;
f2=formonly.f2;
f3=formonly.f3;
f4=formonly.f4;
f5=formonly.f5;
dl_header=formonly.dl_header,
                config="{Demo Application - Field Attribute RIGHT, LEFT & CENTER}",
                widget="label";
dl_sub_header1=formonly.dl_sub_header1,
                config="Left",
                widget="label",
                color=bold Blue;
dl_sub_header3=formonly.dl_sub_header3,
                config="Right",
```

```
            widget="label",
            color=bold Blue;
dl_sub_header2=formonly.dl_sub_header2,
            config="Normal",
            widget="label",
            color=bold Blue;
dl_sub_header4=formonly.dl_sub_header4,
            config="Center",
            widget="label",
            color=bold Blue;
dl_guide=formonly.dl_guide,
            widget="label",
            wordwrap Compress,
            color=Cyan;

INSTRUCTIONS

DELIMITERS "[]"
```

Screen:

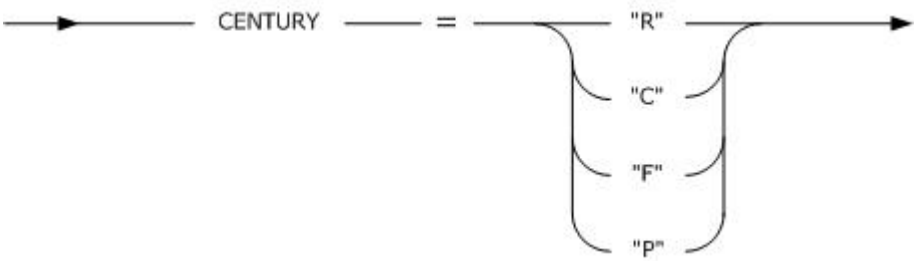


CENTURY

The CENTURY attribute specifies how to expand abbreviated one- and two-digit *year* specifications in a DATE and DATETIME field. Expansion is based on this setting (and on the year value from the system clock at runtime).

In most versions of 4GL earlier than 7.20, if the user enters only the two trailing digits of a year for literal DATE or DATETIME values, these digits are automatically prefixed with the digits 19. For example, 12/31/02 is always expanded to 12/31/1902 regardless of when the program is executed.

CENTURY can specify any of four algorithms to expand abbreviated years into four-digit year values that end with the same digits (or digit) that the user entered. CENTURY supports the same settings as the **DBCENTURY** environment variable but with a scope that is restricted to a single field.



Symbol	Algorithm for Expanding Abbreviated Years
C or c	Use the past, future or current year closest to the current date.
F or f	Use the nearest year in the future to expand the entered value.
P or p	Use the nearest year in the past to expand the entered value.
R or r	Prefix the entered value with the first two digits of the current year.

Here *past*, *closest*, *current*, and *future* are all relative to the system clock.

Unlike **DBCENTURY**, which sets a global rule for expanding abbreviated year values in DATE and DATETIME fields that do not have the CENTURY attribute, CENTURY is not case sensitive; you can substitute lowercase letters (r, c, f, p) for these uppercase letters. If you specify anything else, an error (-2018) is issued. If the CENTURY and **DBCENTURY** settings are different, CENTURY takes precedence.

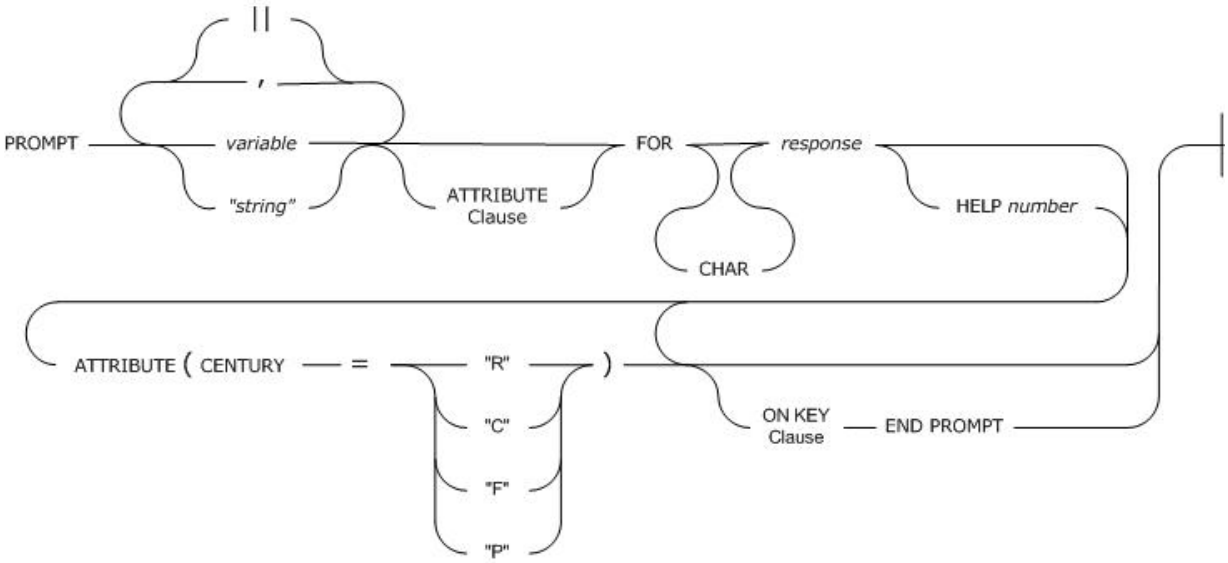
Three-digit years are not expanded. A single-digit year is first expanded to two digits by prefixing it with a zero; CENTURY then expands this value to four digits, according to the setting that you specified. Years between 1 and 99 AD (or CE) require leading zeros (to avoid expansion).

Just as with **DBCENTURY**, expansion of abbreviated years is sensitive to time of execution and to the accuracy of the system clock-calendar.

For examples of the effects of the different CENTURY settings on abbreviated year values, see “DBCENTURY” on page D-15. (Those examples are for **DBCENTURY**, but except for case-sensitivity, **DBCENTURY** and CENTURY have the same semantics.)

	The CENTURY attribute has no effect on DATETIME fields that do not include YEAR as the first time unit, or on fields that are not DATE or DATETIME fields. If an abbreviated year value is entered in a character field or a number field, for example, neither CENTURY nor **DBCENTURY** has any effect.

The ATTRIBUTES clause that can follow the FOR clause of the PROMPT statement can also specify CENTURY as an attribute, using this syntax.



Here *response* must be a DATE or DATETIME variable for CENTURY to be useful. This diagram is simplified, in that any FOR...ATTRIBUTE clause of PROMPT that specifies CENTURY can also specify other display attributes, which are listed in “ATTRIBUTE Clause”. The setting is not case sensitive but must be enclosed within quotation marks.

See “PROMPT” for descriptions of syntax terms that appear in this diagram.

Example Application:

Location:

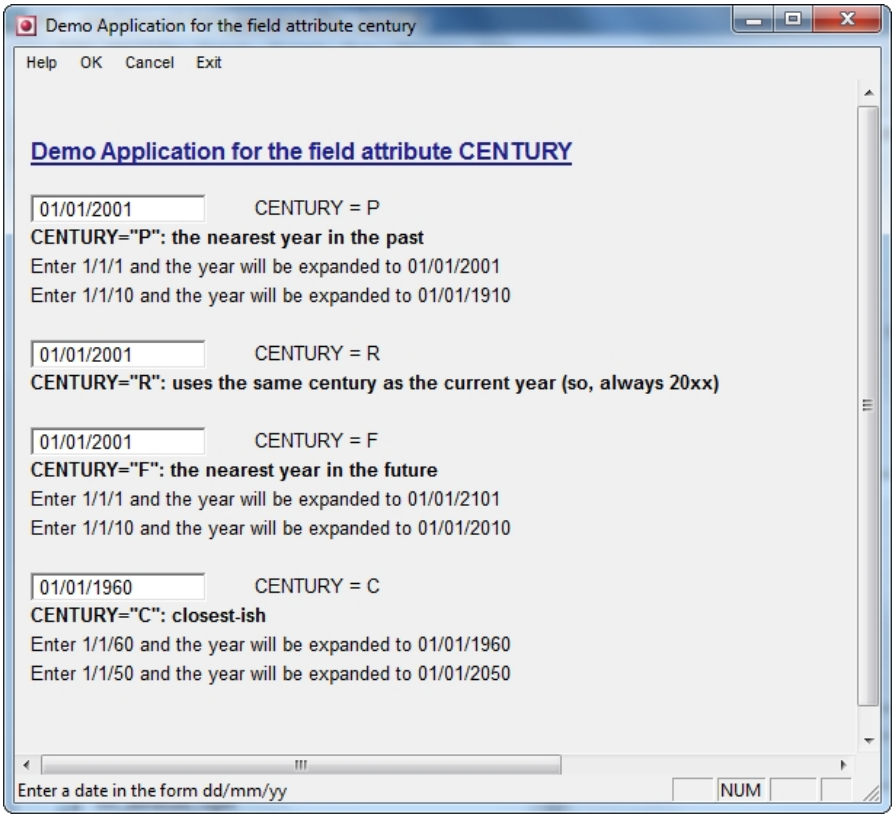
Project: forms
Program: project fm_attribute_century

Form .per Source:

```
DATABASE formonly

SCREEN
{
[dl_header                                        ]

[f1        ]  CENTURY\g \=\g \gP
[dl_info11                                ]
[dl_info12                                                  ]
[dl_info13                                                  ]

[f2        ]  CENTURY\g \=\g \gR
[dl_info21                                                                  ]

[f3        ]  CENTURY\g \=\g \gF
[dl_info31                                    ]
[dl_info32                                                ]
[dl_info33                                                 ]

[f4        ]  CENTURY\g \=\g \gC
[dl_info41              ]
[dl_info42                                                  ]
[dl_info43                                                  ]
}


ATTRIBUTES
f1=formonly.f1 type DATE,
              century="P",
              comments="Enter a date in the form dd/mm/yy";
f2=formonly.f2 type DATE,
              century="R",
              comments="Enter a date in the form dd/mm/yy";
f3=formonly.f3 type DATE,
              century="F",
              comments="Enter a date in the form dd/mm/yy";
f4=formonly.f4 type DATE,
              century="C",
              comments="Enter a date in the form dd/mm/yy";
dl_info11=formonly.dl_info11,
              config="{CENTURY=""P"": the nearest year in the past}",
              widget="label",
              color=bold;
dl_info12=formonly.dl_info12,
              config="{Enter 1/1/1 and the year will be expanded to 01/01/2001}",
              widget="label";
dl_info13=formonly.dl_info13,
```

```
            config="{Enter 1/1/10 and the year will be expanded to 01/01/1910}",
            widget="label";
dl_info21=formonly.dl_info21,
            config="{CENTURY=""R"": uses the same century as the current year (so,
always 20xx)}",
            widget="label",
            color=bold;
dl_info31=formonly.dl_info31,
            config="{CENTURY=""F"": the nearest year in the future}",
            widget="label",
            color=bold;
dl_info32=formonly.dl_info32,
            config="{Enter 1/1/1 and the year will be expanded to 01/01/2101}",
            widget="label";
dl_info33=formonly.dl_info33,
            config="{Enter 1/1/10 and the year will be expanded to 01/01/2010}",
            widget="label";
dl_info41=formonly.dl_info41,
            config="{CENTURY=""C"": closest-ish}",
            widget="label",
            color=bold;
dl_info42=formonly.dl_info42,
            config="{Enter 1/1/60 and the year will be expanded to 01/01/1960}",
            widget="label";
dl_info43=formonly.dl_info43,
            config="{Enter 1/1/50 and the year will be expanded to 01/01/2050}",
            widget="label";
dl_header=formonly.dl_header,
            config="{Demo Application for the field attribute CENTURY}",
            widget="label",
            color=bold Blue underline;

INSTRUCTIONS

DELIMITERS "[]"
```

Example Screen:



CLASS

The CLASS field attribute configures behavioural choices for certain widget types. The widgets that can make use of the CLASS attribute are as follows:

- Radio and Checkbox

  Used to state whether the field returns a value associated with an option selection, or whether the field sends a keypress when the option is selected.

- Combo

  Used to determine whether a field in a combo box can be edited.

- Hotlink

  Used to state whether a hotlink will, when clicked on, perform a hotlink action, or perform a key press or a user defined action event.

Usage

The CLASS attribute may be either unset (for the default behaviour), “key” for extended behaviour in the radio, checkbox and hotlink widgets, or “combo” in the Combo widget. The following code examples set the radio widget CLASS attribute to “key”, and also sets the combo widget CLASS attribute to “combo”:

```
F1 = formonly.F1,
WIDGET = “radio”
CLASS = “key”

F2 = formonly.F2,
WIDGET = “combo”
CLASS = “combo”
```

Example Application:

Location:

Project: forms
Program: project fm_field_widget_combo_class_combo

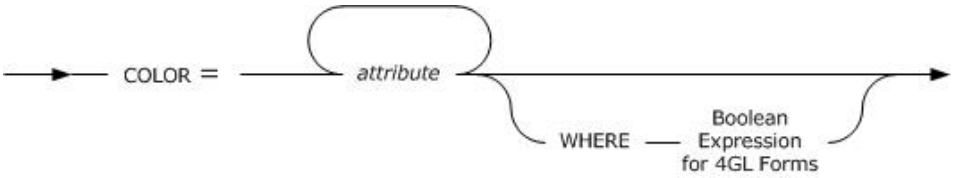
Form .per Source:

```
DATABASE formonly

SCREEN
{
```

```
 Simple\g \gCombo\g \gBox\g \gDemo\g \gApplication

 [f_combo       ]




}


ATTRIBUTES
f_combo=formonly.f_combo,
      widget="combo",
      class="combo",
      include=("Monday",
             "Tuesday",
             "Wednesday",
             "Thursday",
             "Friday",
             "Saturday",
             "Sunday");

INSTRUCTIONS

DELIMITERS "[]"
```

COLOR

The COLOR attribute displays field text in a colour or with other video attributes, either unconditionally or only if a Boolean expression is TRUE.



Element	Description
attribute	This is one of the keywords to specify a colour or intensity. You can specify zero or one colour keyword and zero or more intensity keywords. The colour keywords include BLACK, BLUE, CYAN, GREEN, MAGENTA, RED, WHITE, and YELLOW. The intensity keywords include REVERSE, LEFT, BLINK, and UNDERLINE.

If you are using **terminfo**, the only colour or intensities available are REVERSE and UNDERLINE.

Usage

If you do not use the WHERE keyword to specify a 4GL Boolean expression, the intensity or colour in your display mode list applies to the field. This example (from the 'colors' demonstration application) specifies unconditionally that field text appears in red:

```
fd2 = formonly.fd2, color=dim red;
```

Example Application:

Location:

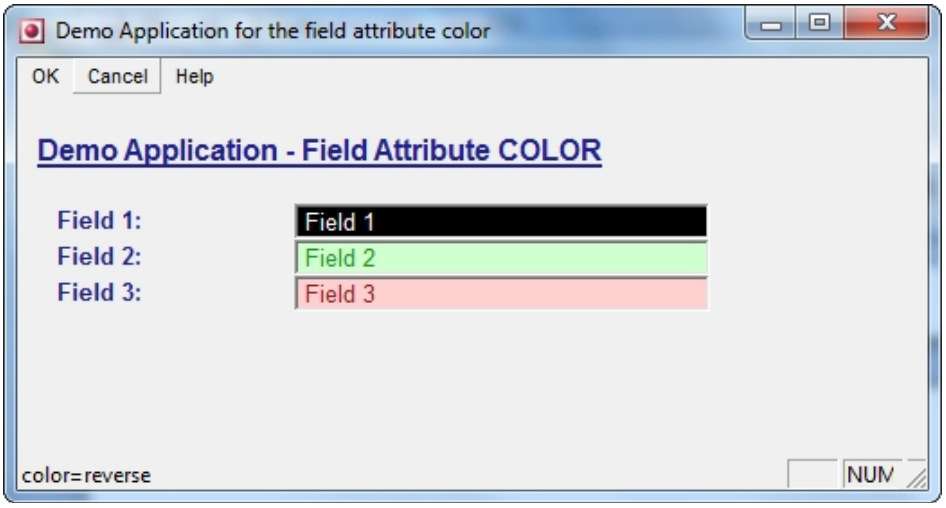
Project: forms
Program: project fm_attribute_color

Form .per Source:

```
DATABASE formonly

SCREEN
{
 [dl_header                                ]
```

```
  [dl_field1] [f1                 ]
  [dl_field2] [f2                 ]
  [dl_field3] [f3                 ]
[dl_guide                                 ]
[dl_guide                                 ]
}


ATTRIBUTES
f1=formonly.f1,
              color=reverse;
f2=formonly.f2,
              color=green;
f3=formonly.f3,
              color=red;
dl_field1=formonly.dl_field1,
              config="{Field 1:}",
              widget="label";
dl_field2=formonly.dl_field2,
              config="{Field 2:}",
              widget="label";
dl_field3=formonly.dl_field3,
              config="{Field 3:}",
              widget="label";
dl_header=formonly.dl_header,
              config="{Demo Application - Field Attribute COLOR}",
              widget="label",
              color=bold Blue;
dl_guide=formonly.dl_guide,
              widget="label",
              wordwrap Compress,
              color=Cyan;

INSTRUCTIONS

DELIMITERS "[]"
```

Example Screen:



Specifying Logical Conditions with the WHERE Option

You can also use the keywords, symbols, and operators that are allowed in 4GL Boolean expressions, including LIKE, MATCHES, TODAY, and CURRENT, in a WHERE clause to specify conditional attributes. If the Boolean expression evaluates as FALSE or NULL, the field is displayed with default characteristics, rather than with those specified by *display mode*.

For more information, see “Default Attributes”.

Example Application:

Location:

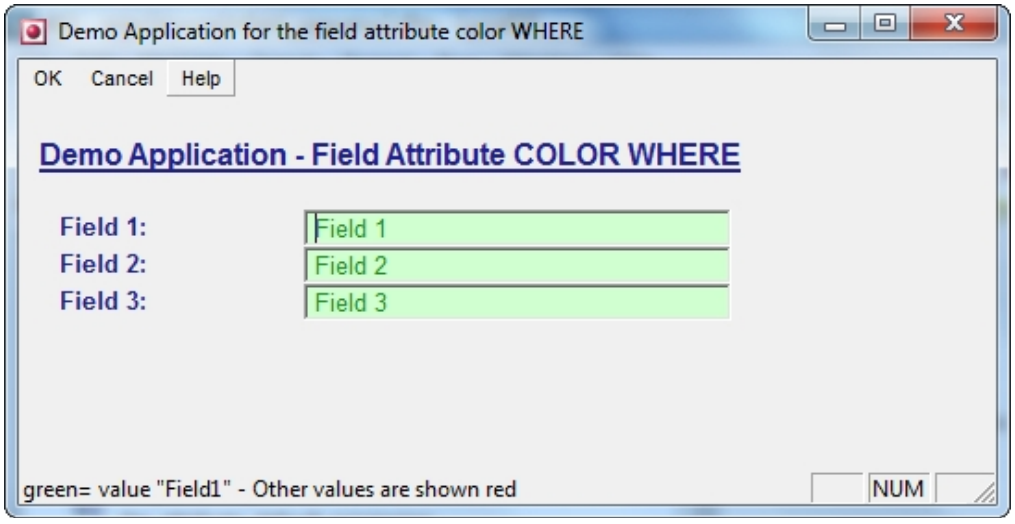
Project: forms
Program: project fm_attribute_color_where

Form .per Source:

```
DATABASE formonly

SCREEN
{
 [dl_header                                ]

  [dl_field1] [f1                  ]
  [dl_field2] [f2                  ]
  [dl_field3] [f3                  ]
[dl_guide                                   ]
[dl_guide                                   ]
}


ATTRIBUTES
f1=formonly.f1,
              comments="green= value ""Field1"" - Other values are shown red",
              color=red,
              color=green WHERE f1="Field 1";
f2=formonly.f2,
              comments="green= value ""Field2"" - Other values are shown red",
              color=red,
              color=green WHERE f2="Field 2";
f3=formonly.f3,
              comments="green= value ""Field3"" - Other values are shown red",
              color=red,
              color=green WHERE f3="Field 3";
dl_field1=formonly.dl_field1,
              config="{Field 1:}",
              widget="label";
```

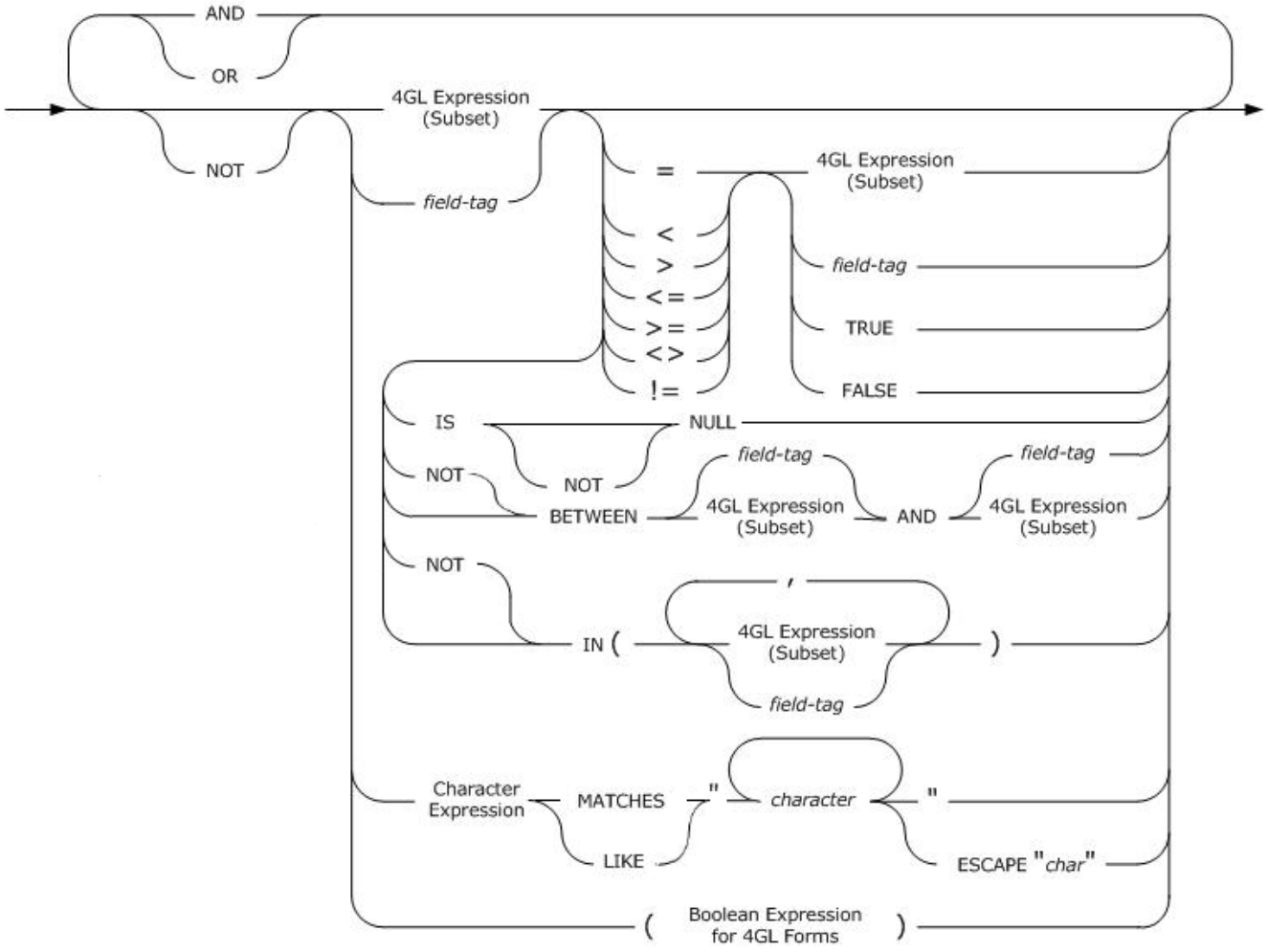
```
dl_field2=formonly.dl_field2,
              config="{Field 2:}",
              widget="label";
dl_field3=formonly.dl_field3,
              config="{Field 3:}",
              widget="label";
dl_header=formonly.dl_header,
              config="{Demo Application - Field Attribute COLOR WHERE}",
              widget="label",
              color=bold Blue;
dl_guide=formonly.dl_guide,
              widget="label",
              wordwrap Compress,
              color=Cyan;

INSTRUCTIONS

DELIMITERS "[]"
```

Example Screen:



Boolean Expressions in 4GL Form Specification Files

The syntax of 4GL Boolean expressions in the WHERE clause of a COLOR attribute specification is shown in the following diagram:



Element	Description
Char	This is a single character, enclosed by pairs of single or double speech marks (‘ x ‘) or (“ x “).
character	This is one or more literal or special characters, enclosed by pairs of single or double speech marks.
field-tag	This is the field-tag of the current field.

In this diagram, terms for other 4GL expressions are restricted subsets. Except for the constants TRUE and FALSE, you cannot reference the *name* of a program variable in the WHERE clause of a COLOR attribute specification.

You can, however, include a *field-tag* or a *literal* value wherever the name of a variable can appear in a 4GL expression that is a component of the 4GL Boolean expression.

If any component of a 4GL Boolean expression is null, the value of the entire 4GL Boolean expression is FALSE (rather than null), unless the IS NULL operator is also included in the expression. As in other Boolean expressions of 4GL, applying the NOT operator to a null value does not change its FALSE evaluation. The conditional attribute is displayed only if the overall condition is true. In the following example, the value of the expression is FALSE if a null value appears in the display field whose field tag is **fb**:

```
fb=formonly.fa_bold,
              color=bold,
              color=bold red where fb = "Red",
              color=bold green where fb = "Green",
              color=bold blue where fb = "Blue",
              color=bold yellow where fb = "Yellow",
              color=bold magenta where fb = "Magenta",
              color=bold cyan where fb = "Cyan",
              color=bold black where fb = "Black",
              color=bold white where fb = "White";
```

If you include a *field-tag* value in a 4GL Boolean expression when you specify a conditional COLOR attribute, 4GL replaces *field-tag* at runtime with the current value in the screen field and evaluates the expression.

If *field-tag* references a field that is linked to a database column of data type TEXT or BYTE or to a FORMONLY field of either of those two data types, only the IS NOT NULL or IS NULL keywords can include that field tag in an expression. The specified colour or intensity is applied to the <BYTE value> message, not to the BYTE data value because only the PROGRAM attribute can display a BYTE value. Values of the TEXT data type are displayed in the field, beginning with the first printable data character. If the TEXT value is too large to fit in the field, characters beyond what the field can hold are not displayed.

Specifying Ranges of Values and Set Membership

Conditional COLOR attributes can specify SQL Boolean operators that are not valid in ordinary 4GL Boolean expressions. You can use the BETWEEN...AND operator to specify a range of number, time, or character values. Here the first expression cannot be greater than the second (for number expressions), later than the second (for time expressions), or later in the code-set order than the second (for character expressions).

The section titled “ASCII Character Set,” lists the numeric values of ASCII characters, which is the collating sequence for U.S. English locales.

The WHERE clause of a COLOR field description can also use the IN operator to specify a comma-separated list (enclosed between parentheses) of values with which to compare the field tag or expression.

"Set Membership and Range Tests" describes the syntax of the BETWEEN...AND and IN Boolean operators in conditional COLOR specifications for 4GL forms.

Data Type Compatibility

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

If a time expression component of a 4GL Boolean expression is an INTERVAL data type, any other time expression that is compared to it 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).

Data Type Conversion in 4GL Boolean Expressions

If you specify a number, character, or time expression in a context where a 4GL Boolean expression is expected, 4GL applies the following rules after evaluating the number, character, or time expression:

- If the value is a non-zero real number (or a character string representing a non-zero number) or a non-zero INTERVAL, or any DATE or DATETIME value, the 4GL Boolean value is TRUE.
- If the value is null, and the IS NULL keywords are also included in the expression, the value of the 4GL Boolean expression is TRUE.
- Otherwise, the 4GL Boolean expression is FALSE.

The Display Modes

The display and intensity keywords of the COLOR attribute have the same effects on a field as the same keywords of the **upscol** utility.

The following table shows the effects of the colour attribute keywords on a monochrome terminal.

Attribute	Display	Attribute	Display
WHITE	Normal	CYAN	Dim
YELLOW	Bold	GREEN	Dim
MAGENTA	Bold	BLUE	Dim
RED	Bold	BLACK	Dim

The following table shows the effects of the intensity attribute keywords on a colour terminal.

Attribute	Display
NORMAL	White
BOLD	Red
DIM	Blue

The LEFT attribute produces a left-aligned display in a screen field of any number data type. It has no effect on fields of other data types. (Without the COLOR = LEFT specification, number values are right-aligned by default.)

The next lines specify display attributes if Boolean expressions are TRUE:

```
f001 = invoice.inv_total, COLOR = RED BLINK WHERE \
f001 = 0.00;
f002 = contact.fname, COLOR = RED WHERE \
f002 = "John";
f003 = contact.lname, COLOR = RED WHERE \
f003 LIKE "Hoelzl";
f004 = invoice_line.line_qty, COLOR = GREEN WHERE f004 < 5;
f005 = formonly.check, COLOR = BLUE REVERSE WHERE \
f005 IS NULL,
COLOR = YELLOW WHERE f005 BETWEEN 5000 AND 10000,
COLOR = GREEN BLINK WHERE f005 > 10000;
```

The following expression is TRUE if the field **f_pattern** does not include the underscore character:

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

Related Attributes

DISPLAY LIKE, INVISIBLE, REVERSE

CONFIG

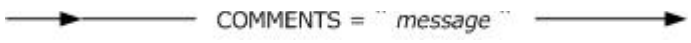
The config attribute is used in conjunction with most widget attributes.

Widget	Class	Configuration Values	Example
bmp	N/a	The keypress associated with the field, and the default image to be displayed on the button	WIDGET="bmp", CONFIG="myicon.jpg \ F1";
browser	N/a	The URL to be opened with the form	WIDGET="browser" CONFIG="www.querix.com"
button	N/a	The keypress associated with the button, and the default label for the button	WIDGET="button", CONFIG="F1 OK";
calendar	N/a	N/a	WIDGET="calendar";
check		Value1: The checked value of the field Option2: The unchecked value for the field Option3: the label associated with the field	WIDGET="check" CONFIG="on off \ {My Label}";
	key	Option 1: The keypress to send when the field is 'checked' Option 2: The keypress to send when the field is 'unchecked' Option 3:	WIDGET="check", CLASS="key", CONFIG="F1 F2 \ {My Label}";
combo	Combo	N/a	WIDGET="combo", INCLUDE=("1","2");
label	N/a	The default text for the label	WIDGET="label", CONFIG="My Label";
Field_bmp	N/a		WIDGET="field_bmp", CONFIG="myimage.jpg \ F1";
radio		Option 1a: The value associated with the option. Option 1b: The label associated with this option	WIDGET="radio", CONFIG="1 {Option 1} \ 2 {Option 2}";
	key	Option 1a: The keypress when the option is selected Option 1b: The label for this option	WIDGET="radio", CONFIG="F1 {Option 1} \ F2 {Option 2}";

COMMENTS

The COMMENTS attribute displays a message on the Comment line/location. The message is displayed when the cursor moves to the specified field and is erased when the cursor moves to another field.

Syntax Diagram



Where the *message* element is a character string between speech marks.

Difference between text mode and GUI clients

Text Mode:

The comments text will only be displayed if the cursor moves during an input in a field with a comments text. Comments will be displayed in the default comment line (last line) or the line number specified using the OPTIONS COMMENT LINE <n>. It will always start in column 1.

You can not specify any attributes for the comments text format.

GUI Clients:

The comments text will be displayed (by default) in the 4GL specified comment line location if the cursor moves into a field. If the user moves the mouse cursor over a field with a comment text, a tooltip with the comments text will be displayed.

The window script property 'commentLine' specifies the comments text display location. Available options are:

- default — 4GL specified comment line
- <n> — Any other line number
- statusbar — Statusbar
- messagebox — Message box
- console — Console Window
- frame — MDI Container Frame statusbar

For example:

```
?.?.commentLine: statusbar
```

will redirect any comment text to the statusbar.

For more detailed information on the topic ‘comment line’ redirection, refer to the manual ‘Graphical Client Reference Guide’.

Usage

The *message* string must appear between quotation marks (") on a single line of the form specification file.

In the following example, the field description specifies a message for the Comment line to display. The message appears when the screen cursor enters the field that is linked to the **cont_name** column of the **contact** table. In the **cms** database, this column contains the first name of a contact:

```
f001=contact.cont_name,comments="Enter first name",
       required,upshift;
```

The most common application of the COMMENTS attribute is to give information or instructions to the user. This application is particularly appropriate when the field accepts only a limited set of values.

We advise keeping the information brief as the same form may be used by different program and function modules.

The Position of the Comment Line

The default position of the Comment line in the 4GL screen is line 23. You can reset this position with the OPTIONS COMMENT LINE <n> statement. The default position of the Comment line in a 4GL window is LAST. You can reset this position in the OPTIONS statement to specify a new position in all 4GL windows at runtime. Alternatively, you can reset it in the ATTRIBUTE clause of the appropriate OPEN WINDOW statement if you want the new position in a specific 4GL window.

If the OPEN WINDOW statement specifies COMMENT LINE OFF, any output to the Comment line of the specified 4GL window is hidden even if the window displays a form that includes fields that include the COMMENTS attribute.

We strongly recommend controlling the comment line location with the OPTIONS statement, not with any window attribute.

Example Program

Location:

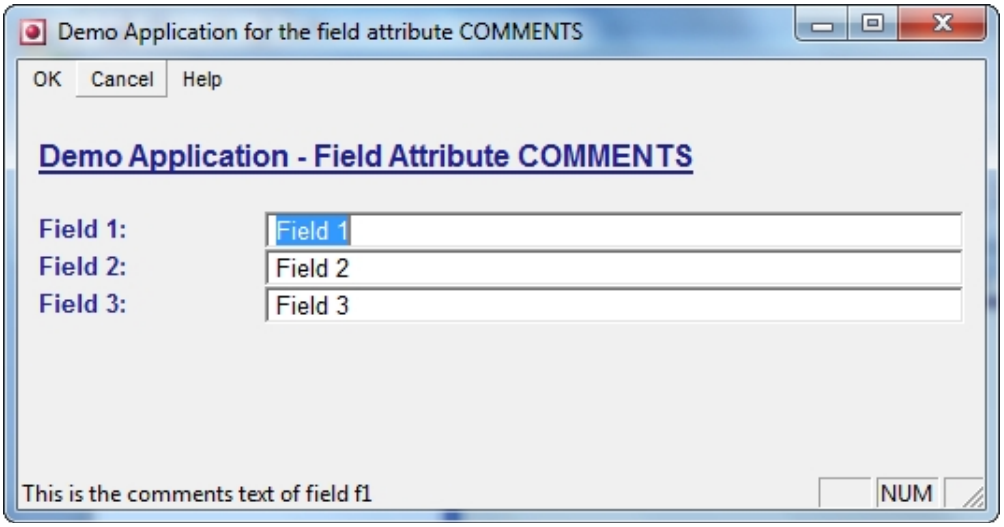
Project: forms
Program: project fm_attribute_comments

Form .per Source:

```
DATABASE formonly

SCREEN
{
```

```
  [dl_header                          ]

  [dl_field1] [f1                     ]
  [dl_field2] [f2                     ]
  [dl_field3] [f3                     ]
}


ATTRIBUTES
f1=formonly.f1,
             comments="This is the comments text of field f1";
f2=formonly.f2,
             comments="This is the comments text of field f2";
f3=formonly.f3,
             comments="This is the comments text of field f3";
dl_field1=formonly.dl_field1,
             config="{Field 1:}",
             widget="label";
dl_field2=formonly.dl_field2,
             config="{Field 2:}",
             widget="label";
dl_field3=formonly.dl_field3,
             config="{Field 3:}",
             widget="label";
dl_header=formonly.dl_header,
             config="{Demo Application - Field Attribute COMMENTS}",
             widget="label",
             color=bold Blue;

INSTRUCTIONS

DELIMITERS "[]"
```

Example Screen:



DEFAULT

The DEFAULT attribute assigns a default value to a field during data entry.

Syntax Diagram

Where *value* is a default value for the field. This is a restricted expression (static value), and cannot reference a variable or a function defined by the programmer.

Usage

Default values have no effect when you execute the INPUT statement by using the WITHOUT DEFAULTS option. In this case, 4GL displays the values in the program variables list on the screen. The situation is the same for the INPUT ARRAY statement except that 4GL displays the default values when the user inserts a new row.

If the field is FORMONLY, you must also specify a data type when you assign the DEFAULT attribute to a field with a non-char data type. For more information on this see “FORMONLY Fields”.

If you do not use the WITHOUT NULL INPUT option in the DATABASE section, all fields default to null values unless you use the DEFAULT attribute. If you use the WITHOUT NULL INPUT option in the DATABASE section and you do not use the DEFAULT attribute, character fields default to blanks, number and INTERVAL fields to 0, and MONEY fields to $0.00. The default DATE value is 12/31/1899. The default DATETIME value is 1899-12-31 23:59:59.99999.

	You can not specify expressions or variables as default values. Also, you cannot assign the DEFAULT attribute to fields of data type TEXT or BYTE.

Literal Values

The *value* can be a quoted string, a literal number, a literal DATE value, a literal DATETIME value, or a literal INTERVAL value. Or the value can be a built-in function or operator that returns a single value of a data type compatible with that of the field. For details, see the “Built-In Functions” Reference Guide.

If you include in the *value* list a character string that contains a blank space, a comma, or any special characters, or a string that does not begin with a letter, you must enclose the entire string in quotation marks.

For a DATE field, you must also enclose any literal value in quotation marks ("). For a DATETIME or INTERVAL field, you can enclose a static *value* in quotation marks.

Built-In 4GL Operators and Functions as Values

Besides these literal values, you can also specify a built-in 4GL function or operator that returns a single value of the appropriate data type.

Arguments or operands must be a literal value, a built-in 4GL function or operator that returns a single value, or the named constants TRUE or FALSE. For example, a default value of data type INTERVAL can be specified in the format:

```
integer UNITS time-unit
```

Here *integer* can be a positive or negative literal integer, as described in “Literal Integers”, or an expression in parentheses that evaluates to an integer, and *time-unit* is a keyword from an INTERVAL qualifier, such as MONTH, DAY, HOUR. This qualifier must be consistent with the explicit or implied data type declaration of the field; do not, for example, specify YEAR or MONTH as the *time-unit* value for a DAY TO FRACTION field.

For a DATETIME or INTERVAL field, you can enter it as an unquoted literal using built-in functions/operators, as in the following examples:

```
DATETIME (2012-12) YEAR TO MONTH
INTERVAL (10:12) HOUR TO MINUTE
- INTERVAL (28735) DAY TO DAY
```

For more information, see “DATETIME Qualifier” and “INTERVAL Literal”.

Use the TODAY operator as *value* to assign the current date as the default value of a DATE field. Use the CURRENT operator as *value* to assign the current date and time as the default for a DATETIME field. (4GL does not assign these values automatically as defaults, so you must specify them explicitly.) These expressions are evaluated at runtime, not at compile time.

The following field descriptions specify DEFAULT values:

```
f1 = formonly.f1 TYPE DATE, DEFAULT=TODAY;
f2 = formonly.f2 TYPE DATETIME YEAR TO SECOND, DEFAULT=CURRENT;
f3 = formonly.f3 TYPE CHAR, DEFAULT=”Yes”;
```

Related Attributes

INCLUDE, REQUIRED, VALIDATE LIKE

Example Application:

Location:

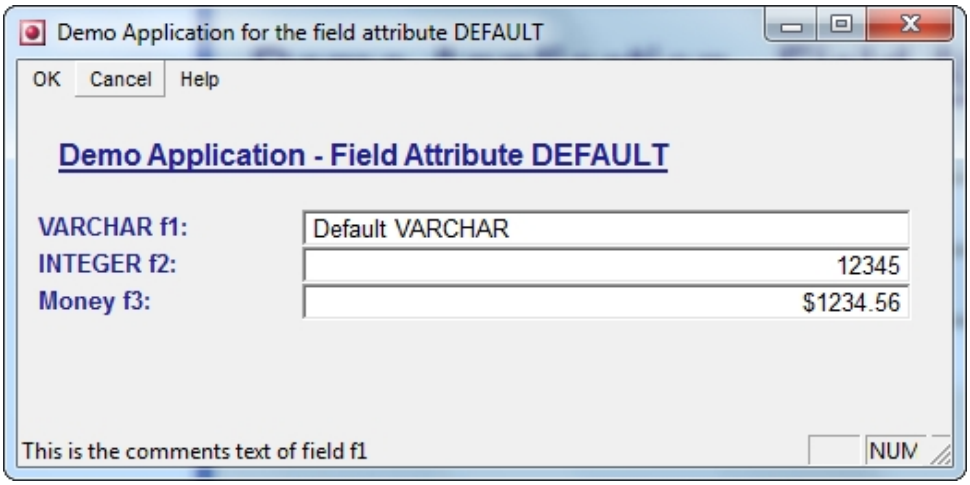
Project: forms
Program: project fm_attribute_default and fm_attribute_default_expression

Form .per Source:

```
DATABASE formonly

SCREEN
{
  [dl_header                              ]

 [dl_field1  ][f1                         ]
 [dl_field2  ][f2                         ]
 [dl_field3  ][f3                         ]
 [dl_guide                                ]
 [dl_guide                                ]
}


ATTRIBUTES
f1=formonly.f1 type VARCHAR,
              comments="This is the comments text of field f1",
              default="Default VARCHAR";
f2=formonly.f2 type INTEGER,
              comments="This is the comments text of field f2",
              default=12345;
f3=formonly.f3 type MONEY,
              comments="This is the comments text of field f3",
              default=1234.56;
dl_field1=formonly.dl_field1,
              config="{VARCHAR f1:}",
              widget="label";
dl_field2=formonly.dl_field2,
              config="{INTEGER f2:}",
              widget="label";
dl_field3=formonly.dl_field3,
              config="{Money f3:}",
              widget="label";
dl_header=formonly.dl_header,
              config="{Demo Application - Field Attribute DEFAULT}",
              widget="label",
              color=bold Blue;
dl_guide=formonly.dl_guide,
              widget="label",
              wordwrap Compress,
```

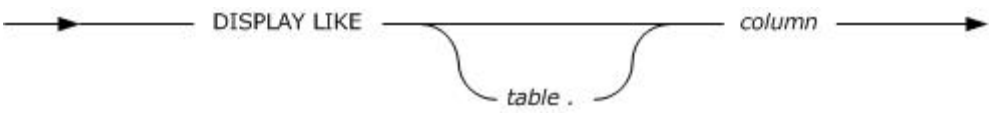
```
                color=Cyan;

    INSTRUCTIONS

    DELIMITERS "[]"
```

Example Screen:



DISPLAY LIKE

The DISPLAY LIKE attribute takes attributes that the **upscol** utility assigned to a specified column in the **syscolatt** table and applies them to a field. It works in much the same way as the FORMAT attribute, except the data is taken from a database rather than an input string.



Element	Description
table	This is the unqualified name or alias of a database table, synonym, or view, as declared in the TABLES section. (This value is not required unless several columns in different tables have the same name or if the table is an external table or an external, distributed table.)
column	This is the name of a column in *table* or, if you omit *table*, the unique identifier of a column in one of the tables that you declared in the TABLES section. The column cannot be of data type BYTE

Usage

Specifying this attribute is equivalent to listing all the attributes that are assigned to ***table.column*** in the **syscolatt** table. You do not need to specify the DISPLAY LIKE attribute if the field is linked to *table.column* in the field name specification.

The following example instructs Lycia to apply the default display attributes of the **invoice.inv_total** column to a FORMONLY field:

```
s12 = FORMONLY.total, DISPLAY LIKE invoice.inv_total;
```

Lycia evaluates the LIKE clause at compile time, not at runtime. If the database schema changes, you might need to recompile a program that uses the LIKE clause. Even if all of the fields in the form are FORMONLY, this attribute requires QFORM to access the database that contains *table*.

Related Attribute

VALIDATE LIKE

DOWNSHIFT

Assign the DOWNSHIFT attribute to a character field when you want Lycia to convert uppercase letters entered by the user to lowercase letters, both on the screen and in the corresponding program variable. The field attribute UPSHIFT does the opposite.

Usage

Because uppercase and lowercase letters have different values, storing character strings in one or the other format can simplify sorting and querying a database.

By specifying the DOWNSHIFT attribute, you instruct Lycia to convert character input data to lowercase letters in the program variable when the characters are written to the variable as well as in the input field at the time of data entry.

The maximum length of a character value to which you can apply the DOWNSHIFT attribute is 511 characters.

Related Attribute

UPSHIFT

Example Application:

Location:

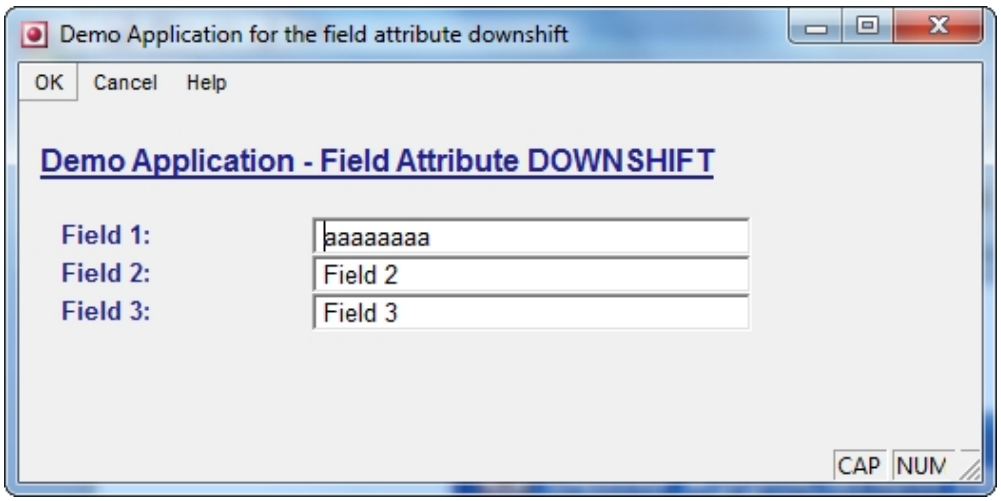
Project: forms
Program: project fm_attribute_downshift

Form .per Source:

```
DATABASE formonly

SCREEN
{
 [dl_header                                  ]

  [dl_field1] [f1                  ]
  [dl_field2] [f2                  ]
  [dl_field3] [f3                  ]
[dl_guide                                   ]
[dl_guide                                   ]
}

ATTRIBUTES
f1=formonly.f1,
             downshift;
f2=formonly.f2,
             downshift;
```

```
f3=formonly.f3,
              downshift;
dl_field1=formonly.dl_field1,
              config="{Field 1:}",
              widget="label";
dl_field2=formonly.dl_field2,
              config="{Field 2:}",
              widget="label";
dl_field3=formonly.dl_field3,
              config="{Field 3:}",
              widget="label";
dl_header=formonly.dl_header,
              config="{Demo Application - Field Attribute DOWNSHIFT}",
              widget="label",
              color=bold Blue;
dl_guide=formonly.dl_guide,
              widget="label",
              wordwrap Compress,
              color=Cyan;

INSTRUCTIONS

DELIMITERS "[]"
```

Example Screen:



FORMAT

You can use the FORMAT attribute with a DECIMAL, SMALLFLOAT, FLOAT, or DATE field to control the format of output displays.

Syntax Diagram

Where *format-string* is a string of characters that describes the format in which data is to be displayed. The string must be placed between speech marks.

Usage

This attribute can format data that the application displays in the field (Use the PICTURE attribute to format data entered in the field by the user Lycia right-aligns the data in the field.

Formatting Number Values

For DECIMAL, SMALLFLOAT, and FLOAT data types, *format-string* consists of hash symbols (#) that represent digits and a decimal point. For example, "###.##" produces at least three places to the left of the decimal point and exactly two to the right.

If the actual number displayed requires fewer characters than *format-string* specifies, Lycia right-aligns it and pads the left with blanks.

If necessary to satisfy the *format-string* specification, Lycia rounds number values before it displays them.

In GLS, the NUMERIC setting in the locale files affects how *format-string* is interpreted for numeric data. In the format string, the period symbol (.) is not a literal character but a placeholder for the decimal separator specified by environment variables or by locale file settings. Likewise, the comma (,) is a placeholder for the thousands separator specified by environment variables.

Thus, the format string #,###.## formats the value 1234.56 as 1,234.56 in a U.S. English locale but as 1.234,56 in a German locale.

When using FORMAT with widgets that generate values (such as radio), be careful to ensure that the FORMAT corresponds to the expected value

Formatting DATE Values

For DATE data types, 4GL recognizes these symbols as special in *format-string*.

Symbol	Effect
mm	Displays a two-digit format of the month, such as 06 for June, or 07 for July.
mmm	Displays a three letter English abbreviation of the month, such as Jun or Jul.
dd	Displays a two-digit representation of the day of the month.
ddd	Displays a three letter English abbreviation for the day of the week, such as Mon or Tue.
yy	Displays the last two-digits of the year.
yyyy	Displays the year as a four-digit figure.

For DATE fields, QFORM interprets any other characters as literals and displays them wherever you place them within *format-string*.

These *format-string* examples and their corresponding display formats for DATE fields display the twenty-third day of September 1999.

Input	Result
no FORMAT *attribute*	09/23/1999
FORMAT = “mm/dd/yy”	09/23/99
FORMAT = “mmm dd, yyyy”	Sep 23, 1999
FORMAT = “yymmdd”	990923
FORMAT = “dd-mm-yy”	23-09-99
FORMAT = “(ddd.) mmm. dd, yyy”	(Thu.) Sep. 23, 1999

In GLS, the **mmm** and **ddd** specifiers in a format string can display language-specific month name and day name abbreviations. To do this you will need to install message files in a subdirectory of **$INFORMIXDIR/msg** and then reference that subdirectory by using the environment variable **DBLANG**.

For example, in a Spanish locale, the **ddd** specifier translates the day Saturday into the day name abbreviation **Sab**, which stands for *Sabado*.

Related Attribute

PICTURE

Example Application

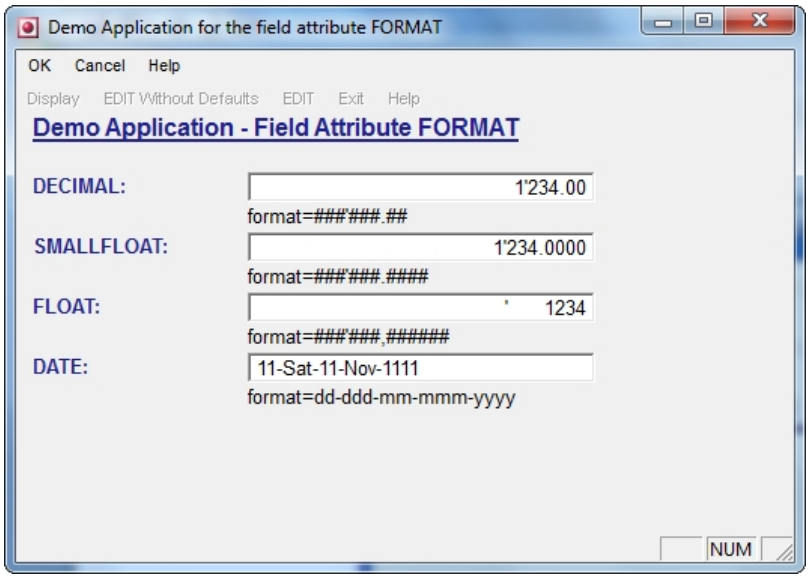
Location:

Project: forms
Program: project fm_attribute_format

Form .per Source

```
DATABASE formonly

SCREEN
{
 [dl_header                                  ]

 [dl_field1  ][f1                  ]
              [dl_attrib_f1        ]
 [dl_field2  ][f2                  ]
              [dl_attrib_f2        ]
 [dl_field3  ][f3                  ]
              [dl_attrib_f3        ]
 [dl_field4  ][f4                  ]
              [dl_attrib_f4         ]
[dl_guide                                   ]
[dl_guide                                   ]
}


ATTRIBUTES
f1=formonly.f1 type DECIMAL,
     format="###'###.##";
f2=formonly.f2 type SMALLFLOAT,
     format="###'###.####";
f3=formonly.f3 type FLOAT,
     format="###'###,######";
f4=formonly.f4 type DATE,
     format="dd-ddd-mm-mmm-yyyy";
dl_field1=formonly.dl_field1,
     config="DECIMAL:",
     widget="label";
dl_field2=formonly.dl_field2,
     config="SMALLFLOAT:",
     widget="label";
dl_field3=formonly.dl_field3,
     config="FLOAT:",
     widget="label";
dl_header=formonly.dl_header,
     config="{Demo Application - Field Attribute FORMAT}",
     widget="label",
     color=bold Blue;
dl_guide=formonly.dl_guide,
     widget="label",
     wordwrap Compress,
     color=Cyan;
dl_field4=formonly.dl_field4,
     config="DATE:",
```

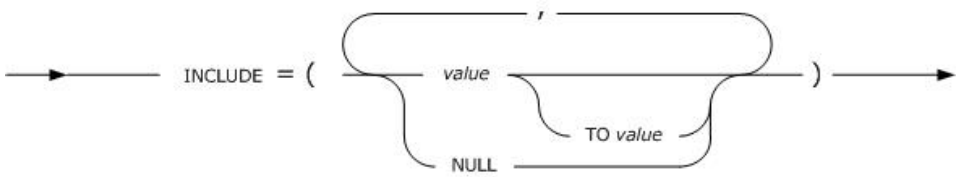
```
     widget="label";
dl_attrib_f1=formonly.dl_attrib_f1,
     config="format=###'###.##",
     widget="label";
dl_attrib_f2=formonly.dl_attrib_f2,
     config="format=###'###.####",
     widget="label";
dl_attrib_f3=formonly.dl_attrib_f3,
     config="format=###'###,######",
     widget="label";
dl_attrib_f4=formonly.dl_attrib_f4,
     config="format=dd-ddd-mm-mmm-yyyy",
     widget="label";

INSTRUCTIONS

DELIMITERS "[]"
```



INCLUDE

The INCLUDE attribute specifies acceptable values for a field and causes Lycia to check at runtime before accepting an input value.

Syntax Diagram



The element *value* should be placed within parentheses. *Value* can be one or more values, and/or ranges of values. If more than one value is used, the list must be comma separated.

Difference between text mode and GUI clients

Text Mode:

Include fields will be rendered as normal text fields where the application verifies the contents of the field with the contents of the list.

GUI Clients:

Include fields convert by default a normal text field to a combo list box. The contents of the include list will be used to populate the combo list. This behaviour can be turned off by using the corresponding script option. It is also possible to specify a maximum range for the combo boxes; if the number of list values is larger than this number, no automatic combo box generation will take place. For more information on combo list related script options, refer to the 'graphical client reference guide'.

If your application operates only with GUI clients, we strongly recommend using combo list widget instead.

Additionally, there is a fast range of combo list functions fgl_list_xxx() to manage, populate, search etc., the contents of the combo list box dynamically at runtime. For more information, refer to the 'built-in functions' guide.

Usage

Each *value* specification is a restricted expression that cannot include the name of any 4GL variable or programmer-defined function. It can include literal values, built-in functions, operators, and the constants TRUE, FALSE, and NOTFOUND.

The same rules for DEFAULT attribute values also apply to INCLUDE values. TEXT and BYTE fields cannot have the INCLUDE attribute.

If a field has the INCLUDE attribute, the user must enter an acceptable value (from the *value* list) before 4GL accepts a new row. If the *value* list does not include the default value, the INCLUDE attribute behaves like the REQUIRED attribute, and an acceptable entry is required. Include the NULL keyword in the *value* list to specify that it is acceptable for the user to press RETURN without entering any value.

```
f006 =        formomnly.terms,
              INCLUDE = (NULL, "Y", "N");
```

In this example, the NULL keyword allows the user to enter nothing. You *cannot* accomplish the same thing by substituting a string of blanks for the NULL keyword in the INCLUDE specification because for most data types a null value is different from ASCII 32, the blank character.

Including a COMMENTS attribute for the same field to describe acceptable values makes data entry easier because you can display a message to advise the user of whatever restrictions you have imposed on data entry:

```
i18 =         formonly.rating,
              INCLUDE = (1 TO 10),
              COMMENTS = "Enter your rating (1 lowest, 10 highest)";
```

If you include in the *value* list a character string that contains a blank space, a comma, or any special characters or a string that does not begin with a letter, you must enclose the entire string in quotation marks ("). (If you omit the quotation marks, any uppercase letters are down-shifted.)

Ranges of Values

You can use the TO keyword to specify an inclusive range of acceptable values. For example, ranges in the following field description include the consonants of the alphabet:

```
i20 = formonly.consonents,
INCLUDE = (NULL, "B" TO "D", "F" TO "H", “J” TO “N”, “P” TO “T”, “V” TO “Z”),
```

When you specify a range of values, the *lower* value must appear first. The meaning of *lower* depends on the data type of the field:

- For number or INTERVAL fields, it is the larger (or only) negative value, or (if neither value is negative) the value closer to zero.
- For other time fields, it is the earlier DATE or DATETIME value.
- For character fields, the lower value is the string that starts with a character closer to the beginning of the collating sequence.
- When using non-default locales, the code-set order is used as the collating sequence.

In a number field, for example, the range "5 TO 10" is valid. In a character field, however, it produces a problem at runtime.

FORMONLY Fields

You must specify a data type when you assign the INCLUDE attribute to a FORMONLY field with a datatype other than CHAR. The TYPE clause is required in the following example:

```
f006 = FORMONLY.agree TYPE SMALLINT,
INCLUDE = (1, 2, 3);
```

Related Attributes

COMMENTS, DEFAULT, REQUIRED

Example Application:

Location:

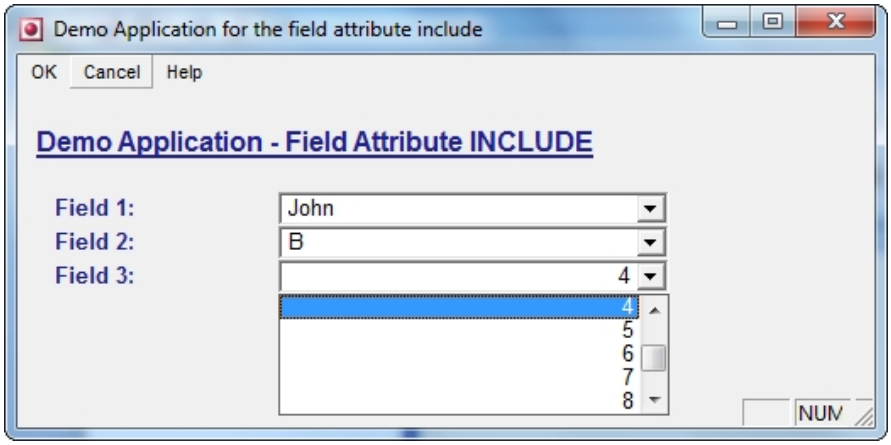
Project: forms
Program: project fm_attribute_include

Form .per Source:

```
DATABASE formonly

SCREEN
{
 [dl_header                                ]

  [dl_field1] [f1                  ]
  [dl_field2] [f2                  ]
  [dl_field3] [f3                  ]
[dl_guide                                 ]
[dl_guide                                 ]
}


ATTRIBUTES
f1=formonly.f1 type VARCHAR,
             include=("John","Dom","Sean","Tim","Mary","Gemma","Lucy","Marta","Tom"
,"Rowan");
f2=formonly.f2 type CHAR,
             include=("A" TO "G");
f3=formonly.f3 type SMALLINT,
             include=(1 TO 10);
dl_field1=formonly.dl_field1,
             config="{Field 1:}",
             widget="label";
```

```
dl_field2=formonly.dl_field2,
              config="{Field 2:}",
              widget="label";
dl_field3=formonly.dl_field3,
              config="{Field 3:}",
              widget="label";
dl_header=formonly.dl_header,
              config="{Demo Application - Field Attribute INCLUDE}",
              widget="label",
              color=bold Blue;
dl_guide=formonly.dl_guide,
              widget="label",
              wordwrap Compress,
              color=Cyan;

INSTRUCTIONS

DELIMITERS "[]"
```

Example Screen:



INVISIBLE

The INVISIBLE attribute prevents user-entered data from being echoed on the screen during a CONSTRUCT, INPUT, INPUT ARRAY, or PROMPT statement. It behaves closely like the field attribute password.

Usage

Characters that the user enters in a field with this attribute are not displayed during data entry, but the cursor moves through the field as the user types. No other aspects of data entry are affected by the INVISIBLE attribute.

The following example illustrates the use of the INVISIBLE attribute:

```
f1 =          FORMONLY.secret_password TYPE LIKE users.password,
              INVISIBLE,
              COMMENTS = "Enter your secret password.";
```

Difference between text and GUI clients

Text Mode Client:

Any data displayed to and during the input (entered by the user) will not be shown.

GUI Client:

One asterisk symbol '*' for each character will only be displayed in input mode and in display mode, the field will be empty. It behaves identically to field attribute password with the only difference, that it will not display asterisk '*' characters in the normal display mode

Only use this attribute with normal fields, not with any GUI widgets such as radio buttons and check boxes.

Related Attribute

PASSWORD

Example Application:

Location:

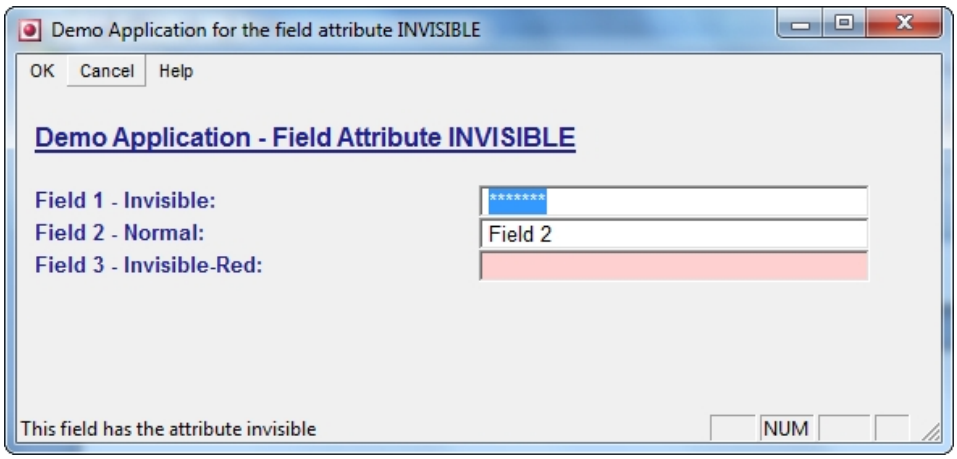
Project: forms
Program: project fm_attribute_invisible

Form .per Source:

```
DATABASE formonly
```

```
SCREEN
{
 [dl_header                                  ]

 [dl_field1              ][f1                ]
 [dl_field2              ][f2                ]
 [dl_field3              ][f3                ]
 [dl_guide                                   ]
 [dl_guide                                   ]
}


ATTRIBUTES
f1=formonly.f1,
              comments="This field has the attribute invisible",
              invisible;
f2=formonly.f2,
              comments="This field has NO invisible attribute";
f3=formonly.f3,
              comments="This field has the attribute invisible",
              invisible,
              color=bold Red;
dl_field1=formonly.dl_field1,
              config="{Field 1 - Invisible:}",
              widget="label";
dl_field2=formonly.dl_field2,
              config="{Field 2 - Normal:}",
              widget="label";
dl_field3=formonly.dl_field3,
              config="{Field 3 - Invisible-Red:}",
              widget="label";
dl_header=formonly.dl_header,
              config="{Demo Application - Field Attribute INVISIBLE}",
              widget="label",
              color=bold Blue;
dl_guide=formonly.dl_guide,
              widget="label",
              wordwrap Compress,
              color=Cyan;

INSTRUCTIONS

DELIMITERS "[]"
```

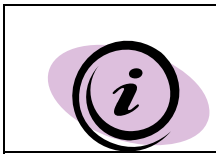
Example Screen:



KEY / KEYS

The field attribute KEY allows the user to define hotkey labels specifically for the given field in a dynamic toolbar.

The dynamic toolbar uses various form definitions, script options and 4GL functions to allow a toolbar to be created, altered or removed during a program's runtime.



This attribute is used to control the graphical toolbar and can for that reason, only be processed by graphical clients. Text mode clients ignore this attribute.

Toolbar Buttons

Toolbar buttons that have been defined in COMMAND KEY, ON ACTION or ON KEY statements are displayed on the top of a window in the form of a toolbar. The developer has the options to simply assign a key or action event with a text label, with an icon image, or with both. The order position number can also be specified optionally.

The buttons can be displayed permanently (Static True) or 'just in time' (Static False) when they are activated by a 4GL instruction.

The following order of precedence is used when editing hot-key button labels:

1. The key attributes of a .per file
2. The fgl_dialog_setkeylabel() function
3. The KEYS section in a .per file
4. The fgl_setkeylabel function
5. The fglprofile file

Defining Hot-Key Labels in fglprofile

You can define a label for each hot-key in the $QUERIXDIR/etc/fglprofile configuration file. The resource you use to do this is:

```
key."key".text = "label"
```

You can set the order of appearance of the hot-keys the right-hand frame of the application window, in fglprofile, with the following resource:

```
key."key_name".order
```

Defining Hot-Key Labels in the Form File

You can also define hot-key labels by editing the KEYS section of the form specification file, using syntax like that shown here:

```
KEYS
key = "new_key_label" ("path/icon") ORDER integer
```

Defining hot-key labels in the form file is discussed in details in the *KEYS Section* of this document.

Setting the KEY/ACTION Field Attribute

You can use the KEY and ACTION field attributes in the form specification file to change the parameters of the existing buttons. To do this, you have to specify the key or action name of the existing button and specify its new label, image, etc:

```
ATTRIBUTES
f001 – customer.customer_num,
   KEY F10 = "SEARCH" (icos/search_01.ico) ORDER 1,
   ACTION my_act = "CLEAR" (icos/clear_01.ico) ORDER 2,
   REVERSE
```

This code sample would cause the labels and icons of the F10 hot-key and *my_act* action button to be changed only when the cursor is in the field that corresponds to the f001 tag.

You should keep in mind that if you specify new properties for an existing button, *all* its default properties are ignored. It means that, for example, if a button has an icon by default, and you do not specify any icon in the KEY or ACTION form field attribute, the button will be displayed without the icon when the user positions the cursor to the corresponding field.

Using 4GL Functions

You can use calls on 4GL functions in source code modules, to change the labels on hot keys. This can be done both in functions that execute within a dialog with the user (such as INPUT, INPUT ARRAY, and CONSTRUCT), and those that are not specific to an active user dialog.

To find the current text of a hot-key label, use the fgl_dialog_getkeylabel(*keyname*) function. To set a hot-key label for a specific user dialog, use the fgl_dialog_setkeylabel(*keyname, label*). All of these functions are described in more detail in the Built-In Functions volume.

To find or set a hot-key label on a global, rather than dialog specific, basis, use the fgl_getkeylabel () and fgl_setkeylabel() functions.

The names of keys and actions are case-insensitive. Function Keys have a range from F1 – F1024.

Example Application:

Location:

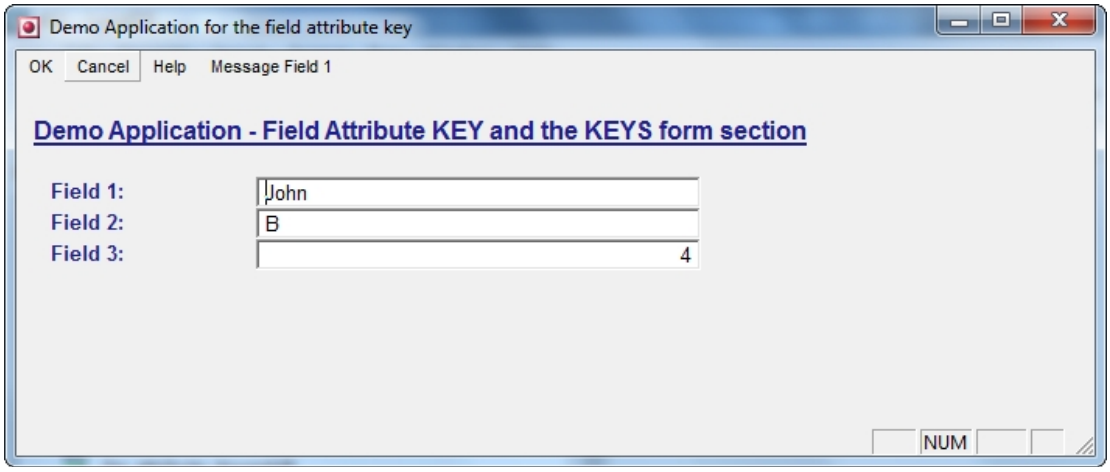
Project: forms
Program: project fm_attribute_key

Form .per Source:

```
DATABASE formonly

SCREEN
{
 [dl_header                                ]

  [dl_field1] [f1                  ]
  [dl_field2] [f2                  ]
  [dl_field3] [f3                  ]
[dl_guide                                 ]
[dl_guide                                 ]
}


ATTRIBUTES
f1=formonly.f1 type VARCHAR,
        KEY F13 = "Message Field 1" ORDER 11;
f2=formonly.f2 type VARCHAR,
        KEY F14 = "Message Field 2" ("icon16/info01.ico") ORDER 12;
f3=formonly.f3 type SMALLINT,
        ACTION act_message3 = "Message Field 3" ("icon16/info01.ico") ORDER 13;
dl_field1=formonly.dl_field1,
             config="{Field 1:}",
             widget="label";
dl_field2=formonly.dl_field2,
             config="{Field 2:}",
             widget="label";
dl_field3=formonly.dl_field3,
             config="{Field 3:}",
             widget="label";
dl_header=formonly.dl_header,
             config="{Demo Application - Field Attribute INCLUDE}",
             widget="label",
             color=bold Blue;
dl_guide=formonly.dl_guide,
             widget="label",
             wordwrap Compress,
             color=Cyan;

INSTRUCTIONS
```

```
DELIMITERS "[]"

KEYS
F2="OK" ORDER 1
F3="Cancel" ORDER 2
F5="Edit" ("icon16/record_edit01.ico") ORDER 5
F12="Exit" ("icon16/exit01.ico") ORDER 99
HELP="Help" ("icon16/help01.ico") ORDER 101
```

Example Screen:



LEFT

The left attribute aligns fields to the left regardless of the default formatting characteristics of the datatype being used.

For example, by default, 4GL left justifies all CHAR/VARCHAR fields, and right justifies all numeric datatypes. The example program which follows shows the right attribute, left attribute, center attribute and the default alignment of the datatypes.

Usage

The contents of the field ‘cust_name’ would be aligned left.

```
cust_name=  customer.cust_name,
            LEFT;
```

Example Program

Location:

Project: forms
Program: project fm_attribute_right, fm_attribute_left and fm_attribute_center

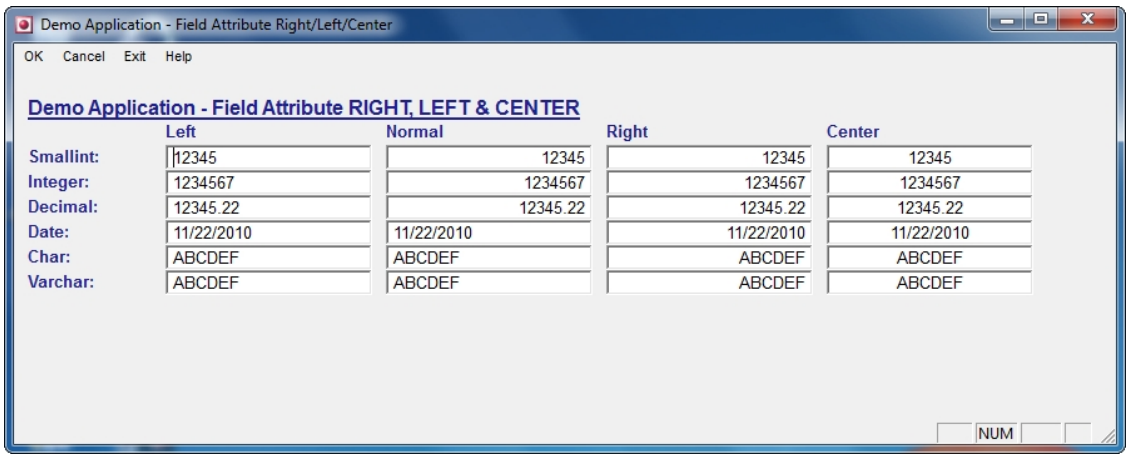
Form .per Source code:

```
DATABASE formonly

SCREEN
{
 [dl_header                                      ]
          [dl_sub_header1][dl_sub_header2][dl_sub_header3][dl_sub_header4]
 Smallint: [f0l          ][f0           ][f0r          ][f0c          ]
 Integer:  [f1l          ][f1           ][f1r          ][f1c          ]
 Decimal:  [f2l          ][f2           ][f2r          ][f2c          ]
 Date:     [f3l          ][f3           ][f3r          ][f3c          ]
 Char:     [f4l          ][f4           ][f4r          ][f4c          ]
 Varchar:  [f5l          ][f5           ][f5r          ][f5c          ]

[dl_guide                                                              ]
[dl_guide                                                              ]
}


ATTRIBUTES
f0r=formonly.f0r,
            right;
f1r=formonly.f1r,
```

```
                right;
f2r=formonly.f2r,
                right;
f3r=formonly.f3r,
                right;
f4r=formonly.f4r,
                right;
f5r=formonly.f5r,
                right;
f0l=formonly.f0l,
                left;
f1l=formonly.f1l,
                left;
f2l=formonly.f2l,
                left;
f3l=formonly.f3l,
                left;
f4l=formonly.f4l,
                left;
f5l=formonly.f5l,
                left;
f0c=formonly.f0c,
                center;
f1c=formonly.f1c,
                center;
f2c=formonly.f2c,
                center;
f3c=formonly.f3c,
                center;
f4c=formonly.f4c,
                center;
f5c=formonly.f5c,
                center;
f0=formonly.f0;
f1=formonly.f1;
f2=formonly.f2;
f3=formonly.f3;
f4=formonly.f4;
f5=formonly.f5;
dl_header=formonly.dl_header,
                config="{Demo Application - Field Attribute RIGHT, LEFT & CENTER}",
                widget="label";
dl_sub_header1=formonly.dl_sub_header1,
                config="Left",
                widget="label",
                color=bold Blue;
dl_sub_header3=formonly.dl_sub_header3,
                config="Right",
```

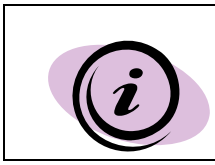
```
            widget="label",
            color=bold Blue;
dl_sub_header2=formonly.dl_sub_header2,
            config="Normal",
            widget="label",
            color=bold Blue;
dl_sub_header4=formonly.dl_sub_header4,
            config="Center",
            widget="label",
            color=bold Blue;
dl_guide=formonly.dl_guide,
            widget="label",
            wordwrap Compress,
            color=Cyan;

INSTRUCTIONS

DELIMITERS "[]"
```

Screen:



NOENTRY

The NOENTRY attribute prevents data entry in the field during an INPUT or INPUT ARRAY statement.



For fields with a noentry attribute, BEFORE and AFTER FIELD clauses will be ignored.

Usage

The following example illustrates the use of the NOENTRY attribute:

```
f1 = contact.cont_id, NOENTRY;
```

When the user enters data in the **contact** table, the **cont_id** column is not available because this SERIAL column gets its value from the database server during the INSERT statement.

The NOENTRY attribute does *not* prevent data entry into a field during a CONSTRUCT statement (e.g. for a search query).

Related Attribute

INVISIBLE

Example Program

Location:

Project: forms
Program: project fm_attribute_noentry

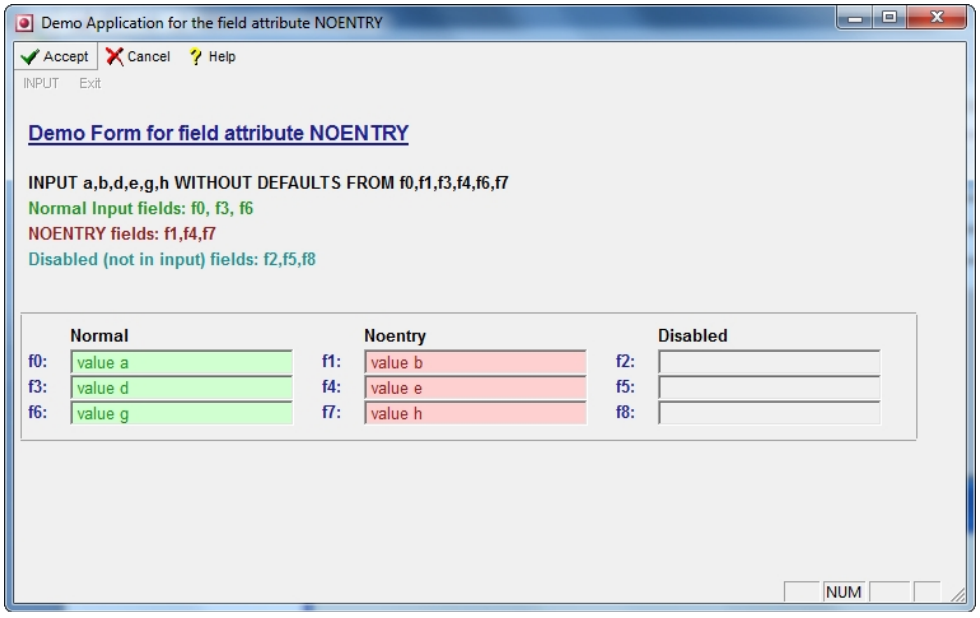
Form .per Source code:

```
DATABASE formonly

SCREEN
{
 [dl_header                          ]

 [f9                                                    ]
 [dl_1                         ]
 [dl_2                   ]
 [dl_3                              ]

\gp------------------------------------------------------------q  \g
```

```
\g|   \g[dl_normal      ]    [dl_noentry     ]    [dl_disabled    ] \g|  \g
\g|\gf0:[f0            ] f1:[f1            ] f2:[f2            ] \g|  \g
\g|\gf3:[f3            ] f4:[f4            ] f5:[f5            ] \g|  \g
\g|\gf6:[f6            ] f7:[f7            ] f8:[f8            ] \g|  \g
\gb-------------------------------------------------------------d  \g
}


ATTRIBUTES
f0=formonly.f0,
             color=green;
f1=formonly.f1,
             noentry,
             color=red;
f2=formonly.f2,
             color=cyan;
f3=formonly.f3,
             color=green;
f4=formonly.f4,
             noentry,
             color=red;
f5=formonly.f5,
             color=cyan;
f6=formonly.f6,
             color=green;
f7=formonly.f7,
             noentry,
             color=red;
f8=formonly.f8,
             color=cyan;
dl_header=formonly.dl_header,
             config="{Demo Form for field attribute NOENTRY}",
             widget="label",
             color=bold blue underline;
dl_1=formonly.dl_1,
             config="{Normal Input fields: f0, f3, f6}",
             widget="label",
             color=bold Green;
dl_3=formonly.dl_3,
             config="{Disabled (not in input) fields: f2,f5,f8}",
             widget="label",
             color=bold Cyan;
dl_2=formonly.dl_2,
             config="{NOENTRY fields: f1,f4,f7}",
             widget="label",
             color=bold Red;
f9=formonly.f9,
             config="{INPUT a,b,d,e,g,h WITHOUT DEFAULTS FROM f0,f1,f3,f4,f6,f7}",
```

```
            widget="label",
            color=bold;
dl_normal=formonly.dl_normal,
            config="Normal",
            widget="label",
            color=bold;
dl_noentry=formonly.dl_noentry,
            config="Noentry",
            widget="label",
            color=bold;
dl_disabled=formonly.dl_disabled,
            config="Disabled",
            widget="label",
            color=bold;

INSTRUCTIONS

DELIMITERS "[]"
```

Screen:



NOPROMPT

This option affects Text Mode clients only.
GUI clients support in-field scrolling by default and this behaviour can be turned off by using the field attribute 'noscroll'

Unlike Informix 4GL, you can control the in-field scrolling behaviour. The options are:

Data entered in a field during input is:

- limited to field width (not variable length) – INFORMIX behaviour
- limited to variable length – In Field scrolling
- limited to variable with prompt – If the field width is smaller than the variable, a prompt line will be displayed on the first line to enter and view the entire string

Classic Informix 4GL Field Non-Scrolling behaviour

In character mode rendering, the default behaviour of Informix 4GL is to restrict the number of characters the user can input in a field to the size of the field. For example, considering the following form:

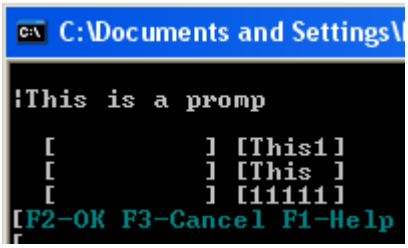
```
SCREEN {
[f1    ]
}
ATTRIBUTES
f1 = formonly.f1 TYPE CHAR;
```

The field f1 is 5 characters wide. This means that a maximum of five characters may be input into this field by default.

Again, the behaviour depends on the form compile flag.

Lycia Field Non-Scrolling – Using Prompts

Queirix4GL does not enforce this restriction, and can allow the user to input as many characters as the 4GL variable bound to the field will allow. If a field is entered where the variable is larger than the field width, a prompt area will be displayed in the first line of the window.



Querix4GL Field Scrolling – Field Attribute NoPrompt

If a field has the attribute 'NoPrompt', the user can enter data in the entire length of the underlying variable; the cursor will scroll within in the field. For example, if the field f1 is used to enter data for a CHAR(20), then it is possible to enter up to 20 characters if the SCROLL attribute is used for the field.

The noprompt attribute prevents the prompt field from being created, and allows the input field to scroll instead.

```
SCREEN {
[f1    ]
}
ATTRIBUTES
f1 =          formonly.f1 TYPE CHAR,
              NOPROMPT;
```

Related Attribute

SCROLL

Example Program

Location:

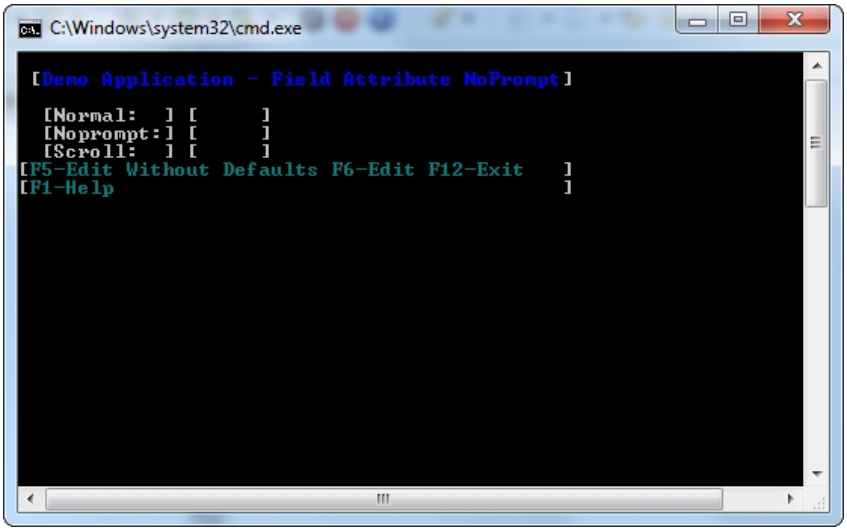
Project: forms
Program: project fm_attribute_noprompt

Form .per Source code:

```
DATABASE formonly

SCREEN
{
 [dl_header                                ]
```

```
  [dl_field1] [f1   ]
  [dl_field2] [f2   ]
  [dl_field3] [f3   ]
[dl_guide                                   ]
[dl_guide                                   ]
}


ATTRIBUTES
f1=formonly.f1;
f2=formonly.f2,
             noprompt;
f3=formonly.f3,
             scroll;
dl_field1=formonly.dl_field1,
             config="{Normal:}",
             widget="label";
dl_field2=formonly.dl_field2,
             config="{Noprompt:}",
             widget="label";
dl_field3=formonly.dl_field3,
             config="{Scroll:}",
             widget="label";
dl_header=formonly.dl_header,
             config="{Demo Application - Field Attribute NoPrompt}",
             widget="label",
             color=bold Blue;
dl_guide=formonly.dl_guide,
             widget="label",
             wordwrap Compress,
             color=Cyan;

INSTRUCTIONS

DELIMITERS "[]"
```

Screen:


C:\Windows\system32\cmd.exe
[Demo Application - Field Attribute NoPrompt]
[Normal:] [ ]
[Noprompt:] [ ]
[Scroll:] [ ]
[F5-Edit Without Defaults F6-Edit F12-Exit]
[F1-Help]

NOSCROLL

The noscroll attribute limits the user input to the length of the input field, regardless of the size of the underlying variable. This means that if you wanted to force the user to input a set number of characters, for a password for example, you could set the input field to that number and give it the noscroll attribute.

Usage:

The user could scroll in the field one but not scroll (limited to field boundary) in field2.

```
Field1=     formonly.field1,
            SCROLL;
Field2=     formonly.field2,
            NOSCROLL;
```

Example Program

```
DATABASE formonly

SCREEN
{
[dl_header                              ]

Scroll:   [field1]
NoScroll: [field2]
}


ATTRIBUTES
field1=formonly.field1,
        scroll,
        color=blue;
field2=formonly.field2,
             noscroll,
        color=red;

dl_header=formonly.dl_header,
             config="{Demo Field Attribute Scroll & NoScroll}",
             widget="label",
             color=bold Blue underline;

INSTRUCTIONS

DELIMITERS "[]"
```

PASSWORD

The password attribute causes a string of asterisks to be displayed in the field in place of the actual field content. This happens in input and display mode.

Usage:

The following field ‘passw’ would display all characters in display and input as asterisk ‘*’ symbols.

```
passw=security.passw,
              password;
```

Program Example:

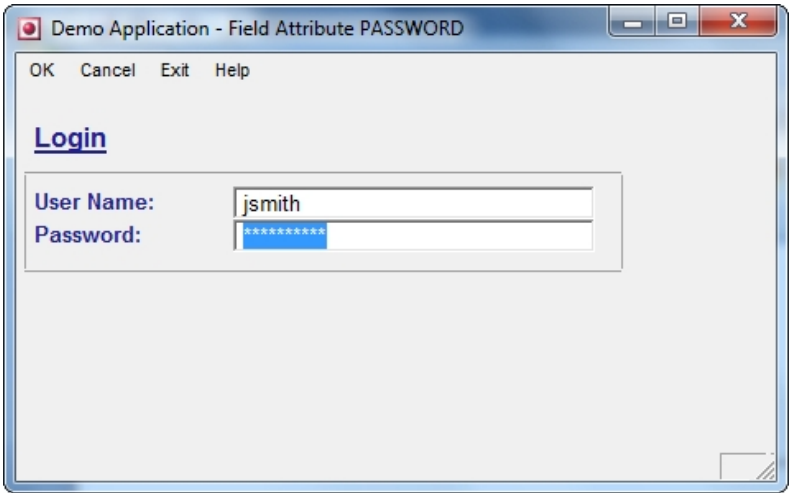
Location:

Project: forms
Program: project fm_attribute_password

Form Code .per Source Code:

```
DATABASE formonly

SCREEN
{
 [dl_header]
\gp--------------------------------q\g
\g|\gUser\g \gName: [f0                ]\g|\g
\g|\gPassword:  [f1                ]\g|\g
\gb--------------------------------d\g
}


ATTRIBUTES
f0=formonly.f0,
              autonext;
f1=formonly.f1,
              password;
dl_header=formonly.dl_header,
              config="Login",
              widget="label",
              color=bold Blue;

INSTRUCTIONS

DELIMITERS "[]"
```

Sample Screen:



PICTURE

The PICTURE attribute specifies a character pattern for data entry in a text field and prevents entry of values that conflict with the specified pattern.

Within the PICTURE = *"format-string"* syntax, *format-string* is a character string that describes the pattern for data entry. The string must be placed between speech marks.

Usage

A *format-string* value can include literals and these three special symbols:

- A representing any letter
- # representing any digit
- X representing any character

Querix4GL treats any other character in *format-string* as a literal. The cursor will pass over any literals during data entry. Querix4GL displays the literal characters in the display field and leaves blanks elsewhere.

For example, note the following field specification:

```
c10 = customer.phone,
picture = "###-###-####x#####";
```

It displays these symbols in the **customer.phone** field before data entry:

```
[   -   -    x    ]
```

If the user attempts to enter a character that conflicts with *format-string*, the terminal beeps, and Querix4GL does not display that character.

The *format-string* value must fill the entire width of the display field. The PICTURE attribute, however, does not require data entry into the entire field. It only requires that whatever characters are entered conform to *format-string*.

When PICTURE specifies input formats for DATETIME or INTERVAL fields, QFORM does not check the syntax of *format-string*, but your form will work if the syntax is correct. Any error in *format-string*, however, such as an incorrect field separator, produces a runtime error.

As another example, suppose you specify a field for part numbers like this:

```
f1 = part_no, picture = "AA#####-AA(X)";
```

Lycia accepts any of the following inputs:

```
LF49367-BB(*)
```

```
TG38524-AS(3)
YG67489-ZZ(D)
```

The user does not enter the hyphen or the parentheses, but Lycia includes them in the string that it passes to the program variable.

The PICTURE attribute is not affected by GLS environment variables because it only formats character information.

Editing keys during data entry in character mode

In character mode, Lycia supports pressing **CONTROL-D** and **CONTROL-X** in fields that specify the PICTURE attribute:

- **CONTROL-D** deletes characters from the current cursor position to the end of the field.
- **CONTROL-X** deletes the current character.

Related Attribute

FORMAT

Example Program

Location:

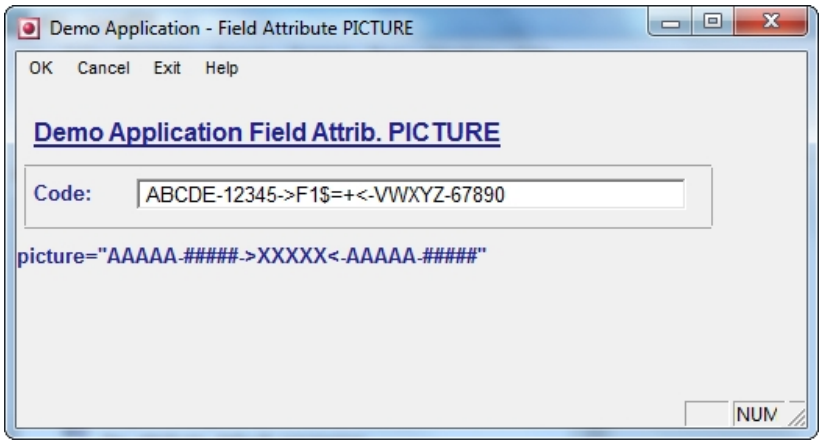
Project: forms
Program: project fm_attribute_picture

Form .per Source

```
DATABASE formonly

SCREEN
{
 [dl_header                                ]
\gp----------------------------------------q  \g
\g|\gCode: [f0                              ]\g|  \g
\gb----------------------------------------d  \g
 [dl_field2                                ]
}


ATTRIBUTES
f0=formonly.f0,
     picture="AAAAA-#####->XXXXX<-AAAAA-#####";
dl_header=formonly.dl_header,
     config="{Demo Application Field Attrib. PICTURE}",
     widget="label",
     color=bold Blue;
```

```
dl_field2=formonly.dl_field2,
     config="picture=""AAAAA-#####->XXXXX<-AAAAA-#####""",
     widget="label";

INSTRUCTIONS

DELIMITERS "[]"
```

Sample Screen



PROGRAM

The PROGRAM attribute can specify an external application program to work with screen fields of data type TEXT or BYTE.

Within the PROGRAM = *"command"* syntax, *command* can be either a string, or the name of a shell script, that invokes an editing program. The *command* string must be placed between speech marks.

Usage

Character mode

You can assign the PROGRAM attribute to a BYTE or TEXT field to call an external program to work with the BYTE or TEXT values. Users can invoke the external program by pressing the exclamation mark (!) key while the screen cursor is in a BYTE or TEXT field. The external program then takes over control of the screen. When the user exits from the external program, the form is redisplayed with any display attributes besides PROGRAM in effect. For example, this field description designates **vi** as the external editor of a multiple-segment TEXT field that also has the WORDWRAP attribute:

```
fbody=formonly.body type TEXT,comments="E-Mail Body - Message",program="vi";
```

Here the WORDWRAP attribute specifies that as much of the TEXT value as possible be displayed in successive segments of the multiple-segment field when the form displays a value in the field, but the WORDWRAP editor cannot edit a TEXT value.

If the cursor enters the field whose tag is **fbody** in the same example and presses the "!" key, the form is cleared from the screen. Now the user can run the **vi** utility to view or edit the TEXT value. When the editing session ends, the form is restored on the screen, and control returns to the 4GL application.

When a display field is of data type TEXT, but the screen cursor is not in that field, the 4GL application can display as many of the leading characters of a TEXT data value as can fit in the field. (For BYTE fields, Lycia displays <BYTE value> in the field.) This behaviour is independent of the PROGRAM attribute.

GUI mode

This works in much the same way as when in character mode, except instead of pressing the exclamation mark to invoke the external program, the user needs to click on the input field. This will then open the program associated with the given file extension on your system. If there is no default program for this file type defined, a dialog box will open asking you to choose a program to open the file.

Default Editors

If a user moves the cursor to a TEXT field and presses the exclamation point key in the first character position of the field, 4GL attempts to invoke an external program. The program invoked for a TEXT field is chosen from among the following programs, in *descending* order of priority:

- The program (if any) identified by the PROGRAM ="command" attribute specification for the field
- The program (if any) named in the **DBEDIT** environment variable
- The default editor, which depends on the host operating system

Specify the editor to use with the **DBEDIT** environment variable; this variable should contain the name of a UNIX application such as **vi** or **emacs**. When the user exits from the editor, control returns to the 4GL screen.

4GL applications that display or modify a value in a BYTE field must use the PROGRAM attribute explicitly to assign an editor. For BYTE fields, the default editor is not called, and the **DBEDIT** variable is not examined.

The Command String

Before invoking the program, your application copies the BYTE or TEXT field to a temporary disk file. It then issues a system command composed of the *command* string that you specify after the PROGRAM keyword, followed by the name of the temporary file.

The *command* string can be longer than a single word. You can add additional command parameters. The *command* string can also be the name of a shell script, so that you can initiate a whole series of actions.

Your Querix4GL program needs to execute an INSERT or UPDATE statement by using the appropriate program variables after input is terminated. For example, you would use a statement similar to that which follows:

```
INSERT INTO contact (cont_picture) VALUES (l_cont_picture)
UPDATE contact SET cont_picture = l_cont_picture
```

Example Application:

Location:

Project: forms
Program: project fm_attribute_program_notepad

Form .per Source:

```
DATABASE formonly

SCREEN
{
 [dl_header                          ]
```

```
 [dl_field1           ]
 [f1                                ]

 [dl_field2           ]
 [f2                                ]

[dl_guide                             ]
[dl_guide                             ]
}


ATTRIBUTES
f1=formonly.f1 type TEXT,
     program="notepad";
f2=formonly.f2 type BYTE,
     program="notepad";
dl_header=formonly.dl_header,
     config="{Field Attribute PROGRAM - notepad}",
     widget="label",
     color=bold Blue;
dl_field2=formonly.dl_field2,
     config="{Field f1 - Type Byte:}",
     widget="label",
     color=bold Blue;
dl_field1=formonly.dl_field1,
     config="{Field f1 - Type TEXT:}",
     widget="label",
     color=bold Blue;
dl_guide=formonly.dl_guide,
     widget="label",
     wordwrap Compress,
     color=Cyan;

INSTRUCTIONS

DELIMITERS "[]"
```

REQUIRED

The REQUIRED attribute forces the user to enter data in the field during an INPUT or INPUT ARRAY statement (field value cannot be NULL/Empty).

Usage

The REQUIRED keyword is effective only when the field name appears in the list of screen fields of an INPUT or INPUT ARRAY statement. For example, suppose the ATTRIBUTES section includes the following field description:

```
f1 =          invoice.inv_number,
              REQUIRED;
```

Because of the REQUIRED specification, Lycia requires the entry of a purchase order value when the form is used to collect information for a new order.

You can also specify a default value for a REQUIRED field. This means the required condition is true if the user does not remove the default inserted value.

Related Attribute

NOENTRY

Example Application:

Location:

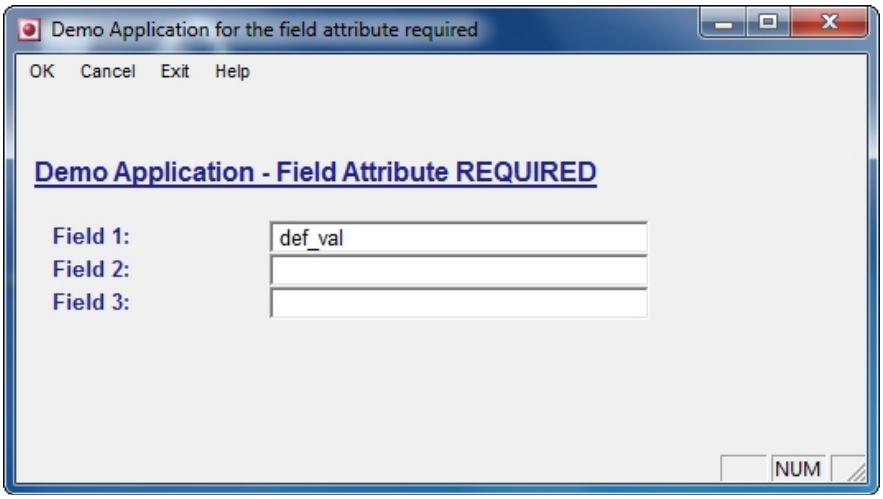
Project: forms
Program: project fm_attribute_required

Form .per Source:

```
DATABASE formonly

SCREEN
{
 [dl_header                                ]

  [dl_field1] [f1                  ]
  [dl_field2] [f2                  ]
  [dl_field3] [f3                  ]
}


ATTRIBUTES
f1=formonly.f1,
              required;
```

```
f2=formonly.f2,
              required;
f3=formonly.f3,
              required;
dl_field1=formonly.dl_field1,
              config="{Field 1:}",
              widget="label";
dl_field2=formonly.dl_field2,
              config="{Field 2:}",
              widget="label";
dl_field3=formonly.dl_field3,
              config="{Field 3:}",
              widget="label";
dl_header=formonly.dl_header,
              config="{Demo Application - Field Attribute Required}",
              widget="label",
              color=bold Blue;

INSTRUCTIONS

DELIMITERS "[]"
```

Example Screen:



REVERSE

The REVERSE attribute displays any value in the field in reverse video (dark characters in a bright field).

Usage

The following example specifies that a field linked to the **contact.cont_id** column displays data in reverse (sometimes called *inverse*) video:

```
f1 = contact.cont_id, REVERSE;
```

The REVERSE attribute disables any other COLOR attribute for the same field.

Related Attribute

COLOR

Example Program

Location:

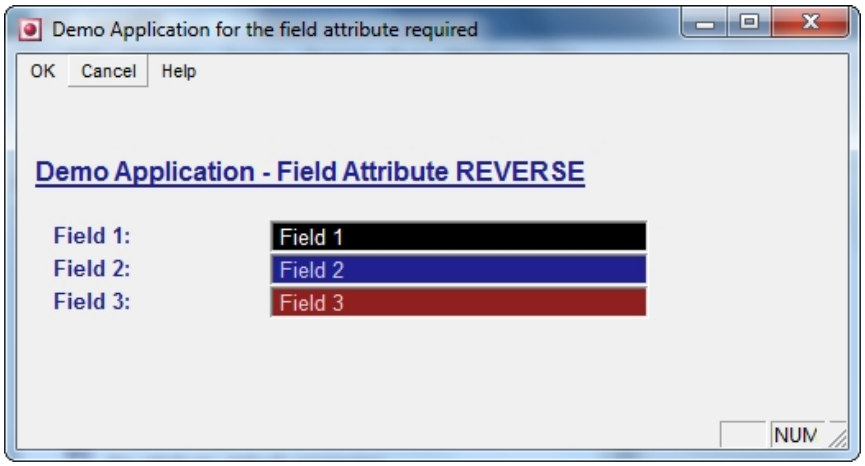
Project: forms
Program: project fm_attribute_reverse

Form .per Source Code:

```
DATABASE formonly

SCREEN
{
 [dl_header                                ]

  [dl_field1] [f1                  ]
  [dl_field2] [f2                  ]
  [dl_field3] [f3                  ]
}


ATTRIBUTES
f1=formonly.f1,
             reverse;
f2=formonly.f2,
             color=blue reverse;
f3=formonly.f3,
             color=red reverse;
dl_field1=formonly.dl_field1,
             config="{Field 1:}",
             widget="label";
```

```
dl_field2=formonly.dl_field2,
              config="{Field 2:}",
              widget="label";
dl_field3=formonly.dl_field3,
              config="{Field 3:}",
              widget="label";
dl_header=formonly.dl_header,
              config="{Demo Application - Field Attribute Required}",
              widget="label",
              color=bold Blue;

INSTRUCTIONS

DELIMITERS "[]"
```

Screen Example:



RIGHT

The right attribute aligns the field contents right regardless of the default formatting characteristics of the datatype being used.

For example, by default, Lycia left justifies all CHAR/VARCHAR fields, and right justifies all numeric datatypes. The example program which follows shows the right attribute, the left attribute, the center attribute and the default alignment of the datatypes.

Usage:

The contents of the field ‘cust_name’ would be aligned right.

```
Cust_name=    customer.cust_name,
              RIGHT;
```

Example Program

Location:

Project: forms
Program: project fm_attribute_right, fm_attribute_left and fm_attribute_center

Form .per Source code:

```
DATABASE formonly

SCREEN
{
 [dl_header                                        ]
           [dl_sub_header1][dl_sub_header2][dl_sub_header3][dl_sub_header4]
 Smallint: [f0l           ][f0            ][f0r           ][f0c           ]
 Integer:  [f1l           ][f1            ][f1r           ][f1c           ]
 Decimal:  [f2l           ][f2            ][f2r           ][f2c           ]
 Date:     [f3l           ][f3            ][f3r           ][f3c           ]
 Char:     [f4l           ][f4            ][f4r           ][f4c           ]
 Varchar:  [f5l           ][f5            ][f5r           ][f5c           ]

[dl_guide                                                               ]
[dl_guide                                                               ]
}


ATTRIBUTES
f0r=formonly.f0r,
              right;
f1r=formonly.f1r,
```

```
                right;
f2r=formonly.f2r,
                right;
f3r=formonly.f3r,
                right;
f4r=formonly.f4r,
                right;
f5r=formonly.f5r,
                right;
f0l=formonly.f0l,
                left;
f1l=formonly.f1l,
                left;
f2l=formonly.f2l,
                left;
f3l=formonly.f3l,
                left;
f4l=formonly.f4l,
                left;
f5l=formonly.f5l,
                left;
f0c=formonly.f0c,
                center;
f1c=formonly.f1c,
                center;
f2c=formonly.f2c,
                center;
f3c=formonly.f3c,
                center;
f4c=formonly.f4c,
                center;
f5c=formonly.f5c,
                center;
f0=formonly.f0;
f1=formonly.f1;
f2=formonly.f2;
f3=formonly.f3;
f4=formonly.f4;
f5=formonly.f5;
dl_header=formonly.dl_header,
                config="{Demo Application - Field Attribute RIGHT, LEFT & CENTER}",
                widget="label";
dl_sub_header1=formonly.dl_sub_header1,
                config="Left",
                widget="label",
                color=bold Blue;
dl_sub_header3=formonly.dl_sub_header3,
                config="Right",
```

```
            widget="label",
            color=bold Blue;
dl_sub_header2=formonly.dl_sub_header2,
            config="Normal",
            widget="label",
            color=bold Blue;
dl_sub_header4=formonly.dl_sub_header4,
            config="Center",
            widget="label",
            color=bold Blue;
dl_guide=formonly.dl_guide,
            widget="label",
            wordwrap Compress,
            color=Cyan;

INSTRUCTIONS

DELIMITERS "[]"
```

SCROLL

The scroll attribute allows the user to input data to the full extent of the underlying variable, as opposed to the length of the input field.

Usage:

The user could scroll in the field one but not scroll (limited to field boundary) in field2.

```
Field1=     formonly.field1,
      SCROLL;
Field2=     formonly.field2,
      NOSCROLL;
```

Example Program

Location:

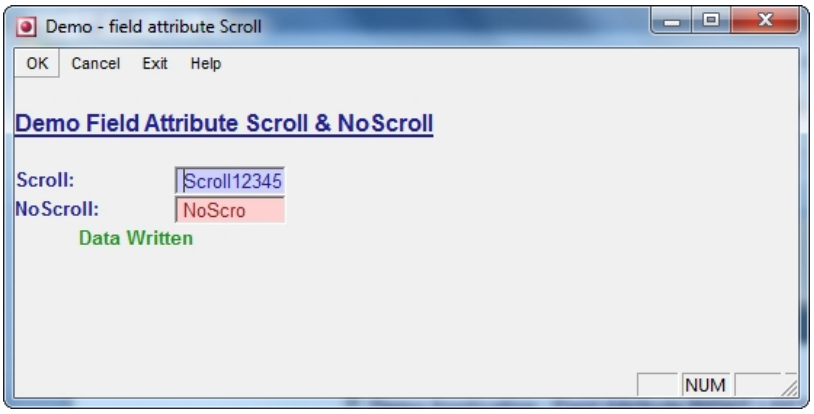
Project: forms
Program: project fm_attribute_scroll and fm_attribute_noscroll

Form .per Source code:

```
DATABASE formonly

SCREEN
{
[dl_header                              ]

Scroll:   [field1]
NoScroll: [field2]
}


ATTRIBUTES
field1=formonly.field1,
        scroll,
        color=blue;
field2=formonly.field2,
             noscroll,
        color=red;

dl_header=formonly.dl_header,
             config="{Demo Field Attribute Scroll & NoScroll}",
             widget="label",
             color=bold Blue underline;
```

```
INSTRUCTIONS

DELIMITERS "[]"
```

Screen:



UPSHIFT

During data entry in a character field, the UPSHIFT attribute converts lowercase letters to uppercase letters and in the 4GL program variable that stores the contents of that field.

Usage

Because uppercase and lowercase letters have different code-set values, storing all character strings in one or the other format can simplify sorting and querying a database.

The following example includes UPSHIFT in the attribute list of a field. Any data entered into this field during an input will be converted to ‘upper case letters’:

```
f1 =          formonly.user_id,
              UPSHIFT;
```

Because of the UPSHIFT attribute, Lycia enters uppercase characters in the **user_id** field regardless of the case used to enter them.

Related Attribute

DOWNSHIFT

Example Program

Location:

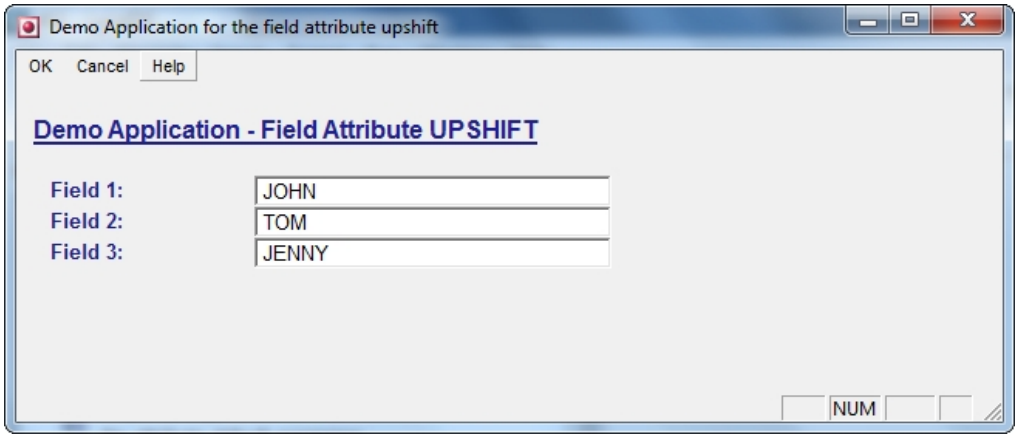
Project: forms
Program: fm_attribute_upshift

Form .per Source code:

```
DATABASE formonly

SCREEN
{
 [dl_header                                ]

   [dl_field1] [f1                  ]
   [dl_field2] [f2                  ]
   [dl_field3] [f3                  ]
}


ATTRIBUTES
f1=formonly.f1,
              upshift;
f2=formonly.f2,
```

```
                upshift;
    f3=formonly.f3,
                upshift;
    dl_field1=formonly.dl_field1,
                config="{Field 1:}",
                widget="label";
    dl_field2=formonly.dl_field2,
                config="{Field 2:}",
                widget="label";
    dl_field3=formonly.dl_field3,
                config="{Field 3:}",
                widget="label";
    dl_header=formonly.dl_header,
                config="{Demo Application - Field Attribute Required}",
                widget="label",
                color=bold Blue;

    INSTRUCTIONS

    DELIMITERS "[]"
```

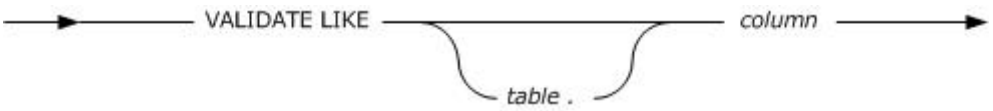
Screen:



VALIDATE LIKE

The VALIDATE LIKE attribute instructs Lycia to validate the data entered in the field by using the validation rules that the **upscol** utility assigned to the specified database column in the **syscolval** table. It works in much the same way as the INCLUDE attribute, except that it uses a database instead of an input string.

Syntax Diagram



Element	Description
column	This is the name of a column in *table* or, if *table* is left out, the unique identifier of a column in one of the tables declared in the TABLES section.
table	This is the unqualified name or alias of a table, synonym, or view, declared in the TABLES section. This is only needed if several tables all have a column of the same name, or if the table is an external, or external, distributed table.

Usage

This attribute is equivalent to listing all the attributes that you have assigned to *table.column* in the **syscolval** table. “Default Attributes” describes the **syscolval** table and the effects of this table in an ANSI-compliant database. The following example assigns the default attributes of the **pay_methods.pay_name** column to a FORMONLY field:

```
s13 = FORMONLY.state, VALIDATE LIKE pay_methods.pay_name;
```

The restrictions on the DISPLAY LIKE attribute also apply to this attribute. You do not need the VALIDATE LIKE attribute if *table.column* is the same as *field name*. You cannot specify a column of data type BYTE as *table.column*. Even if all of the fields in the form are FORMONLY, this attribute requires QFORM to access the database that contains *table*.

Related Attribute

DISPLAY LIKE

VERIFY

The VERIFY attribute requires users to enter data in the field twice to reduce the probability of erroneous data entry.

Usage

Because some data is critical, this attribute supplies an additional step in data entry to ensure the integrity of your data. After the user enters a value into a VERIFY field and presses **RETURN**, Querix4GL erases the field and requests re-entry of the value. The user must enter exactly the same data each time, character for character: 15000 is not exactly the same as 15000.00.

For example, if you specify a field for a password in the following way, Lycia requires entry of exactly the same data twice:

```
f1 = user.password, PASSWORD, VERIFY;
```

An error message appears if the user does not enter the same keystrokes.

The VERIFY attribute takes effect while INPUT, INPUT ARRAY, or UPDATE statements of 4GL are executing. It has no effect on CONSTRUCT statements.

Related Attributes

INCLUDE, REQUIRED, VALIDATE LIKE

Example Program

Location:

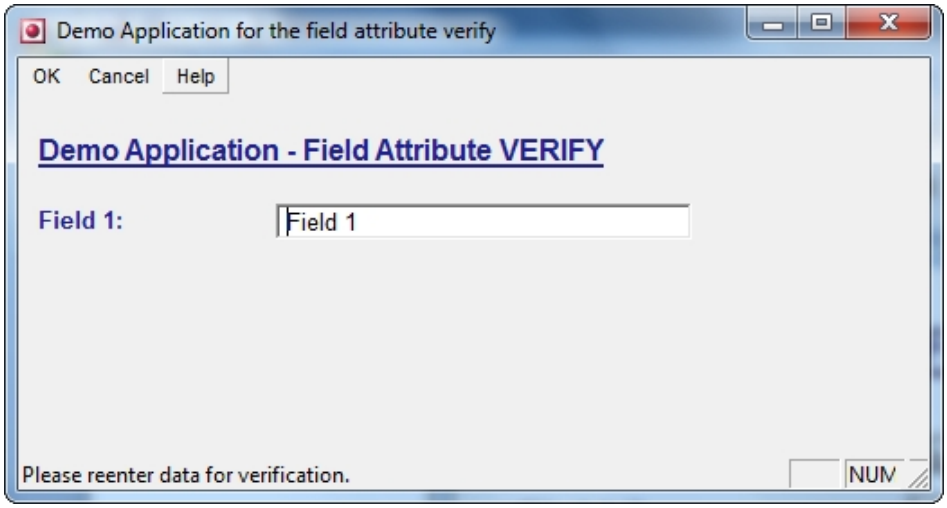
Project: forms
Program: fm_attribute_verify

Form .per Source code:

```
DATABASE formonly

SCREEN
{
 [dl_header                                ]

 [dl_field1] [f1                  ]
}


ATTRIBUTES
f1=formonly.f1,
             verify;
dl_field1=formonly.dl_field1,
```

```
            config="{Field 1:}",
            widget="label";
dl_header=formonly.dl_header,
            config="{Demo Application - Field Attribute VERIFY}",
            widget="label",
            color=bold Blue;

INSTRUCTIONS

DELIMITERS "[]"
```

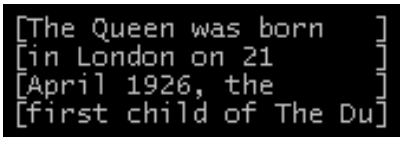
Screen:



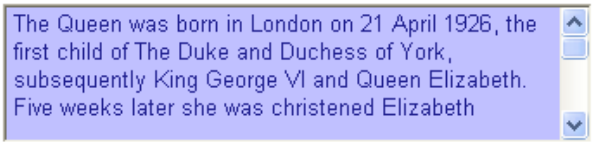
WORDWRAP

In a multiple-segment field, the WORDWRAP attribute enables a multiple-line editor. This editor can *wrap* long character strings to the next line of a multiple-segment field for data entry, data editing, and data display.

Text mode clients render each line as an individual; GUI clients show one single large area, which is a 'memo-notes' style field.



A word-wrap field seen in a text mode client



The same word-wrap field seen in a GUI client

Usage

If the same field tag is repeated in two or more lines in the screen layout, this attribute instructs Lycia to treat all the instances of that field tag as successive segments of a *multiple-segment field* (described in "Multiple-Segment Fields"). These fields can display data strings that are too long to fit on a single line of the screen form. For example, the following excerpt from a form specification file shows a VARCHAR field linked to the **history** column in the **employee** table.

```
history [f002 ]
[f002 ]
[f002 ]
attributes
f002 = employee.history, WORDWRAP COMPRESS;
```

Lycia replaces each set of multiple-segment fields with a single WORDWRAP field of a rectangular shape. The COMPRESS keyword option is applied to this field, and the delimiters are replaced with blank spaces.

When a variable is bound to the WORDWRAP field during INPUT, only the number of characters allowed by the bound variable can be entered. If necessary, text in the field scrolls to allow the full number of characters to be entered. Data compression takes place before storage in the bound variable.

If the lines of the multiple-segment field are not contiguous or if the field has an irregular shape, the WORDWRAP field that results is based on the maximum height and width of the multiple-segment field as a unit.

The resulting WORDWRAP field can overlap or be overlapped by labels or individual form fields. To prevent such unpredictable effects, consolidate the segments of multiple-segment fields into rectangular shapes.

Data Display with WORDWRAP

If a CHAR, VARCHAR, or TEXT value is displayed in a WORDWRAP field, Lycia puts the first data character in the first character position of the first segment and displays consecutive data characters in successive positions to the right.

If the entire data string is too long to fit in the first field segment, Lycia continues the display in the next field segment, dividing the data string at blank characters. This process continues until all the field segments are filled or until the end of the data string is reached.

GUI clients render wordwrap fields like windows style memo fields. The edit and navigation behaviour is just like any other native windows application. All additional wordrwap information such as compress, edit keys and navigation which follows below only apply for text mode clients.

Text Client Data Entry and Editing with WORDWRAP

When text is entered into a multiple-segment WORDWRAP field, Lycia breaks character strings into segments at blanks (if it can) and pads field segments with blanks to the right. Where possible, contiguous nonblank substrings (*words*) within a string are not broken at field segment boundaries.

When keyboard input reaches the end of a line, the multiple-line editor brings the current word down to the next field segment and moves text down to subsequent lines as necessary. (The next field segment is determined by the left-to-right, top-to-bottom order of field segments within the screen layout.) When the user deletes text, the editor pulls words up from lower field segments whenever it can.

When non-default locales are used, WORDWRAP fields can process non-ASCII characters that the locale supports. For the special case of Japanese locales, WORDWRAP fields support Kinsoku processing.

Multi Line Editor with WORDWRAP (text mode)

The WORDWRAP attribute displays a TEXT field so that it fits into the form without any field segments that begin with a blank. For a TEXT field, the WORDWRAP attribute only affects how the value is displayed; WORDWRAP does not enable the multiple-line editor. To let users edit a TEXT field, you must use the PROGRAM attribute to indicate the name of an external editor.

Displaying Program Variables with WORDWRAP

Text in WORDWRAP fields can include printable ASCII characters as well as the **TAB** (ASCII 9) character and the new line (ASCII 10) character. These characters are retained in the program variable. Other

nonprintable characters might result in runtime errors. The **TAB** character aligns the display at the next tab stop, and the new line character continues the display at the start of the next line. By default, tab stops are in every eighth column, beginning at the left edge of the field. In non-English locales, WORDWRAP fields can include printable characters that the locale supports.

Ordinarily, the length of the variable should not be greater than the total length of all the field segments. If the data string is longer than the field (or if too much padding is required for WORDWRAP), Lycia fills the field and discards the excess data. This process displays a long variable in summary form. If, however, a truncated value is used to update the database, characters are lost.

The editor distinguishes between *intentional* blanks (from the database or typed by the user) and *editor* blanks (inserted at the ends of lines for wordwrap or to align after a new line character). Intentional blanks are retained as part of the data. Editor blanks are inserted and deleted automatically as required. When designing a multiple-segment field, allow room for editor blanks, over and above the data length. The expected number of editor blanks is half the length of an average word per segment. Text that requires more space than you expect might be truncated after the final field segment.

The COMPRESS and UNCOMPRESS Options

The COMPRESS keyword prevents blanks produced by the editor from being included in the program variable.

COMPRESS is applied by default and can cause truncation to occur if the sum of intentional characters exceeds the field or column size. Because of editing blanks in the WORDWRAP field, the stored value might not correspond exactly to its multiple-line display, so a 4GL report generally cannot display the data in identical form.

To suppress COMPRESS, specify UNCOMPRESS after the WORDWRAP keyword. This option causes any editor blanks to be saved when the WORDWRAP string is saved in a database column, in a variable, or in a file.

Default value is COMPRESS

In the following fragment of a form specification file, a CHAR value in the column **charcolm** is displayed in the multiple-segment field whose tag is **mlf**:

```
SCREEN SIZE 24 by 80
{
Enter text:
[mlf ]
[mlf ]
. . .
[mlf ]
[mlf ]
}
TABLES tablet . . .
ATTRIBUTES
mlf = tablet.charcolm, WORDWRAP COMPRESS;
```

If the data string is too long to fit in the first line, successive segments are displayed in successive lines, until all of the lines are filled or until the last text character is displayed (whichever happens first).

If the form is used to insert data into **tablet.charcolm**, the keyword COMPRESS specifies that 4GL will not store editor blanks.

WORDWRAP Editing Keys

GUI Mode:

When the user works in a wordwrap field using a GUI client, the usage is identical to any other windows driven application. The cursor movement behaviour differs from the text client only in the following ways:

- **TAB/ENTER** Moves the cursor to next field
- **CTRL-TAB** inserts a tab
- **CTRL-ENTER** inserts a new line

The rest of the keys are the same.

Text Client:

When data is entered or updated in a WORDWRAP field, the user can use keys to move the screen cursor over the data and to insert, delete, and type over the data. The cursor never pauses on editor blanks.

The editor has two modes, *insert* (to add data at the cursor) and *typeover* (to replace existing data with entered data). You cannot overwrite a new line character. If the cursor in typeover mode encounters a new line character, the cursor mode automatically changes to insert, *pushing* the new line character to the right. Some keystrokes behave differently in the two modes.

When the cursor first enters a multiple-segment field, it is positioned on the first character of the first field segment, and the editing mode is set to *typeover*. The cursor movement keys are as follows:

- **RETURN** leaves the entire multiple-segment field and goes to the first character of the next field.
- **BACKSPACE** or **LEFT ARROW** moves the cursor left one character, unless at the left edge of a field segment. From the left edge of the first segment, these keys either move the cursor to the first character of the preceding field or only beep, depending on whether INPUT WRAP is in effect. (Input wrap mode is controlled by the OPTIONS statement.) From the left edge of a lower field segment, these keys move the cursor to the right-most intentional character of the previous field segment.
- **RIGHT ARROW** moves the cursor right one character, unless at the right-most intentional character in a segment. From the right-most intentional character of the last segment, this key either moves the cursor to the first character of the next field or only beeps, depending on INPUT WRAP mode. From the right-most intentional character of a higher segment, this key moves the cursor to the first intentional character in a lower segment.

- **UP ARROW** moves the cursor from the top-most segment to the first character of the preceding field. From a lower segment, this key moves the cursor to the character in the same column of the next higher segment, jogging left, if required, to avoid editor blanks, or if it encounters a tab.

- **DOWN ARROW** moves the cursor from the lowest segment to the first character of the next field. From a higher segment, this key moves the cursor to the character in the same column in the next lower segment, jogging left if required to avoid editor blanks, or if it encounters a tab.

- **TAB** enters a **TAB** character in insert mode and moves the cursor to the next tab stop. In typeover mode, this key moves the cursor to the next tab stop that falls on an intentional character, going to the next field segment if required.

The character keys enter data. Any following data shifts right, and words can move down to subsequent segments. This action can result in characters being discarded from the final field segment. These keystrokes can also alter data:

- **CONTROL-A** switches between typeover and insert mode.

- **CONTROL-X** deletes the character under the cursor, possibly causing words to be pulled up from subsequent segments.

- **CONTROL-D** deletes all text from the cursor to the end of the multiple line field (not merely to the end of the current field segment).

- **CONTROL-N** inserts a newline character, causing subsequent text to align at the first column of the next segment of the field and possibly moving words down to subsequent segments. This action can result in characters being discarded from the final segment of the field.

Note: If any of these keys are assigned to the accept key then the accept key behaviour overrides the behaviour mentioned above.

Non-WORDWRAP Displays

The appearance of a character value on the screen can vary, depending on whether or not it is displayed in a multiple-segment WORDWRAP field. For instance, if a value that was entered by using WORDWRAP is displayed without this attribute, words will generally be broken, not wrapped, and TAB and NEWLINE characters will be displayed as question marks. These differences do not represent any loss of data but only a different mode of display.

Note: You can view this effect, for example, if you also have INFORMIX-SQL installed on your system, and you use the **Query Language** menu to display character data values that were entered using WORDWRAP.

If a value prepared under the multiple-line editor is again edited without WORDWRAP, however, some formatting might be lost. For example, a user might type over a TAB or NEWLINE character, not realizing what it was. Similarly, a user might remove a blank from the first column of a line and thus join a word to the last word on the previous line. These mistakes will be visible when the value is next displayed in a WORDWRAP field or in a 4GL report that uses the WORDWRAP operator.

Example Program

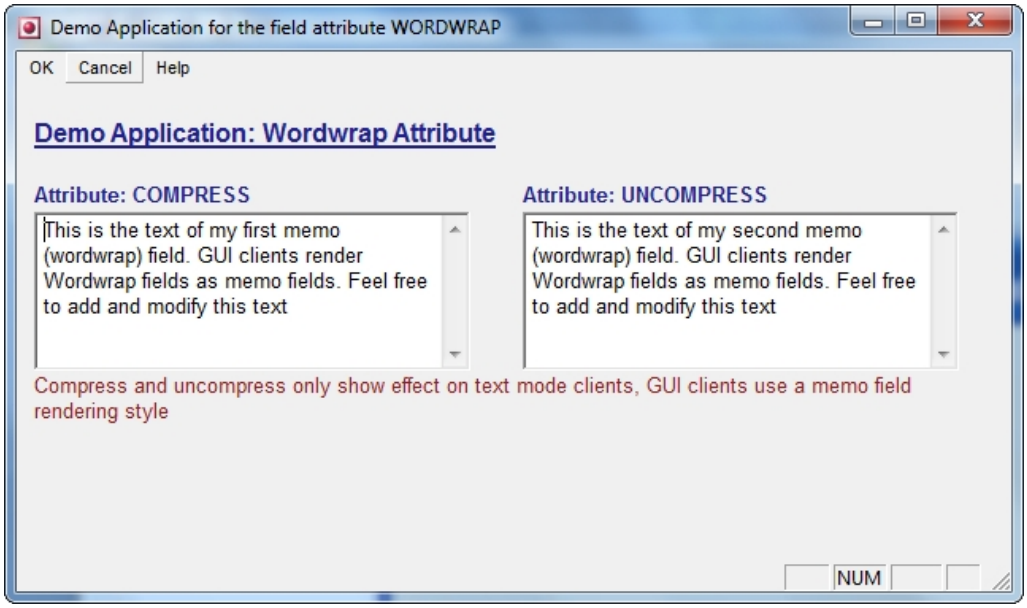
Location:

Project: forms
Program: fm_attribute_wordwrap

Form .per Source code:

```
DATABASE formonly

SCREEN
{
 [dl_header       ]

 [dl_subheader_1    ]        [dl_subheader2       ]
 [f_memo_compress         ] [f_memo_uncompress       ]
 [f_memo_compress         ] [f_memo_uncompress       ]
 [f_memo_compress         ] [f_memo_uncompress       ]
 [f_memo_compress         ] [f_memo_uncompress       ]
 [f_memo_compress         ] [f_memo_uncompress       ]
}


ATTRIBUTES
f_memo_compress=formonly.f_memo_compress,
             wordwrap Compress;
f_memo_uncompress=formonly.f_memo_uncompress,
             wordwrap Noncompress;
dl_header=formonly.dl_header,
             config="{Example: Wordwrap}",
             widget="label";
dl_subheader_1=formonly.dl_subheader_1,
             config="{Attribute: COMPRESS}",
             widget="label",
             color=bold Blue;
dl_subheader2=formonly.dl_subheader2,
             config="{Attribute: UNCOMPRESS}",
             widget="label",
             color=bold Blue;

INSTRUCTIONS

DELIMITERS "[]"
```

Screen:


Demo Application for the field attribute WORDWRAP
OK
Cancel
Help
Demo Application: Wordwrap Attribute
Attribute: COMPRESS
This is the text of my first memo (wordwrap) field. GUI clients render Wordwrap fields as memo fields. Feel free to add and modify this text
Attribute: UNCOMPRESS
This is the text of my second memo (wordwrap) field. GUI clients render Wordwrap fields as memo fields. Feel free to add and modify this text
Compress and uncompress only show effect on text mode clients, GUI clients use a memo field rendering style
NUM

CHAPTER 5

Creating and Compiling a Form

For your Querix4GL program to work with a screen form, you must create a form specification file that conforms to the appropriate syntax, and then compile the form. You can compile the form either in LyciaStudio, or at the command line.

For either method of compiling the database, any tables referenced in the form must already exist, and the database server be running so that the program can access the database.

Also, a section on using default forms is included.

Compiling a Form in LyciaStudio

LyciaStudio is a fully integrated graphical design environment for 4GL Forms. Compiling a form can be done simply by using the menu commands. With a form file (.per) open within LyciaStudio, right-click on its name and choose the Compile menu option.

For full details of compiling forms in LyciaStudio, see the 'Lycia Developers Guide'.

Compiling a Form at the Command Line

The QFORM command line has the following syntax:

```
qform [-V] [-db database] [-xmlin] [-xmlout]
```

Element	Description
Database	This is the SQL identifier of the database to be used.
Filename	This is the name of the Form File, without its .per extension

The **-v** option instructs the form compiler to verify that all fields are as wide as any corresponding character fields that the ATTRIBUTES section specifies.

The **-V** option instructs the form compiler to display the version number and then exit, without compiling anything.

Use the **-db** option to compile the form file against a database which differs from the database specified in the DETABASE section.

The **-xmlin** option converts a graphical form file (.4fm) to text format (.per).

The **-xmlout** option converts a text form file (.per) to graphical format (.4fm).

Using PERFORM Forms in Lycia

The syntax of QFORM forms is different in several significant ways from the syntax of PERFORM, the screen form generation utility of INFORMIX-SQL. You can use PERFORM forms with Lycia, but you must first recompile them using QFORM. In addition, not all PERFORM features are operative. You must also use Querix4GL user-interaction statements like OPEN FORM, OPEN WINDOW, INPUT, DISPLAY FORM, CLEAR FORM, and CONSTRUCT to write a *form driver* to support data entry and data display through the Lycia form.

If you have designed forms for the PERFORM screen transaction program of INFORMIX-SQL, you need to know how those forms behave when used with Lycia. The following features differ from PERFORM to Lycia:

Only the DELIMITERS keyword in the INSTRUCTIONS section of a PERFORM form is supported by Lycia. Other keywords in that section are ignored. To support other INSTRUCTIONS features of PERFORM requires coding in your 4GL program. (See the BEFORE and AFTER clauses of the INPUT statement.)

There is no concept of *current table* in Lycia. An INPUT or INPUTARRAY statement allows the user to enter data into fields that correspond to columns in different tables or even in different databases.

Joins defined in the PERFORM form are ignored in Lycia. You can associate two field names with the same field tag by using the same notation as in a PERFORM join, but no join is effected. However, you can create more complex joins and lookups in Lycia with the full power of SQL.

The PERFORM attributes LOOKUP, NOUPDATE, QUERYCLEAR, and ZEROFILL are inoperative in Lycia. The DISPLAY, DISPLAY ARRAY, DISPLAYFORM, MESSAGE, and OPENWINDOW statements of Querix4GL all ignore the INVISIBLE attribute.

The *conditions* of a COLOR attribute cannot reference other field tags or aggregate functions.