

XBasic

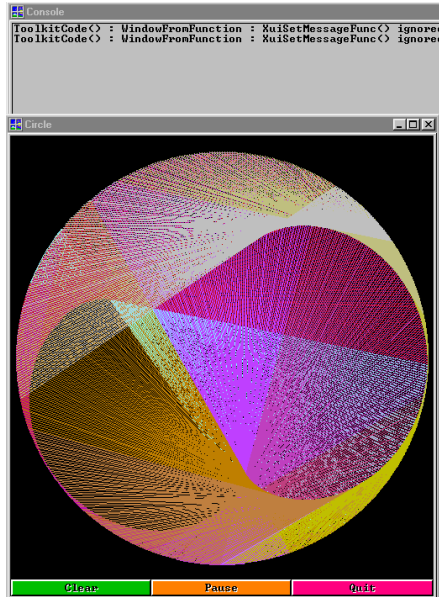
Program Development Environment
(PDE)

GuiDesigner

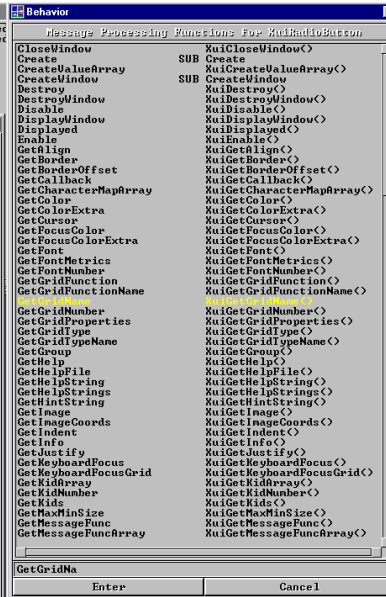
Programmer Guide

*Revision 0.0016
November 1, 1995
Copyright 1990-2000*

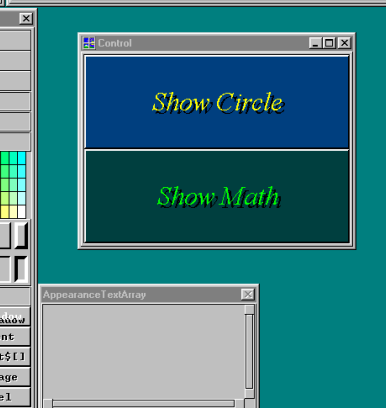
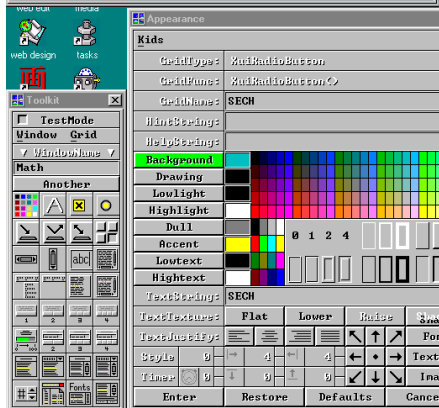
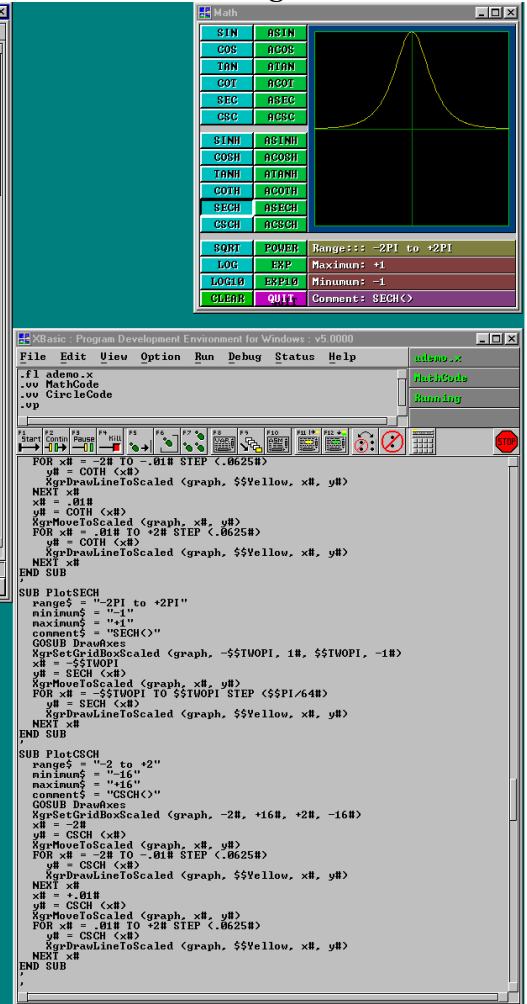
Console Window



Behavior Window



Design Window



Toolkit Appearance Window

Main Window

More windows than normally displayed at once.

- **Main Window** - enter, edit, load, save, run, debug programs - and set options, get help, do most development.
- **Console Window** - conventional BASIC console I/O.
- **Toolkit Window** - create, destroy, load, save design windows - and select controls into visible design window.
- **Design Window** - interactive graphical GUI design layout.
- **Appearance Window** - set appearance properties of the selected grid.
- **Behavior Window** - display the actions of the selected grid for each message.

Table of Contents

Introduction.....	1
What's a GUI ?.....	1
GUI Approach.....	1
Easy To Learn.....	1
GuiDesigner.....	1
Instant Help.....	2
The Ideal Lifetime Program Development Environment.....	2
A Quick Tour of GuiDesigner.....	3
Quick Tour.....	3
GuiDesigner Toolkit.....	3
Window Menu.....	4
Grid Menu.....	5
Appearance Window.....	6
Your Own GUI Programs.....	7
Modify Your GUI Windows.....	8
Fundamental Concepts.....	9
Window.....	9
Window Type.....	9
Window Function.....	9
Grid.....	9
Kid.....	9
Grid Type.....	10
GridFunction.....	10
Callback Function aka Code Function.....	11
GraphicsDesigner Messages.....	11
GuiDesigner Messages.....	11
Send Message.....	12
GraphicsDesigner vs GuiDesigner.....	12
GuiDesigner Programs.....	13
GuiDesigner Convenience Function Programs.....	13
GuiDesigner Programs.....	13
Basic Steps.....	13
Core GuiDesigner Program.....	13
PROLOG.....	14
Entry().....	14
Entry() is your GUI program.....	14
The Nature of GUI Programs.....	14
Entry() - Initialization.....	15
Entry() - Create Windows.....	15
Entry() - Message Loop.....	15
InitGui().....	16
InitProgram() and InitWindows().....	16
CreateWindows().....	16
Design().....	16
DesignCode().....	16

Interactive Window Design.....	17
Design Window.....	17
GridFunction.....	17
Layout Grids.....	18
Move and Resize Grids.....	18
No Overlap.....	18
Nesting Grids.....	18
Grid Appearance.....	18
Grid Behavior.....	18
Design Mode vs Test Mode.....	19
WindowToFunction.....	19
Quick Start.....	19
Callback Arguments.....	19
WindowFromFunction.....	20
Operating Grid Functions.....	21
To Code or Not To Code.....	22
Learning Curve.....	22
Instant Help.....	23
InstantHelp.....	23
Help On Everything.....	23
Posting InstantHelp.....	23
Browsing Programs.....	23
Solve the "Great Icon Problem".....	23
Copy from the InstantHelp Window.....	23
Update Instant Help.....	23
Help Files.....	24
HelpFile Format.....	24
HelpString.....	24
:entryname.....	25
Default :entryname.....	25
Multiple HelpFiles.....	25
Set HelpFile.....	25

Messages.....	27
Messages.....	27
GraphicsDesigner Messages.....	27
Window Messages and Grid Messages.....	27
Window Messages.....	27
Grid Messages.....	27
Message Anatomy.....	28
window, grid, wingrid.....	28
message.....	28
v0,v1,v2,v3,r0,r1.....	28
kid.....	28
GraphicsDesigner Messages.....	29
Keyboard Messages.....	30
Keyboard Focus.....	30
xWin, yWin.....	30
state.....	30
time.....	30
Examples.....	30
WindowKeyUp vs WindowKeyDown.....	30
Virtual Key Codes.....	31
Mouse Messages.....	32
x,y.....	32
state.....	32
time.....	32
Mouse Message Algorithm.....	32
Message Queue.....	33
Process Message.....	33
XgrProcessMessages().....	33
Message Loop.....	34
Process a Message.....	35
Window Function.....	35
Window Functions Process Window Messages.....	36
Window Functions Process Grid Messages.....	36
Grid Function.....	37
Send Message.....	38
XuiSendMessage().....	38
Runtime Messages.....	39
Example.....	39
Callback Messages.....	40
Callback Functions.....	40
Monitor Messages.....	41
CEO Function.....	41
Slow Pokes.....	42
Advanced Message Processing.....	42

Anatomy of Grid Functions.....	43
Overview.....	43
Grid Functions and Callback Functions.....	43
Merged Grid Function.....	43
Grid Function Example.....	44
Function Declaration.....	47
Function Definition.....	47
Variable Declarations.....	48
Kid Constant Definitions.....	48
Initialize - Process Message - RETURN.....	49
Initialize.....	49
Process Message with Message Processing Function.....	49
Process Message with Message Processing Subroutine.....	49
Done.....	49
Callback Subroutine.....	50
Create Subroutine.....	51
CreateWindow Subroutine.....	51
GetSmallestSize Subroutine.....	52
Resize Subroutine.....	53
Selection Subroutine.....	54
Initialize Subroutine.....	55
Get Default Message Functions.....	56
Establish Message Functions.....	56
Establish Message Subroutines and Register Grid Type.....	57
Establish Grid Type Properties.....	58

Introduction

What's a GUI ?

GUI is an acronym for *Graphical User Interface*.

Most popular computer programs interact with people through a GUI. Users prefer GUI programs because they are easier to learn. GUI programs ...

- Display text and graphics in windows.
- Accept input from a keyboard and mouse.

GUI Approach

GUI programs repeat three basic steps indefinitely ...

1. The program displays its capabilities on labels, buttons, lists, menus, etc# .
2. The user presses a button, selects a menu or list entry, to tell the program what to do.
3. The program executes the user command, then returns to step 1.

This process repeats until the user tells the program to quit.

Easy To Learn

Well designed GUI programs are easy to learn because they anticipate and accommodate user wishes. Conventional programs often make users conform to the program.

GuiDesigner

GuiDesigner and the program development environment help you develop attractive, efficient programs that are easy to learn, a delight to operate, and portable between diverse computer systems.

You can create sophisticated custom GUIs quickly, interactively, and graphically with *GuiDesigner* design windows. *GuiDesigner* creates the basic core of your program, complete with functions that create and operate the GUI you designed graphically. You can focus on the goals of your program. *You're the ProgramDesigner.*

Your programs can be as large and sophisticated as you like, because every program is fully 32 or 64-bit, top to bottom, no holds barred. And your programs run *fast* because the compiler translates them into binary machine code - *the fastest kind there is!*

GUI components like labels, buttons, lists and menus are commonly called *grids*, *controls*, *widgets*, and probably other names too, depending on the source. This documentation calls GUI components *grids*, a name derived from "*graphics identity*" by *GraphicsDesigner*.

Instant Help

GuiDesigner programs are easier to learn than other GUI programs. That's because *GuiDesigner* imbeds automatic, user-extensible documentation into every program you develop. Users can click the right mouse button on *any* GUI component to learn its purpose and operation. *GuiDesigner* displays an *InstantHelp* window and fills it with whatever informative text you provided for the occasion.

But there's more. *Users can edit the InstantHelp window!*

Users can copy text from the *InstantHelp* window and insert it into any text component in their GUI. Which means they can extract samples from your help text and try 'em out. But that's not all.

Users can change the help text. They can copy, delete, insert, and type in the *InstantHelp* window just like any other text area. So they can expand, contract, reword, and update help text in whatever way best suits their style, experience, and preferences. When they press the Update button, *GuiDesigner* updates the help files. Henceforth the *InstantHelp* window will contain the new, improved information!

The Ideal Lifetime Program Development Environment

The program development environment and *GuiDesigner* are powerful, efficient, professional software development tools, appropriate for all kinds of programming projects.

At the same time, the program development environment and *GuiDesigner* are easy and fun to learn, so they're perfect for novice programmers too. That's why they're ideal lifetime programming tools.

A Quick Tour of GuiDesigner


Quick Tour

The best way to learn *GuiDesigner* is to look it over a bit, then create a simple GUI program.

Often it's better to experiment with GUI programs for awhile before reading about them in abstract terms. It's easier to relate to general discussions that refer to things you've already "seen and touched".










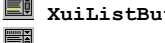






GuiDesigner Toolkit



Click on the  toolkit hot button in the PDE main window to display the toolkit along the lower left side of the display.

TestMode CheckBox : Click to toggle between DesignMode and TestMode.
Toolkit Menu : Work with design windows and grids in the design windows.

Design Window Name : Give a valid function name to every design window.
Another Button : Create another grid of the same type as the last selected grid.

-  **XuiColor** : Color grid (select one of the 125 standard *GraphicsDesigner* colors).
-  **XuiLabel** : Label grid (display single or multi-line message text and/or image).
-  **XuiCheckBox** : CheckBox grid (toggle on/off).
-  **XuiRadioButton** : RadioButton grid (toggle on turns all other RadioButton grids in group off).
-  **XuiPushButton** : PushButton grid (selection on MouseDown - MouseUp).
-  **XuiToggleButton** : ToggleButton grid (toggle on/off on MouseDown).
-  **XuiScrollBarH** : ScrollBarH grid (horizontal motion / position control).
-  **XuiScrollBarV** : ScrollBarV grid (vertical motion / position control).
-  **XuiTextLine** : TextLine grid (one line text input / output).
-  **XuiTextArea** : TextArea grid (multi-line text input / output).
-  **XuiMenu** : Menu grid (create / manage pulldown list for MenuBar entries).
-  **XuiMenuBar** : MenuBar grid (select one of several horizontal entries).
-  **XuiPullDown** : PullDown grid (select one of several vertical entries).
-  **XuiList** : List grid (scrollable list - select one of several vertical entries).
-  **XuiMessage1B/2B/3B/4B** : Message box with 1,2,3,4 buttons.
-  **XuiProgress** : Progress box.
-  **XuiDialog2B/3B/4B** : Dialog box with 2,3,4 buttons.
-  **XuiDropButton** : Button activated PullDown List.
-  **XuiDropBox** : PullDownList box.
-  **XuiListButton** : Button activated scrollable List.
-  **XuiListBox** : Scrollable List box.
-  **XuiRange** : Set value within a Range.
-  **XuiFile** : Select File Dialog List.
-  **XuiFont** : Select Font Dialog List.
-  **XuiListDialog2B** : Select List Entry Dialog with 2 buttons.

Window Menu



Select **Window** in the toolkit menu and the following entries appear:

WindowNew Create and display a new design window. Hide any current one.
WindowHide Hide design window.
WindowLoad Load a grid function from disk and convert into a design window.
WindowSave Convert design window into a grid function and save it on disk.
WindowDelete Delete design window and remove it from display.
WindowToFunction ... Convert design window into grid/callback functions in your program.
WindowFromFunction .. Convert grid function in your program to design window.
WindowCloseToolkit .. Hide toolkit from view - redisplay with toolkit hot button.

WindowNew creates a new design window and displays it in the upper right corner of the screen. Only one design window can be visible at a time, so any displayed design windows are hidden before the new one is created.

WindowHide removes any visible design window from the display.

WindowLoad reads a grid function from disk, converts it into a new design window, and displays it on the screen.

WindowSave converts a design window into a grid function, then saves it to disk. The grid function can be loaded with **WindowLoad**.

WindowDelete destroys the visible design window. All information is lost and is not recoverable, so think carefully before you delete.

WindowToFunction converts the visible design window into:

*A grid function with the same name as the design window.
A callback function called by the grid function when important events occur in the design window.
Three lines of code that create, activate, initialize, and display the design window.*

WindowFromFunction converts the displayed grid function into a design window that you can modify, then convert back to a function.

WindowCloseToolkit hides the toolkit from view. Click on the toolkit hot button to redisplay the toolkit.

The names of all design windows follow **WindowCloseToolkit**. Select any of these design window names to hide the currently visible design window and display the design window you selected.

Grid Menu



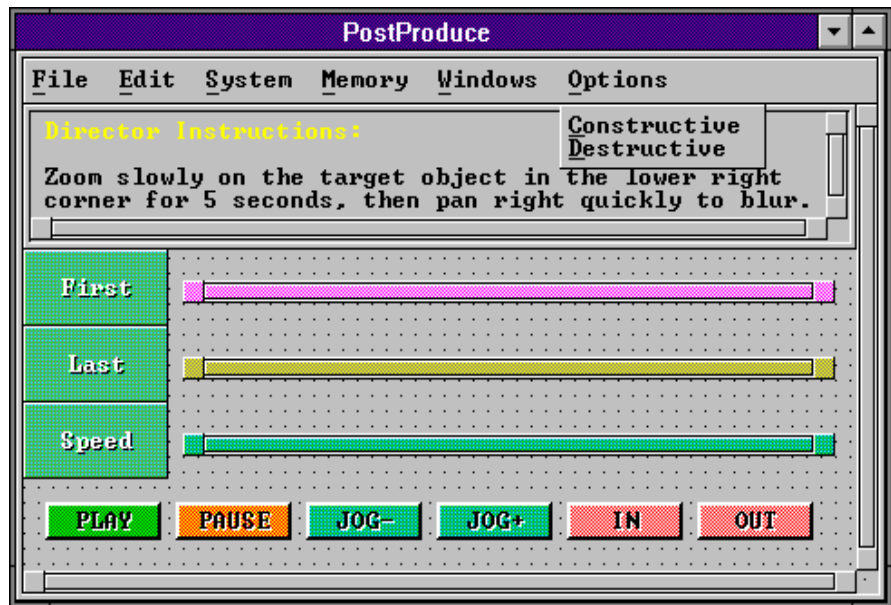
Select **Grid** in the toolkit menu and an **XuiPullDown** with the following entries appears:

GridAppearance displays an AppearanceWindow that shows the appearance properties of the selected grid. The appearance properties can be changed interactively. You can also make the Appearance window appear by double clicking on a selected grid.

GridBehavior displays a Behavior window that lists the function and/or subroutine that processes every message recognized by the selected grid.

GridSortKids reorders the grids in the design window so that grids enclosed by other grids are higher kid numbers, and therefore drawn later. The sort orders from top to bottom, left to right.

GridDelete destroys the selected grid and removes it from the design window. A delete keystroke will also delete a selected grid.



A simple design window

Appearance Window

Double-click a selected grid to display an *Appearance Window* that displays *grid properties*, including its name, help string, text string, border, eight currently selected color properties, and lots more.

You can enter new text values to change `gridName`, `helpString`, `hintString`, and `textString` properties. Click `Text$[]` to display a window that accepts the multiline `textArray` property. Click `Text$[]` again to hide the window and update the grid.

You can change any or all of the color properties by clicking on the color property button you want to change, then colors in the palette. The appearance of the grid changes instantly to reflect your choices.

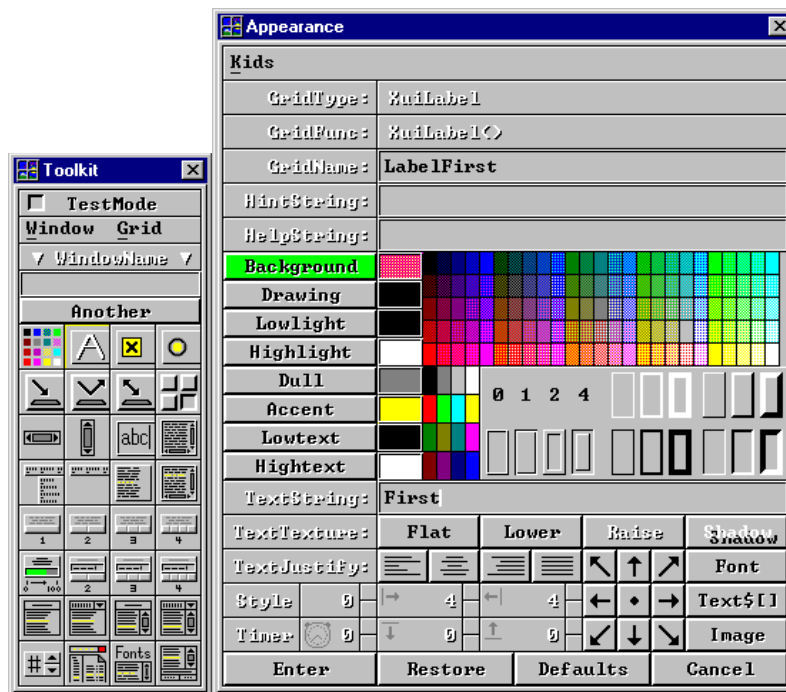
You can change border styles by clicking on border samples. To set the secondary border style for grids that support two styles, hold the `shift` key down when you click on the border sample.

You can align text to one of nine different nominal positions, then fine tune text location them with indent controls. And you can select left, center, right, or both justification for grids that support it.

Click `Restore` to reset the properties of the grid to the values they had when you starting editing it.

Click `Defaults` to make all its properties revert to the nominal values of its grid type.

Click `Cancel` to hide the Appearance window.



Toolkit

Appearance Window

Your Own GUI Programs

To create your first *GuiDesigner* program, perform the following steps:

PS: If you have trouble, compare your program with `afirst.x` in your development directory.

- Select **FileNew GuiProgram** in the main menu bar.
GuiDesigner creates a skeleton GUI program, some of which appears in the lower text area.
- Select the **Toolkit** hot button in the main window.
A toolkit of GUI components (called grids) appears along the lower left side of the display.
- Select **WindowNew** in the toolkit menu bar.
A design window appears in the upper right corner of the display screen.
- Enter "First" into the text line near the top of the toolkit.
This names the design window. The name appears in the design window title bar.
- Click on the **PushButton** button in the toolkit.
A double-box "grip" appears in the design window - the movable, resizable "selected grid".
Only one grid can be selected at a time. Selecting one deselects any other.
Click on background or another grid to deselect the selected grid and make it appear normally.
- Resize and move the grip in the design window (about 5 dots high and 10 dots wide).
Move by dragging center of grip. Resize by dragging sides or corners of grip.
Position it near the top center of the design window.
- Click on the **PushButton** button in the toolkit again to create another push button.
Position it below your first push button.
- Click on your first push button to select it.
The selected grid is always drawn as the resizable grip.
- Double click on the grip.
The Appearance window appears next to the toolkit.
- Enter "First" in the TextString entry in the Appearance window and press <enter>.
"First" appears on the first push button you created.
- Enter "FirstButton" in the GridName entry in the Appearance window and press <enter>.
The button is named "FirstButton", but nothing visible changes.
- Click on medium-light colors until you find one you like.
Try a medium-light green or cyan - top row, 11th or 16th color from left.
- Click on your second push button to select it.
Your second push button is selected and turns into a grip.
- Double click on the grip.
The contents of the Appearance window changes to reflect your second push button.
- Set color of your second push button, set TextString to "Second", set GridName to "SecondButton".
"Second" appears in your second push button.
- Click the **Cancel** button in the Appearance window to make the Appearance window disappear.
- Select **WindowToFunction** in the toolkit menu bar.
GuiDesigner adds `First()` and `FirstCode()` function declarations to your program.
GuiDesigner converts the design window into two functions, `First()` and `FirstCode()`.
`First()` is a "grid function" that contains the code that creates and operates your window.
`FirstCode()` is a "callback function" that responds to important events in your window.
`FirstCode()` is visible in the lower text area of the main window.
When `FirstCode()` gets callbacks, the first executable line reports the message and arguments.
- Select **WindowDelete** in the toolkit menu bar.
Your design window is deleted and disappears.
- Click on the **Start** hot button.
Your program parses, compiles, runs. Your window appears in the upper right portion of the display.
The ReportMessage window appears in the upper left portion of the display (for testing only).
- Look in ReportMessage window as you click on your "First" and "Second" push buttons.
A "Selection" callback message prints in the ReportMessage window each time you click a button.
The `kid` argument should be 1 for your "FirstButton" button and 2 for your "SecondButton" button.
- Click the **Kill** hot button in the main window to terminate your program.
- Complete the following code in the `SELECT CASE` block in `FirstCode()`:

```
CASE $FirstButton : XuiSendMessage (grid, #SetTextString, 0, 0, 0, 0, kid, "Hello")
                  XuiSendMessage (grid, #Redraw, 0, 0, 0, 0, kid, 0)
CASE $SecondButton : XuiSendMessage (grid, #SetTextString, 0, 0, 0, 0, kid, "World")
                  XuiSendMessage (grid, #Redraw, 0, 0, 0, 0, kid, 0)
```
- Select the **Start** hot button again and click the buttons again.
The labels on your buttons will change the first time they're clicked.
Hey, it works !!!



Toolkit hot button



PushButton button



Start hot button



Kill hot button

Congratulations. You've created a simple but complete *GuiDesigner* program that presents a GUI window and responds to user events!

What if you want to modify your window design? The next page tells how.

Modify Your GUI Windows

- Click the **Kill** button in the main window to stop your program.
Your window disappears.
- Select **ViewFunction** in the main menu bar.
A list of the functions in your program appears.
- Select function **First()** or **FirstCode()** from the function list.
First() or **FirstCode()** is displayed in the lower text area of the main window.
- Click the *GuiDesigner* toolkit hot button near the right end of the horizontal row of hot buttons in the main window.
The toolkit window reappears.
- Select **WindowFromFunction** in the toolkit menu bar.
GuiDesigner converts the *code* in the displayed function **First()** into a new design window and displays it.
- Click on the **PushButton** button in the toolkit.
A grip appears in the recreated design window.
- Position and resize the grip below your "Second" push button.
- Double click the grip to redisplay the Appearance window.
- Set the color of your third button, set GridName to "ThirdButton" set TextString to "Third".
- Click the **Cancel** button in the Appearance window.
The Appearance window disappears.
- Select **WindowToFunction** in the toolkit menu.
GuiDesigner converts the modified design window into a modified **First()** function.
GuiDesigner asks whether you want to **Update** or **Replace** the existing **First()** and **FirstCode()**.
- Click the **Update** buttons so the code you added to **FirstCode()** isn't changed.
GuiDesigner updates your existing **First()** and **FirstCode()** functions to reflect your changes.
- Select **WindowDelete** in the toolkit menu.
The modified design window is deleted and disappears.
- Add the following code to the **SELECT CASE** block at the bottom of **FirstCode()**.

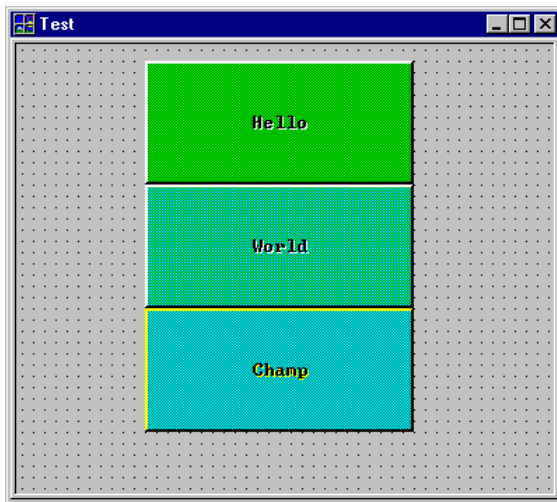
```

CASE $ThirdButton : XuiSendMessage (grid, #SetTextString, 0, 0, 0, 0, kid, "Champ")
                   XuiSendMessage (grid, #Redraw, 0, 0, 0, 0, kid, 0)

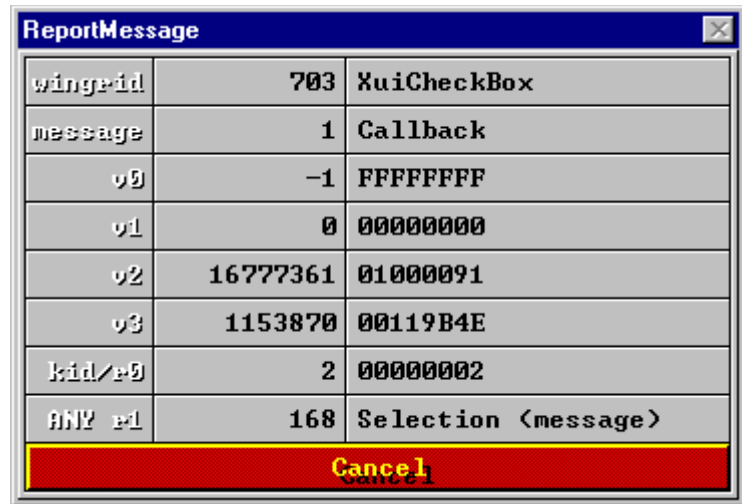
```
- Click the **Start** hot button in the main window.
Your modified design window appears.
- Click your buttons and watch your program work.

Congratulations. You've now modified your first *GuiDesigner* program and made it work again.

Better yet, you're a *GuiDesigner* pro! That's because developing all GUI programs is essentially the same. And you've already been there.



afirst.x - Your First GUI Design



ReportMessage Window

Fundamental Concepts

Window

Your programs can create and operate any number of graphics *windows*, each of which is a rectangular area on the *display* screen.

Each window can contain any number of GUI components: buttons, labels, text-areas, dialog-boxes, etc.

Window Type

Windows have several optional features, including minimize and maximize buttons, system menu button, title-bar, and resize frame. Any combination of these defines the *window type* of a window.

Window Function

The function responsible for operating a particular window is called its *window function*. In most programs, every window is controlled by `XuiWindow()`, the *GuiDesigner* standard window function, so you may never think about window functions again.

Grid

A GUI components is called a *grid*, a term from *GraphicsDesigner*.

To *GraphicsDesigner* programs, a grid is a rectangular area in a window with its own coordinate system and graphics properties like `backgroundColor`, `drawingColor`, etc.

To *GuiDesigner* programs, a grid is a familiar GUI component like a label, pushbutton, text-line, dialog-box, etc. At heart, all grids are graphics grids. But you'll probably never think of them that way because *GuiDesigner* adds properties and a "grid function" to make them look and act like the GUI components in your GUI programs.

Each *GuiDesigner* grid is:

- A basic graphics grid from *GraphicsDesigner*, plus
- Properties that make it look and behave like a GUI component, plus
- A "grid function" that creates, operates, and responds to events in the grid

Kid

Some kinds of grids contain other grids. Dialog boxes, for example, contain a Label grid, a TextLine grid, and one or more PushButton grids. The grids inside other grids are called *kid grids*, or just *kids*, and are identified by kid numbers. The Dialog box is their *parent*.

Kid numbers begin with 0 for the grid itself, followed by 1,2,3... for its kids, numbered from left to right from top to bottom.

Grid Type

Many kinds of grids are supplied with *GuiDesigner*. Each kind is called a *grid type* and has a name like:

Grid Type	Grid Function	- Description
XuiArea	XuiArea()	- mouse/keyboard even forwarding
XuiCheckBox	XuiCheckBox()	- check / uncheck an item
XuiDialog2B	XuiDialog2B()	- Label + TextLine + 2 buttons
XuiDialog3B	XuiDialog3B()	- Label + TextLine + 3 buttons
XuiFile	XuiFile()	- select drive/directory/file
XuiLabel	XuiLabel()	- message Label
XuiList	XuiList()	- scrollable List
XuiListBox	XuiListBox()	- pulldown ListBox
XuiMessage2B	XuiMessage2B()	- message Label + 2 PushButtons
XuiMessage3B	XuiMessage3B()	- message Label + 3 PushButtons
XuiMessage4B	XuiMessage4B()	- message Label + 4 PushButtons
XuiPullDown	XuiPullDown()	- fixed length list (no scroll)
XuiPushButton	XuiPushButton()	- selected by down/up click
XuiRadioButton	XuiRadioButton()	- only one in group selected
XuiScrollBarH	XuiScrollBarH()	- horizontal scroll bar
XuiScrollBarV	XuiScrollBarV()	- vertical scroll bar
XuiTextArea	XuiTextArea()	- multiple line editable text
XuiTextLine	XuiTextLine()	- single line editable text
XuiToggleButton	XuiToggleButton()	- toggles ON / OFF

Grid type names always contain a prefix that identifies the origin of the grid, hence grid type names like `XuiLabel`, `XuiPushButton`, etc. `Xui` identifies standard grid types. Grids from other vendors carry their own pre-arranged identifying prefix.

GridFunction

The function that creates and operates a particular grid is called its *grid function*.

- *Every grid type has its own grid function.*
- *Each grid function creates and operates every grid of its grid type.*
- *The the name of every grid function is the same as its grid type.*

Grid functions give each grid type its individual characteristics. `XuiLabel()` and `XuiPushButton()` make `XuiLabel` and `XuiPushButton` grids look and act different.

When `WindowToFunction` converts design windows to functions, it's converting them to grid functions.

You don't need to modify grid functions. They know how to create grids and perform all routine chores that make the grids functional.

To perform their function, however, most grid functions have to report certain events or conditions to your program. For example, when a user clicks on a `PushButton`, your program needs to know. Otherwise nothing would happen.

In other words, grid functions create and operate the GUI, and report important events. How your program responds to these reports determines the nature of your program.

Callback Function aka Code Function

WindowToFunction converts any design window into a grid function and a *callback function*.

When important events occur in a grid, its grid function calls the callback function and passes it message arguments that describe the situation. You can put code in callback functions to respond to any combination of events reported by the grid function, from none to all.

What constitutes an important event depends on the grid type. **MouseDown** + **MouseUp** generates a callback message in an **XuiPushButton** grid, but not in an **XuiTextLine**, while an Enter keystroke generates a callback in both grid types.

GraphicsDesigner Messages

When a basic event takes place in a graphics window, *GraphicsDesigner* creates a *message* to describe the event. Whenever the state of keyboard, mouse, or window changes, *GraphicsDesigner* creates a message. Every message contains 8 arguments:

- A window or grid number to identify the window or grid the message refers to.*
- A message number to identify the nature of the event that occurred.*
- 6 general purpose arguments whose meanings depend on the message number.*

GraphicsDesigner messages include **WindowKeyDown**, **WindowResized**, **MouseDown**, **TimeOut**.

GuiDesigner Messages

Processing each *GraphicsDesigner* message typically causes other activity, involving additional *GuiDesigner* messages. *GuiDesigner* automatically maintains the GUI, tells your program about the event, and processes messages your program sends to update the GUI.

Each step in this process is a function in *GuiDesigner* or your program processing a message sent to it by another function.

GuiDesigner recognizes over 100 messages like **Create**, **Destroy**, **HideWindow**, **Redraw**, **Resize**, **SetColor**, **SetTextString**.

Messages are a fundamental part of *GuiDesigner* and GUI programs. Messages are pre-defined for every standard aspect of GUIs, and standard message processing functions are provided with *GuiDesigner* to process these messages appropriately.

- Routine messages are processed automatically by grid functions.
- Your programs *send messages* to grids to control the GUI.
- Grid functions send *callback messages* to your program when important events occur.

Send Message

To *send a message* means to call a function and pass it a standard 8 argument message.

For example, to send a message to a window or grid, call:

```
XuiSendMessage ( window, message, v0, v1, v2, v3, r0, r1 )  
XuiSendMessage ( grid, message, v0, v1, v2, v3, kid, r1 )
```

window or **grid** contains a window number or grid number, **message** contains the message number of the desired operation to perform. The meaning of the other arguments depends on **message**. For messages sent to grids, the **kid** argument contains the kid number within **grid** to send the message to, or 0 for **grid** itself.

XuiSendMessage() looks up the function responsible for operating **window** or **grid**, then calls it, passing it all 8 message arguments.

GraphicsDesigner vs GuiDesigner

GraphicsDesigner and *GuiDesigner* have many similar functions.

You can write pure graphics programs (without GUI) by calling only *GraphicsDesigner* functions (prefix **xgr**).

But GUI programs *must* call *GuiDesigner* functions whenever equivalent functions exist in both function libraries, or *GuiDesigner* will malfunction. Always check *GuiDesigner* first.

To draw graphics, however, your program calls **xgr** drawing functions in the *GraphicsDesigner* library. *GuiDesigner grids are GraphicsDesigner grids*. So your program passes *GuiDesigner* grid numbers to *GraphicsDesigner* functions to draw in them.

GuiDesigner Programs

GuiDesigner Convenience Function Programs

You can develop programs with simple GUIs without designing your own windows. Your programs call *convenience functions* that create and operate windows containing just about any kind of grid. *GuiDesigner* convenience functions let you avoid messages, window functions and grid functions, because all are hidden from your programs by the convenience functions.

With convenience functions you can write programs that create and operate simple GUIs with nothing more than simple function calls. The *GuiDesigner* convenience functions, *GuiDesigner* convenience function programs, and the standard *GuiDesigner* toolkit grids are described in a separate document.

GuiDesigner Programs

Though you can develop reasonably capable programs with convenience functions, they tap only part of the power and flexibility *GuiDesigner* has to offer. Eventually you'll want to design your own windows and/or grids, which are always part of standard *GuiDesigner* programs.

GuiDesigner programs that control GUI windows of your own design are both easy and fun to create. You can create a basic *GuiDesigner* program, complete with any number of custom GUI windows that you design interactively and graphically - all without writing single line of code.

Basic Steps

To create a complete, functional, standard *GuiDesigner* program:

- *You tell GuiDesigner to create a complete core GUI program.*
- *You interactively design a main window, or choose an existing one.*
- *You write code to respond to user requests when they operate the GUI.*

Core GuiDesigner Program

All *GuiDesigner* programs are created equal. They don't stay that way, but they do start out that way. When you select **FileNew GuiProgram** from the main menu, *GuiDesigner* creates a GUI program:

- **PROLOG** - *Declarations of composite types, functions, shared constants.*
- **Entry()** - *Calls initialization functions and enters main message loop.*
- **InitGui()** - *Initializes message numbers, and basic GUI variables.*
- **InitProgram()** - *Initializes your program.*
- **CreateWindows()** - *Creates, initializes, and displays the GUI windows you designed.*
- **InitWindows()** - *Initializes windows created by your program.*

GuiDesigner creates this core *GuiDesigner* program for you, though it doesn't put anything in **InitProgram()** or **InitWindows()** because it has no way to know what your program needs to do. After all, *you're the ProgramDesigner*.

PROLOG

The core `PROLOG` contains:

- `IMPORT "xgr"` - Declares *GraphicsDesigner* types, functions, constants.
- `IMPORT "xui"` - Declares *GuiDesigner* types, functions, constants.
- Declarations for functions *GuiDesigner* automatically creates for you.

Entry ()

`Entry ()` contains three sections:

1. Initialize *GuiDesigner*, *GuiDesigner* variables, and your program.

```
Xui ()           ' initialize GuiDesigner
InitGui ()       ' initialize messages, etc.
InitProgram ()  ' initialize your program
```
2. Create and display the main window and any others you design.

```
CreateWindows () ' create your program's windows
InitWindows ()   ' initialize windows if needed
```
3. Process messages in the message loop.

```
DO           ' the message loop ...
  XgrProcessMessages ( 1 ) ' processes one message ...
LOOP UNTIL terminateProgram ' and repeat until program terminates
```

`Entry ()` is your *GUI program*

No matter how large, sophisticated, and elaborate your GUI program is, `Entry ()` is your *program*. Everything else is support for `Entry ()`. That may seem far fetched, but consider:

`Entry ()` initializes everything, displays your main window, then enters the message loop where it waits for a message. From then on, the message loop is the "base of operations" for your program. The message loop calls other functions to process user-generated messages as they occur, but those functions always return to the message loop when they're done. Your program spends its life in the message loop, except for occasional excursions to process a message.

The Nature of GUI Programs

That's how GUI programs work. Most of the time they sleep in the message loop at the bottom of the entry function, waiting for activity. When the user presses a keyboard key or mouse button or otherwise generates a message, the message loop wakes up and calls the function that knows how to process the message. After the function responds to the user action, it returns to the message loop where the program again falls asleep until the user generates another message.

1. Sleep in the message loop until a message arrives. When it does...
2. Call the function that knows how to process the message.
3. Go to step 1.

The basic structure of *GuiDesigner* programs is amazingly simple. What's more amazing is that this basic structure is flexible enough for any kind of program.

Hey !!! GUI programs don't have to be difficult after all !!!

Entry () - Initialization

Xui () initializes *GuiDesigner*.

InitGui () initializes message variables in your program.

InitProgram () initializes your program.

If your program calls library functions, their initialization functions should be called after **Xui ()**.

Entry () - Create Windows

CreateWindows () is a function that calls the grid functions that create the windows in your program. Each time you design a new window for your program, *GuiDesigner* adds code to **CreateWindows ()** to create, activate, and display the window.

The code typically looks like the following:

```
Design          (@Design, #Create, x, y, width, height, 0, &XuiWindow())
XuiSendMessage ( Design, #SetCallback, Design, &DesignCode(), -1, -1, -1, 0)
XuiSendMessage ( Design, #DisplayWindow, 0, 0, 0, 0, 0, 0)
```

This creates a window, sets its callback function, and displays it. Comment out the **DisplayWindow** line for any window you don't want to appear when your program starts up.

These three lines set the following important example:

- Your programs must call grid functions directly to create grids and windows.
- Your programs can send all other messages with **XuiSendMessage ()**.

Specifically, your programs *must* call grid functions directly with **Create** and **CreateWindow** messages. For all other messages your programs can call **XuiSendMessage ()** and let it look up and call the function associated with the grid argument.

Entry () - Message Loop

The message loop is the base of operation for your program.

XgrProcessMessages (1) goes to sleep until a message becomes available. This keeps your program from wasting computer time, and lets other programs run when they can.

When the user operates the keyboard or mouse, *GraphicsDesigner* adds a message to the queue and wakes up **XgrProcessMessages ()**.

XgrProcessMessages (1) calls the window function associated with the message. The window function processes the message and returns, or sends the message to a grid function, then returns.

Until processing a message makes **terminateProgram** non-zero, the message loop repeats indefinitely.

InitGui ()

`InitGui ()` initializes message number variables for every message *GraphicsDesigner* and *GuiDesigner* recognize. If you create new messages of your own, add them to `InitProgram ()`. Don't forget to make your new message variables shared, as in `#MyMessage`.

InitProgram () and InitWindows ()

`InitProgram ()` and `InitWindows ()` are empty functions waiting for code your program needs to initialize itself and / or its windows. If your program doesn't need initialization, leave `InitProgram ()` and / or `InitWindows ()` empty, but don't delete them.

CreateWindows ()

`CreateWindows ()` is the function that creates and displays the windows for your program with the following lines:

```
Design          (@Design, #CreateWindow, x, y, width, height, 0, "")
XuiSendMessage ( Design, #SetCallback, Design, &DesignCode(), -1, -1, 0, 0)
XuiSendMessage ( Design, #DisplayWindow, 0, 0, 0, 0, 0, 0)
```

As you design more windows for your program, *GuiDesigner* adds similar sets of lines. This creates every window in your program, even if most won't be displayed until later. Comment out the `DisplayWindow` line for windows that shouldn't appear at startup.

Design ()

When you're ready, *GuiDesigner* will convert a design window into:

- Code in `CreateWindows ()` to create, initialize, and display the window.
- A grid function that knows how to create and operate the window.
- A callback function to handle callback messages from the grid function.

The name of the grid function is the name you gave to the design window. The callback function is the same with `Code` appended. For example:

```
Design          - Name you gave to your design window.
Design          - Name of the grid type created from your design window.
Design ()       - Name of the grid function that creates and operates the design window.
DesignCode ()  - Name of the callback function that handles callbacks from the grid function.
```

DesignCode ()

`DesignCode ()` is the callback function or "code function" that callback messages are sent to when important events occur in your window. This connection was established when `CreateWindows ()` sent a `SetCallback` message to the `Design` grid.

`DesignCode ()` is the function in your program that responds to events in the `Design` window. When the user clicks on a `PushButton` or selects an item from a `Menu` or `List`, `DesignCode ()` hears about it, because `Design ()` sends it a callback message.

The code in `DesignCode ()` examines the callback message and arguments to determine the nature of the callback message, and takes appropriate action. Though *GuiDesigner* creates callback functions complete with code to receive callback messages, you have to add the code that responds to user actions.

Interactive Window Design

Design Window

To design custom grids or windows, you select grids from the *GuiDesigner toolkit* and lay them out in *design windows*. To reduce screen clutter and make it easier for you to focus on your work, only one design window is active at a time - all other design windows are hidden from view.

You can display, hide, delete, and switch between design windows whenever you want. Be sure to enter a descriptive name into the toolkit TextLine for each window when you create it.

GridFunction

On request, *GuiDesigner* will convert the active design window into:

- Three lines of code to create, initialize, and display the window.
- A *grid function* containing the code to create and operate the grid.
- A *callback function* to receive callback messages from it.

If your program contains an existing version of the grid and/or callback functions, *GuiDesigner* lets you keep the function unchanged, keep the function but update the kid constants and **Create** subroutine, or replace it with the new one - in which case any code you added to the existing function is lost.

If **CreateWindows ()** contains code to create, initialize, and display a previous version of the window, that code is replaced with equivalent code for the new design window.

Whenever your program calls the grid function with a **CreateWindow** message, a window of the specified size and position is created, containing a grid of the type you designed.

Your program can send messages like **SetBorder**, **SetColor**, **SetTextString** to the grid and its kids to configure and control them. When your program is finished with the window, it can send a **Destroy** message to the grid to destroy the window and grid.

At any time during development you can convert grid functions back into design windows and modify them interactively and graphically, the way you designed them. Or you can edit the grid functions *GuiDesigner* wrote for you just like any other function.

The grid functions *GuiDesigner* writes for you are so modular you don't even have to look at them to use them, though you're free to inspect and modify them when you want.

Layout Grids

Click on a grid type button in the toolkit to create a grid of that type. Whether you choose a simple Label or a complex special purpose grid, *GuiDesigner* draws the selected grid in the design window as a double rectangle with resize grips along the sides and corners.

To select a grid, click on it. To deselect it, click on the background or another grid. Except when they're selected, grids look the same in design windows as finished programs. So you can fine tune your design window layout before you write code to interact with them.

Move and Resize Grids

You can move and resize selected grids anywhere in the design window. To move the selected grid, place the mouse cursor over its central area, press the left button, and drag the mouse. To resize the selected grid, place the mouse cursor over one of the side or corner grips, press the left button, and drag the mouse.

No Overlap

Don't let grids overlap each other. Before you continue working, place the grid properly. You can resize the design window to accommodate any grid layout. But keep windows reasonably small. They share screen space with other windows, including your own.

Programs that create, move or resize grids at runtime must make sure they don't overlap unintentionally. Nothing in *GuiDesigner* prevents it, but peculiar appearance and behavior will usually result.

Nesting Grids

Though grids should not overlap, they can nest within each other to any depth. This is how composite grids like dialog boxes and file boxes are constructed.

Grid Appearance

To display an *Appearance Window*, double click the selected grid or select **GridAppearance** in the toolkit. With it you can change the appearance of the grid and its kids.

When you change color, border style, text, or other property in the AppearanceWindow, the grid changes to reflect the new setting. So it's easy to fine tune your grids to get exactly the look you want.

Grid Behavior

To display a *Behavior Window*, select **GridBehavior** in the toolkit. This window lists the messages a grid processes to give its behavior. Unlisted messages are ignored. Next to every message is the message processing function and/or message processing subroutine that the grid function calls to process the message.

Design Mode vs Test Mode

At any point during design, you can toggle the design window back and forth between *DesignMode* and *TestMode* with the CheckBox at the top of the toolkit.

In design mode you select grids from the toolkit, lay them out in the design window, and assign appearance properties to them.

In test mode, grids in the design window will operate much as they will in your program. Actions in the design window that would cause callback messages print information to the console window for preliminary inspection.

WindowToFunction

When you want to convert a design window into a grid function and callback function, select **WindowToFunction** in the toolkit Menu.

GuiDesigner creates code to create, operate, and respond to the window, and adds it to your program, or updates the current contents.

The callback function is displayed in the lower text area of the program development environment.

Quick Start

Believe it or not, you can run your program right away, even though you haven't written a line of code. *GuiDesigner* puts a line of code in callback functions to display the arguments of every message it receives in a ReportMessage window. So go ahead. Click the Start button to compile and execute your program. When your design windows appear, play with it and watch the ReportMessage window.

Callback Arguments

The primary arguments to the callback function are the **Callback** message in **message**, the original message in **r1**, and a kid number in **kid** or **r0**. Most callbacks contain a **Selection** message in **r1**. The kid number in **kid** or **r0** tells which kid initiated the callback. To respond to a particular event, add code to the callback function to process the arguments you see printed in the Report window.

Activate a grid in the window, see what's printed in the console, then add code to the code function to process it. When you add code to callback functions, be sure to write kid number constants, not the literal kid numbers printed in the console. The kid number constants are defined near the top of callback functions.

You'll probably execute different code for each **message**, **kid**.



ReportMessage		
wing-id	823	Math
message	22	Callback
r0	-1	FFFFFFFF
r1	0	00000000
r2	0	00000000
r3	0	00000000
kid-r0	23	00000017
ANY r1	171	Selection <message>

WindowFromFunction

When you exit the development environment, your design windows are lost. You can reload any program you have saved, of course, but the design windows no longer exist. The grid functions *GuiDesigner* created from your design windows are part of your program, but not the design windows.

So how do you tweak your original designs if you ever want to?

You can edit the grid function, which is not hard, since grid functions are too easy to read. Better yet, select **WindowFromFunction** in the toolkit Menu and let *GuiDesigner* convert the grid function back into a design window.

That's right! *GuiDesigner* reads the text in your grid function, figures out what it means, and recreates a design window like the one that generated the grid function in the first place.

GuiDesigner looks at the kid constant definitions, the **Create** subroutine, plus four lines of code near the bottom that assign values to **designX,designY,designWidth,designHeight**. This is all the information necessary to recreate the design window.

Any custom code you've added, like a **Resize** routine for example, does not affect the design window. When the design window is converted back into a grid function, code outside the **Create** subroutine is not modified if you select **Update**. Any code you added is lost if you specify **Replace** - be careful !!!

Be sure any code you add to grid functions and callback functions contains kid number constants, not literal kid numbers. Kid numbers may change when you change the layout of a design window and convert it back into code, while kid names don't change unless you change them purposely in the Appearance Window.

WindowFromFunction stops at the first blank or comment line in the **Create** subroutine, so don't add any unless that's your intent.

Operating Grid Functions

To create a window that contains a grid you designed, your program calls its grid function, passing it a **CreateWindow** message and other pertinent arguments. The grid function creates a window containing one of your grids and returns its **grid** number.

```
Design ( @grid, #CreateWindow, xDisp, yDisp, width, height, 0, 0 )
```

To request notification of important events from the grid, your program sends a **SetCallback** message:

```
XuiSendMessage ( grid, #SetCallback, grid, &DesignCode(), -1,-1,0,0 )
```

To control the window containing a grid, your program sends messages to the **grid**. For example, to control a window:

```
XuiSendMessage ( grid, #DisplayWindow, 0, 0, 0, 0, 0, 0 )
XuiSendMessage ( grid, #HideWindow, 0, 0, 0, 0, 0, 0 )
XuiSendMessage ( grid, #IconifyWindow, 0, 0, 0, 0, 0, 0 )
```

There are literally dozens of messages your program can send to **grid** and its kids to get their current properties and states, for example:

```
XuiSendMessage ( grid, #GetBorder, @b, @bu, @bd, 0, kid, @width )
XuiSendMessage ( grid, #GetColor, @back, @draw, @hi, @lo, kid, 0 )
XuiSendMessage ( grid, #GetTextString, 0, 0, 0, 0, kid, @text$ )
XuiSendMessage ( grid, #GetTextArray, 0, 0, 0, 0, kid, @text$[] )
```

There are also dozens of messages your program can send to **grid** and its kids to set new properties and states, for example:

```
XuiSendMessage ( grid, #SetBorder, b, bu, bd, -1, kid, 0 )
XuiSendMessage ( grid, #SetColor, back, draw, -1, -1, kid, 0 )
XuiSendMessage ( grid, #SetTextString, 0, 0, 0, 0, kid, @text$ )
XuiSendMessage ( grid, #SetTextArray, 0, 0, 0, 0, kid, @text$[] )
```

The grid functions *GuiDesigner* creates from your design windows are *your* functions. You can view them, learn from them, even modify them, though you probably won't since there's rarely any need.

To Code or Not To Code

In most cases, you can design a GUI without writing a line of code. You might not even have to look at the code *GuiDesigner* wrote. To interact with the GUI, all your program does is send messages to grid functions and receive messages from grid functions.

What messages do your programs send to achieve a particular result? The same ones *GuiDesigner* puts in the grid functions it creates from your design windows. Which means there's an easy way to learn how to do it yourself.

Learning Curve

Let *GuiDesigner* show you. Build a window interactively, then look at the code *GuiDesigner* generates from it. Change the design a little and look again. Change it again and peek again. Learn as you go.

The code *GuiDesigner* puts in the `create` subroutines is the same as the code you need in your program to control grids at runtime. Which means you can make *GuiDesigner* do your work for you.

Say you need code to change the color and text on an `XuiLabel` grid at runtime. Create a design window, give it a name, select an `XuiLabel` from the toolkit, bring up the AppearanceWindow, and give the `XuiLabel` new colors and a new text string.

Then select `WindowToFunction` to make *GuiDesigner* create a grid function from the design window. You'll have no trouble finding the code that accomplishes what you want in the `create` subroutine. Copy the relevant lines, paste them into your function, then delete the cheater window and function. Who needs 'em? You got what you wanted. Right?

By stealing code *GuiDesigner* writes, you can learn just about everything there is to know about writing code to control grids at runtime. It's not even a misdemeanor in most jurisdictions! After a few capers you'll see how simple runtime code is, and you'll realize how easy it is to write yourself. You'll be a pro in no time.

Instant Help

InstantHelp

GuiDesigner programs are typically easy to learn. That's because *GuiDesigner* builds an automatic, user-extensible help mechanism into every grid in every GUI program you develop. You don't even have to write a single line of code.

Help On Everything

Users can click the right mouse button on *any* grid to learn about the grids purpose and operation. *GuiDesigner* displays an *InstantHelp* window that displays information you prepared for the occasion.

Posting InstantHelp

Press the HelpButton to display the *InstantHelp* window. Release it while the mouse cursor is in the same grid to post the *InstantHelp* window and keep it visible. Release the HelpButton over another grid and the *InstantHelp* window disappears.

Browsing Programs

To browse an entire program, users can hold down the HelpButton and drag the mouse cursor over every grid in the main window. Whenever the mouse cursor enters a new GUI component, the old *InstantHelp* window is instantly replaced by a new one. What a fast and easy way to learn a new program !!!

Solve the "Great Icon Problem"

What a great way around the "great icon problem" - the difficulty of finding a really expressive image for every grid in your program. Just point and click the right button for *InstantHelp*.

Copy from the InstantHelp Window

And that's just for starters. *Users can edit the InstantHelp window!*

Users can copy text from the *InstantHelp* window and insert it into any text component in their GUI. Which means they can extract samples from your help text and try 'em out. And that's not all.

Update Instant Help

Users can change help text. They can copy, delete, insert, and type in the *InstantHelp* window just like any other TextArea. So they can expand, contract, and reword documentation in any way they want. When they press the Update button, the help files are updated. Henceforth *InstantHelp* displays new, improved information.

Help Files

The information displayed in the HelpWindow usually comes from a simple text file called a *HelpFile*. The default *HelpFile* starts out the same as your program, except it ends with ".hlp". You shouldn't rely on this name, however, because you might rename your program someday and lose all your help! Call `XuiSetHelpFile()` to explicitly set the default *HelpFile*.

HelpFile Format

Each *HelpFile* contains one or more help *entries*. Each entry starts with a heading line and ends just before the next heading line. Any line that begins with a ":" character is assumed to be a heading line. Two complete entries and the beginning of a third are shown in the following example:

```
:AcceptIdentity
Press "Accept" to confirm the name in
the TextLine grid. This will be your
working identity for this mission.

:RetryIdentity
Press "Retry" to request a new name.
You must accept one of the first three
names offered or you will be terminated.

:AbortMission
```

HelpString

When you design GUIs interactively, you select grids from the toolkit, lay them out in design windows, and set their initial states with the *AppearanceWindow*. One of the properties you can set for each grid is called its *HelpString*, which has one of the following three formats:

```
*** format ***           *** example ***
filename:entryname      mission.hlp:AcceptIdentity
:entryname              :RetryIdentity
"help text string"     "[Abort]\nPrepare to die, Bond."
```

`filename:entryname` explicitly specifies the `filename` of the *HelpFile* to search and the `:entryname` of the help entry within it.

`:entryname` specifies the `:entryname`, but not the *HelpFile*. The default *HelpFile* is searched for the specified `:entryname`.

"`help text string`" is one or more lines of help text, with newline characters ("\n") separating lines. This format supplies help text directly, so no *HelpFile* is searched. If the first line of the entry begins with a "[" character, the first line is displayed on the title label above the *InstantHelp* window in place of `filename:entryname`.

:entryname

Be sure to choose `:entryname` strings carefully. Simple names like `:Retry` aren't very descriptive, and you'll run into problems when you create another "Retry" button. The same help text would be displayed for both buttons.

Remember, `filename:entryname` is the title displayed in the *InstantHelp* window for the world to see, so choose an appropriate title!

Default :entryname

In many programs, you don't need to assign help string names. When help is requested on a grid and *GuiDesigner* finds an empty help string, it synthesizes one by prefixing ":" to the `gridName` property. For example, if a `GridName` is "CancelButton", the synthesized `:entryname` is `":CancelButton"`.

As long as you name your grids intelligently, which is very important anyway, you need not assign help strings to grids. Only where this proves inadequate are explicit help strings necessary. Make sure you don't give the same grid name to two or more grids !

Multiple HelpFiles

Most programs assign `"filename:entryname"` help strings to their grids because it's easy and absolutely prevents confusion. Some programs assign `":entryname"` help strings to their grids to support "personalized" help text stored in multiple helpfiles.

Not so much to support:

```
CARL.HLP      - help for Carl
MARY.HLP      - help for Mary
FRED.HLP      - help for Fred
BILL.HLP      - help for Bill
```

More likely to support:

```
EXPERT.HLP    - help for advanced users
NORMAL.HLP    - help for typical users
NOVICE.HLP    - help for beginners
```

Or perhaps to support:

```
MARKET.HLP    - help for marketing wizards
ENGINE.HLP    - help for engineer wizards
BOSSSES.HLP   - help for administrators
```

Set HelpFile

Programs can call `XuiHelp()` to establish a specific *HelpFile* as follows:

```
XuiHelp (grid, #SetHelpFile, 0, 0, 0, 0, 0, 0, @"helpfile.hlp")
```

`XuiHelp()` accepts any valid grid number to set the *HelpFile*. Programs that support multiple *HelpFiles* can let users select their own category of help from pull down menus or radio buttons.

Messages

Messages

Messages are a fundamental part of *GuiDesigner* and most programs that interact with GUIs. *GuiDesigner* communicates within itself, and with your programs, by sending and receiving messages. To communicate with window functions, grid functions, and *GuiDesigner*, your programs must also send and receive messages.

Processing messages is what animates and gives life to GUIs.

GraphicsDesigner Messages

When an event occurs in a graphics window, *GraphicsDesigner* creates a *message* to describe it. When a keyboard key is pressed, a `WindowKeyDown` message is created. When the mouse moves or one of its buttons is pressed, a `WindowMouseMove` or `WindowMouseDown` message is created. When a user clicks on an inactive window, a `WindowSelected` message is created.

In general, whenever the state of the keyboard, mouse, or a window changes, *GraphicsDesigner* creates a message that describes it. *GuiDesigner* programs rarely expect window messages such as these. That's because window functions convert window messages into corresponding grid messages like `KeyDown`, `MouseMove`, `MouseDown` and sends them to the appropriate grid.

Window Messages and Grid Messages

Any message, like `WindowKeyDown` or `TimeOut`, is a window message or a grid message, *never both*.

Window Messages

- Names *must* start with "Window".
- First argument is a window number, `window`.
- Are sent to window functions by `XuiSendMessage()`.

The following are typical window messages:

<code>WindowDeselected</code>	<code>WindowHide</code>	<code>WindowMinimize</code>
<code>WindowDisplayed</code>	<code>WindowHidden</code>	<code>WindowResize</code>
<code>WindowDestroy</code>	<code>WindowKeyDown</code>	<code>WindowSelect</code>
<code>WindowDestroyed</code>	<code>WindowKeyUp</code>	<code>WindowSelected</code>

Grid Messages

- Names *must not* start with "Window".
- First argument is a grid number, `grid`.
- Are sent to grid functions by `XuiSendMessage()`.

Message Anatomy

Every message contains 8 values, or arguments:

```
( window, message, v0, v1, v2, v3, r0, r1 )
... or ...
( grid, message, v0, v1, v2, v3, kid, r1 )
... or ...
( wingrid, message, v0, v1, v2, v3, r0, r1 )
( wingrid, message, v0, v1, v2, v3, kid, r1 )
```

GraphicsDesigner messages contain only **XLONG** arguments.

The first 7 arguments of *GuiDesigner* messages are always **XLONG** too, but the last may be **XLONG**, **STRING**, or an array of any valid type. The type of the last argument must be the type appropriate to the **message** argument.

window, grid, wingrid

- **window** contains the window number the message refers to.
- **grid** contains the grid number the message refers to.
- **wingrid** contains the window or grid the message refers to.

wingrid is the name given to arguments that contain a window number in some contexts and a grid number in others.

When a message refers to one of the kids of **grid**, it is identified by **grid,kid**. When **kid=0** the message refers to **grid** itself.

message

message contains a message number, a numeric stand-in for an original message name like "KeyDown". Most programs communicate with message numbers exclusively because that's what messages contain. Message numbers are assigned to equivalent **XLONG** message variables like **#KeyDown** in **InitGui()** during initialization.

v0,v1,v2,v3,r0,r1

v0,v1,v2,v3,r0,r1 are general purpose arguments whose meanings depend on **message**. In *GraphicsDesigner* mouse messages, for example, they contain **xWin,yWin,state,time,0,grid**.

In some messages, one or more argument contains no defined value. *GraphicsDesigner* and *GuiDesigner* always fill these arguments with zero and expect other programs to do the same.

kid

r0 aka **kid** is a general purpose argument that always contains a **kid** number in grid messages.

GraphicsDesigner Messages

The messages in the following tables are added to the message queue by *GraphicsDesigner*, and sent to window functions when they are processed by `XgrProcessMessages()`. No function is called by `XgrProcessMessages()` if the window the message relates to has no window function.

window and grid messages are distinguished by `w` or `g` before the message name.

<code>g</code>	<code>MouseDown</code>	A mouse button was depressed.
<code>g</code>	<code>MouseDown</code>	A mouse button was down when the mouse moved.
<code>g</code>	<code>MouseEnter</code>	A mouse cursor entered a grid.
<code>g</code>	<code>MouseExit</code>	A mouse cursor exited a grid.
<code>g</code>	<code>MouseMove</code>	A mouse cursor moved.
<code>g</code>	<code>MouseUp</code>	A mouse button was released.
<code>g</code>	<code>Redraw</code>	A grid should redraw itself and have its kids redraw themselves.
<code>g</code>	<code>RedrawGrid</code>	A grid should redraw itself (but not its kids).
<code>g</code>	<code>TimeOut</code>	A grid timer has timed out (counted down to zero).
<code>w</code>	<code>WindowDeselected</code>	A window was deselected (because another was selected).
<code>w</code>	<code>WindowDestroyed</code>	A window was destroyed (by user or program).
<code>w</code>	<code>WindowDisplayed</code>	A window was just displayed (became visible).
<code>w</code>	<code>WindowHidden</code>	A window was just hidden (is no longer visible).
<code>w</code>	<code>WindowKeyDown</code>	A keyboard key was depressed with keyboard focus in window.
<code>w</code>	<code>WindowKeyUp</code>	A keyboard key was released with keyboard focus in window.
<code>w</code>	<code>WindowMaximized</code>	A window was maximized (asked to fill the screen).
<code>w</code>	<code>WindowMinimized</code>	A window was minimized (is now a small icon).
<code>w</code>	<code>WindowMouseDown</code>	A mouse button was depressed with mouse focus in window.
<code>w</code>	<code>WindowMouseDrag</code>	A mouse button was down when mouse moved with mouse focus in window.
<code>w</code>	<code>WindowMouseEnter</code>	A mouse cursor entered a grid in the window with mouse focus.
<code>w</code>	<code>WindowMouseExit</code>	A mouse cursor exited a grid in the window with mouse focus.
<code>w</code>	<code>WindowMouseMove</code>	A mouse cursor moved in the window with mouse focus.
<code>w</code>	<code>WindowMouseUp</code>	A mouse button was released with mouse focus in window.
<code>w</code>	<code>WindowRedraw</code>	A window was exposed partially or totally and needs redrawing.
<code>w</code>	<code>WindowResized</code>	A window was moved and/or resized (its grids might need resizing).
<code>w</code>	<code>WindowSelected</code>	A window was selected (user pressed mouse button on window).

<code>g</code>	<code>MouseDown</code>	(grid, #MouseDown, x, y, state, time, 0, grid)
<code>g</code>	<code>MouseDown</code>	(grid, #MouseDown, x, y, state, time, 0, grid)
<code>g</code>	<code>MouseEnter</code>	(grid, #MouseEnter, x, y, state, time, 0, grid)
<code>g</code>	<code>MouseExit</code>	(grid, #MouseExit, x, y, state, time, 0, grid)
<code>g</code>	<code>MouseMove</code>	(grid, #MouseMove, x, y, state, time, 0, grid)
<code>g</code>	<code>MouseUp</code>	(grid, #MouseUp, x, y, state, time, 0, grid)
<code>g</code>	<code>Redraw</code>	(grid, #Redraw, x, y, width, height, 0, grid)
<code>g</code>	<code>RedrawGrid</code>	(grid, #RedrawGrid, x, y, width, height, 0, grid)
<code>g</code>	<code>TimeOut</code>	(grid, #TimeOut, 0, 0, 0, 0, 0, grid)
<code>w</code>	<code>WindowDeselected</code>	(window, #WindowDeselected, 0, 0, 0, 0, 0, window)
<code>w</code>	<code>WindowDestroyed</code>	(window, #WindowDestroyed, 0, 0, 0, 0, 0, window)
<code>w</code>	<code>WindowDisplayed</code>	(window, #WindowDisplayed, 0, 0, 0, 0, 0, window)
<code>w</code>	<code>WindowHidden</code>	(window, #WindowHidden, 0, 0, 0, 0, 0, window)
<code>w</code>	<code>WindowKeyDown</code>	(window, #WindowKeyDown, x, y, state, time, 0, window)
<code>w</code>	<code>WindowKeyUp</code>	(window, #WindowKeyUp, x, y, state, time, 0, window)
<code>w</code>	<code>WindowMaximized</code>	(window, #WindowMaximized, 0, 0, 0, 0, 0, window)
<code>w</code>	<code>WindowMinimized</code>	(window, #WindowMinimized, 0, 0, 0, 0, 0, window)
<code>w</code>	<code>WindowMouseDown</code>	(window, #WindowMouseDown, x, y, state, time, 0, grid)
<code>w</code>	<code>WindowMouseDrag</code>	(window, #WindowMouseDrag, x, y, state, time, 0, grid)
<code>w</code>	<code>WindowMouseEnter</code>	(window, #WindowMouseEnter, x, y, state, time, 0, grid)
<code>w</code>	<code>WindowMouseExit</code>	(window, #WindowMouseExit, x, y, state, time, 0, grid)
<code>w</code>	<code>WindowMouseMove</code>	(window, #WindowMouseMove, x, y, state, time, 0, grid)
<code>w</code>	<code>WindowMouseUp</code>	(window, #WindowMouseUp, x, y, state, time, 0, grid)
<code>w</code>	<code>WindowRedraw</code>	(window, #WindowRedraw, x, y, width, height, 0, window)
<code>w</code>	<code>WindowResized</code>	(window, #WindowResized, x, y, width, height, 0, window)
<code>w</code>	<code>WindowSelected</code>	(window, #WindowSelected, 0, 0, 0, 0, 0, window)

Keyboard Messages

Keyboard messages contain the `window` number of the window that was selected when the keyboard event was detected. `v0,v1,v2,v3` contain `xWin,yWin,state,time`.

Keyboard Focus

Window functions generally convert `WindowKeyDown` and `WindowKeyUp` messages into equivalent grid messages `KeyDown` and `KeyUp` and send them to the grid that currently has keyboard focus.

Window functions generally move keyboard focus from grid to grid in response to `Alt+LeftArrow` and `Alt+RightArrow` keystrokes.

`xWin, yWin`

`xWin,yWin` may be the position of the mouse cursor in the window coordinates of the selected `window` at the time the keyboard event was detected. If `xWin` or `yWin` is negative, the mouse cursor was outside the selected window, or the position of the mouse was unavailable.

`state`

`state` contains the state of the keyboard when the keyboard event was detected, and reflects the new state of the keyboard.

```
Bit 00 - 15 : Character code of some kind (see bits 20-22)
Bit 16 - 23 : Keyboard "mode" keys (16=Shift, 17=Control, 18=Alt)
Bit 24 - 31 : Virtual Key Code
```

```
Bit 16 = 1 : Shift key was down when the keyboard event occurred.
Bit 17 = 1 : Control key was down when the keyboard event occurred.
Bit 18 = 1 : Alt key was down when the keyboard event occurred.
Bit 19      : Reserved
Bit 20 - 22 : Type of character code in Bit 00 - 15 (see below)
Bit 20 - 22 : 0 = Bit 00 - 15 = Virtual Key Code (8-bits)
              : 1 = Bit 00 - 15 = ASCII character (8-bits)
              : 2 = Bit 00 - 15 = WIDE character (16-bits)
              : values 3 to 7 are reserved
```

`time`

`time` contains the system time that the keyboard event was detected. `time` is not related to time of day. `time` is a free running millisecond timer that computer systems usually initialize to zero when they start.

Examples

Key	20-22	18	17	16	0-15	."	mode key states, key event
Down	1	0	0	0	97	a	None down, "a" down
Up	0	0	0	0	97	-	None down, "a" up
Down	1	0	0	1	65	A	Shift down, "a" down
Up	0	0	0	1	65	-	Shift down, "a" up
Down	0	0	1	0	65	^A	Ctl down, "a" down
Down	0	0	1	1	65	?	Ctl+Shift down, "a" down
Down	0	1	0	0	65	?	Alt down, "a" down
Down	0	1	0	1	65	?	Alt+Shift down, "a" down
Down	0	1	1	0	65	?	Alt+Ctl down, "a" down
Down	0	1	1	1	65	?	Alt+Ctl+Shift down, "a" down
Up	0	1	1	1	65	?	Alt+Ctl+Shift down, "a" up
Down	0	0	1	0	39	Left	Ctl down, LeftArrow press

WindowKeyUp vs WindowKeyDown

Most programs respond only to `WindowKeyDown` messages, since that is sufficient to respond to all keystrokes. `WindowKeyDown` messages are created at a rate of about 10 per second when a key is held down, followed by a single `WindowKeyUp` when the key is finally released.

Virtual Key Codes

8	0x08	KeyBackspace	48	0x30	Key0	96	0x60	KeyPad0
9	0x09	KeyTab	49	0x31	Key1	97	0x61	KeyPad1
12	0x0C	KeyClear	50	0x32	Key2	98	0x62	KeyPad2
13	0x0D	KeyEnter	51	0x33	Key3	99	0x63	KeyPad3
16	0x10	KeyShift	52	0x34	Key4	100	0x64	KeyPad4
17	0x11	KeyControl	53	0x35	Key5	101	0x65	KeyPad5
18	0x12	KeyAlt	54	0x36	Key6	102	0x66	KeyPad6
19	0x13	KeyPause	55	0x37	Key7	103	0x67	KeyPad7
20	0x14	KeyCapLock	56	0x38	Key8	104	0x68	KeyPad8
27	0x1B	KeyEscape	57	0x39	Key9	105	0x69	KeyPad9
32	0x20	KeySpace	65	0x41	KeyA	106	0x6A	KeyPadMultiply
33	0x21	KeyPageUp	66	0x42	KeyB	107	0x6B	KeyPadAdd
34	0x22	KeyPageDown	67	0x43	KeyC	108	0x6C	_
35	0x23	KeyEnd	68	0x44	KeyD	109	0x6D	KeyPadSubtract
36	0x24	KeyHome	69	0x45	KeyE	110	0x6E	KeyPadDecimalPoint
37	0x25	KeyLeftArrow	70	0x46	KeyF	111	0x6F	KeyPadDivide
38	0x26	KeyUpArrow	71	0x47	KeyG	112	0x70	KeyF1
39	0x27	KeyRightArrow	72	0x48	KeyH	113	0x71	KeyF2
40	0x28	KeyDownArrow	73	0x49	KeyI	114	0x72	KeyF3
44	0x2C	KeyPrintScreen	74	0x4A	KeyJ	115	0x73	KeyF4
45	0x2D	KeyInsert	75	0x4B	KeyK	116	0x74	KeyF5
46	0x2E	KeyDelete	76	0x4C	KeyL	117	0x75	KeyF6
47	0x2F	KeyHelp	77	0x4D	KeyM	118	0x76	KeyF7
			78	0x4E	KeyN	119	0x77	KeyF8
			79	0x4F	KeyO	120	0x78	KeyF9
			80	0x50	KeyP	121	0x79	KeyF10
			81	0x51	KeyQ	122	0x7A	KeyF11
			82	0x52	KeyR	123	0x7B	KeyF12
			83	0x53	KeyS	124	0x7C	KeyF13
			84	0x54	KeyT	125	0x7D	KeyF14
			85	0x55	KeyU	126	0x7E	KeyF15
			86	0x56	KeyV	127	0x7F	KeyF16
			87	0x57	KeyW	144	0x90	KeyNumLock
			88	0x58	KeyX			
			89	0x59	KeyY			
			90	0x5A	KeyZ			

Mouse Messages

Mouse messages contain the `grid` number of the grid with mouse focus when the event was detected. The grid with mouse focus is generally the grid the mouse cursor is in, except in the following situation. When the state of the mouse buttons changes from no buttons depressed to one or more depressed, the grid that contains the mouse cursor grabs mouse focus and keeps it until all mouse buttons are released. `v0,v1,v2,v3` contain `x,y,state,time`.

`x,y`

`x,y` contain the local coordinates of the mouse cursor in grid at the time the mouse event was detected. `x,y` may indicate a mouse cursor position outside the grid if mouse focus has been grabbed.

`state`

```
Bit 00 - 03 : Button # causing event (MouseDown and MouseUp only )
Bit 04 - 06 : # of clicks (MouseDown only)
Bit      07 : 1 if grid has mouse focus
Bit 08 - 15 : Reserved
Bit 16 - 23 : Keyboard "mode" keys (16=Shift, 17=Control, 18=Alt)
Bit 24 - 31 : Up/Down image of up to 8 mouse buttons (1 = down)
```

```
Bit 00 - 03 : Button #: None=0 : Left=1 : Center=2 : Right=3...
Bit 16 = 1  : Shift key is down
Bit 17 = 1  : Control key is down
Bit 18 = 1  : Alt key is down
Bit 24 = 1  : Left button is down
Bit 25 = 1  : Center button is down
Bit 26 = 1  : Right button is down
Bit 27 - 31 : Other buttons down (assignments not guaranteed)
```

`time`

`time` contains the system time that the keyboard event was detected. `time` is not related to time of day. `time` is a free running millisecond timer that computer systems usually initialize to zero when they start.

Mouse Message Algorithm

Your program doesn't receive a mouse message for every change in mouse position the system detects. When your task is too busy to process messages, *GraphicsDesigner* does not flood your message queue with `#MouseMove` messages. Instead, *GraphicsDesigner* adds `#MouseMove` messages to the message queue only when the message queue is empty. All other mouse messages, including `#MouseDown`, are added to the message queue whether it has contents or not.

This method of generating mouse messages is best for most programs, and avoids overflowing your message queue with superfluous `#MouseMove` messages. In rare instances, however, your program may want to receive every mouse message. To achieve high-speed mouse position tracking, programs must access the message queue often. This keeps the message queue empty and assures new mouse messages are put in the queue immediately. Programs can achieve high-speed tracking by responding to messages quickly and returning to the main message loop, or by processing them often with:

`XgrProcessMessages (0)`

`XgrProcessMessages (0)` returns immediately if there are no messages in the message queue and the mouse state has not changed. Otherwise it processes one message, then returns.

Be careful with this technique, however. If your program processes a new `#MouseMove` message while it's in the middle of processing the previous `#MouseMove`, will your program produce adverse effects when it finishes processing the earlier `#MouseMove`?

Message Queue

GraphicsDesigner combines message arguments into a message and adds it to its *message queue*. Messages wait in the message queue until your program is ready to process them.

In summary, *GraphicsDesigner* detects events, makes messages to describe them, and puts them in the message queue - automatically and invisibly, without the knowledge or support of your programs.

Process Message

This is where your program takes over. Your programs must process the messages *GraphicsDesigner* put in the message queue. Otherwise the messages will sit in the queue forever and your programs will be permanently dead and lifeless. Processing messages animates and gives life to your GUI programs. In short, processing messages is how GUI programs run themselves.

XgrProcessMessages ()

Programs call ***XgrProcessMessages (1)*** to process a message. ***XgrProcessMessages ()*** processes messages in the order they were added to the queue. First come, first served. Or in computer lingo, First-In, First-Out ... or FIFO.

The basic operation of GUI programs is almost unbelievably simple. Process a message completely. Then process the next one. Then the next, then the next, then the next... indefinitely. That's all there is!

Processing each *GraphicsDesigner* message is reacting to a keystroke, button click, or other user action. Processing messages is nothing more than obeying a series of user instructions, one by one. If it sounds like the user controls your program, you're right! It's the reason most people prefer GUI programs - they like to run the show.

Actually it's just a variety of the grand illusion. Users can only select from choices your program gives them. They're playing your game. But you don't have to tell them that, and they'll never catch on. People are conditioned not to recognize the grand scam.

Message Loop

GUI programs process messages in a *message loop*. The message loop is usually the last three lines in the entry function of GUI programs. In fact, the entry function of most GUI programs are almost identical to the following example:

```
FUNCTION Entry ()
  SHARED terminateProgram
  STATIC entered
  '
  IF entered THEN RETURN
  entered = $$TRUE
  '
  Xui ()           ' initialize GuiDesigner
  InitGui ()      ' initialize message variables
  InitProgram ()  ' initialize this program
  CreateWindows () ' create program windows
  InitWindows ()  ' initialize windows
  IF LIBRARY(0) THEN RETURN ' libraries don't execute message loop
  '
  DO              ' begin the message loop ...
    XgrProcessMessages ( 1 ) ' process one message ...
  LOOP UNTIL terminateProgram ' repeat until program is terminated
END FUNCTION
```

Entry() initializes *GuiDesigner* and itself, then creates, activates, and displays its main window. Finally it falls into its message loop, where it stays until the program is terminated.

The message loop is the "base of operations" for GUI programs. Until the user operates the keyboard or mouse, programs sleep in **XgrProcessMessages(1)**. Computer time is given to other programs that have work to do.

When the user finally gets around to operating the GUI, *GraphicsDesigner* creates a message, puts it in the queue, and wakes up **XgrProcessMessages(1)**.

XgrProcessMessages(1) processes a message, then **LOOP** loops back to **XgrProcessMessages(1)**, where it awaits the next message.

Only when your program assigns a non-zero value to shared variable **terminateProgram** does the message loop end, usually followed by the entry function and the program.

But how does **XgrProcessMessages(1)** "process a message"?

What does it mean, to "process a message"?

Process a Message

To process a message, `XgrProcessMessages()` calls the window function the message refers to directly or indirectly. For window messages, `XgrProcessMessages()` calls the window function assigned to the `window` argument. For grid messages, `XgrProcessMessages()` calls the window function assigned to the window that contains the `grid` argument.

`XgrProcessMessages()` passes the 8 `XLONG` message arguments to the functions it calls. All window functions accept 8 arguments. But window functions sometimes expect to receive other types from *GuiDesigner* and your program. So the last argument of window functions is declared to be `ANY` type to also accept `STRING` variables and all types of arrays.

Window functions are usually declared and defined as follows:

```
DECLARE FUNCTION Name ( wingrid, message, v0, v1, v2, v3, r0, ANY )
FUNCTION Name ( wingrid, message, v0, v1, v2, v3, r0, (r1, r1$) )
```

`XgrProcessMessages()` calls window functions as follows:

```
@func ( wingrid, message, v0, v1, v2, v3, r0, r1 )
```

Window Function

Every window is assigned a *window function* when it is created. In response to `CreateWindow` messages, most grid functions assign standard window function `XuiWindow()`.

Most GUI programs rely entirely on the standard window function supplied with *GuiDesigner*, `XuiWindow()`. Only when special message processing is necessary and cannot be handled in other ways should you write custom window functions. Custom window functions should only contain code to perform actions that existing standard window functions do not provide.

Messages not requiring custom processing should be passed on to the existing standard window function for default processing. Those that do require custom processing should be passed to an existing standard window function either before or after custom processing, whichever is appropriate. Exceptions are possible, but not common.

Window Functions Process Window Messages

When a window function receives a *window message*, it can:

- *Ignore it and return.*
- *Take action, then return.*
- *Pass to standard window function, then return.*
- *Pass to standard window function, then take action, then return.*
- *Take action, then pass to standard window function, then return.*

Standard window functions generally take action on window messages, then return. Custom window functions pass most messages to a standard window function then return. For messages that require non-standard actions, custom window functions generally perform their custom action either before or after passing the messages to a standard window function.

To process some window messages, window functions have to send messages to grid functions. For example, in response to `WindowKeyDown` and `WindowKeyUp` messages, window functions usually send `KeyDown` and `KeyUp` messages to the grid with keyboard focus.

Window Functions Process Grid Messages

When a window function receives a *grid message*, it can:

- *Ignore it and return.*
- *Take action, then return.*
- *Pass to standard window function then return.*
- *Pass to standard window function, take action, then return.*
- *Take action, pass to standard window function, then return.*
- *Pass to the grid function responsible for operating the grid, then return.*

In almost every case, standard and custom window functions pass grid messages to the grid function responsible for operating the grid specified by the grid argument in the message, then return.

Grid Function

Every grid is created, operated, and destroyed by its *grid function*. Each grid function is responsible for every grid of its grid type. Grid functions give each grid type its individual characteristics.

You'll probably never write a grid function. Grid functions are provided with *GuiDesigner* for popular grids, and *GuiDesigner* will create grid functions for other purposes from design windows you develop interactively and graphically. But you may want to modify the grid functions *GuiDesigner* creates for you - to make your custom windows and grids resizable, or to give special purpose behavior to the new grid types you developed.

Grid functions are usually declared and defined as follows:

```
DECLARE FUNCTION Name ( grid, message, v0, v1, v2, v3, kid, ANY )  
FUNCTION Name (grid, message, v0, v1, v2, v3, kid, (r1, r1$, r1$[]))
```

Grid functions typically ignore dozens of grid messages, and process dozens of others. The character of each grid type is determined by which messages its grid function ignores, and what it does to process the messages it recognizes. To ignore messages, grid functions simply return.

To process most messages, grid functions call standard message processing functions provided with *GuiDesigner*. The standard message processing functions look like message names with an **Xui** prefix, for example **XuiGetBorder()**, **XuiGetColor()**, **XuiSetTextString()**, **XuiRedrawGrid()** etc.

To process messages in a way not supported by standard functions, grid functions either execute a custom message processing subroutine within the grid function, or call a non-standard message processing function written to perform the new activity.

Grid functions ignore messages containing invalid **grid** or **message** arguments, and attempt to ignore messages containing other invalid arguments, though some are difficult to detect.

Send Message

To *send a message* simply means to call a window or grid function, or any other function designed to take window and/or grid messages.

To *send a message to a window* means to call the window function assigned to the `window` specified in the message. That's what `XgrProcessMessages()` does when it processes a message.

To *send a message to a grid* means to call the grid function assigned to the `grid` specified in the message. That's what window functions do to process grid messages.

XuiSendMessage()

The best way to send a message is to pass the whole message to `XuiSendMessage()` and let it look up and call the window or grid function assigned to the `window` or `grid` in the message:

```
XuiSendMessage ( wingrid, message, v0, v1, v2, v3, r0, r1 )
XuiSendMessage ( window, message, v0, v1, v2, v3, r0, r1 )
XuiSendMessage ( grid, message, v0, v1, v2, v3, kid, r1 )
```

`XuiSendMessage()` looks at `message` to determine whether each message is a window message or grid message, which determines whether the first argument is `window` or `grid`. This makes it possible for `XuiSendMessage()` to call the correct window or grid function.

`XuiSendMessage()` passes **8** message arguments to the functions it calls, so all window and grid functions must accept **8** `XLONG` arguments. The last argument of window and grid functions are declared to accept **ANY** data type so they can also receive a `STRING` variable, composite variable, or an array of any valid type.

Your programs can pass any combination of the `v0,v1,v2,v3,r0,r1` arguments by reference to receive return values from the function called by `XuiSendMessage()`.

`XuiSendMessage()` validates the `grid,message,kid` arguments, looks up the `grid` function and calls it, passing all 8 arguments. The last 6 arguments are passed by reference so those arguments passed to `XuiSendMessage()` by reference can be returned to the calling function.

Runtime Messages

To interrogate or control a grid at runtime, *GuiDesigner* and your program send a message to the grid, just like window functions do:

```
XuiSendMessage ( grid, message, v0, v1, v2, v3, kid, r1 )
XuiSendMessage ( grid, message, v0, v1, v2, v3, kid, r1$ )
XuiSendMessage ( grid, message, v0, v1, v2, v3, kid, r1$[] )
```

grid,**kid** are the grid number of the base grid and the kid within it the message is sent to. **kid=0** designates the base grid. **message** is the message sent to **grid**,**kid**.

v0,**v1**,**v2**,**v3**,**r1**,**r1\$**,**r1\$[]** are values passed to the grid function in support of **message**. For many messages, some or all of these arguments are undefined. **0** must be passed for all undefined arguments. For some messages one or more of these arguments are assigned return values by the grid function. To receive a return value the calling function must pass a variable by reference.

Example

For example your program can change the text string on the left XuiPushButton **kid** in a XuiDialog3B **grid** as follows:

```
XuiSendMessage ( grid, #SetTextString, 0, 0, 0, 0, 3, @" Hide " )
XuiSendMessage ( grid, #Redraw, 0, 0, 0, 0, 3, 0 )
```

grid contains the grid number of the XuiDialog3B grid.

#SetTextString is the message number variable that tells grids to set their text string to the value in the **r1\$** argument.

v0,**v1**,**v2**,**v3** are undefined in **SetTextString** messages, so they contain **0**.

kid=3 designates the left pushbutton in XuiDialog3B grids. Kids are numbered from zero, starting with the base grid itself, followed by its kids in the order created or as sorted by **GridSortKids**:

```
kid #0 = Dialog3B
kid #1 = Label
kid #2 = TextLine
kid #3 = LeftButton
kid #4 = CenterButton
kid #5 = RightButton
```

r1\$ contains the text string referred to by the **#SetTextString** message. Since only the final argument of grid functions can receive non-**XLONG** values, text strings are always passed in **r1\$**. Numeric arguments are passed in as many of the **v0**,**v1**,**v2**,**v3**,**r1** arguments as necessary.

Callback Messages

Grid functions process messages automatically, without bothering your program. In other words, grids operate themselves, independent of your program, without its knowledge or assistance.

On the other hand, sometimes events occur that your program needs to know about. For example, when you click on an `XuiPushButton` or press the Enter key in an `XuiTextLine` grid, your program usually needs to know it happened.

That's when grids send a *callback message*.

Callback Functions

Who receives callback messages from a particular grid?

The *callback function* most recently assigned to the grid as follows:

```
XuiSendMessage ( grid, #SetCallback, gridToCall, &FuncToCall(), -1, -1, -1, 0 )
```

This makes `FuncToCall()` the callback function for `grid`. Thereafter, when `grid` has a callback message to send, it calls:

```
FuncToCall ( gridToCall, message, v0, v1, v2, v3, kid, grid )
```

`FuncToCall()` and `gridToCall` are taken from the `SetCallback` message. `message` is the callback message that `grid` has decided to send, `v0,v1,v2,v3` are arguments in support of `message`, and `kid,grid` contain the grid and kid that was the initiator of the callback message.

When you click on an `XuiPushButton` in an `XuiDialog2B`, or type an Enter key into its `XuiTextLine`, `grid` sends a `Selection` message to `FuncToCall()`. What `FuncToCall()` or the functions it calls do is up to *your* program.

Monitor Messages

GuiDesigner processes *GraphicsDesigner* messages without the knowledge or assistance of your programs by sending them to the window function responsible for the window or grid in the message. Window functions process window messages and send grid messages to the appropriate grid functions.

This directs each message to the function designed to handle it, and nowhere else. Some programs, however, need to watch for certain events, no matter where or when they occur.

For example, many programs let the keyboard "function keys" perform specific actions, regardless of which window is selected. When processed by `XgrProcessMessages()`, however, keyboard messages for function keys are sent to the same place as any keyboard message - to the window function of the selected window.

To watch for special messages, *GraphicsDesigner* lets programs register one *CEO function*.

CEO Function

The *CEO function* receives every message processed by `XgrProcessMessages()`.

`XgrProcessMessages()` sends every message to the CEO function first, before it sends it anywhere else. The CEO can cancel the message to prevent its propagation to other functions.

Programs call `XgrSetCEO(&Func())`, to make a CEO function, active, where `&Func()` is the address of the CEO function. `XgrSetCEO(0)` cancels CEO processing.

When messages are processed by `XgrProcessMessages()` they are first sent to the CEO function, then to the appropriate window function. The window function may issue `MonitorContext`, `MonitorKeyboard`, `MonitorMouse` messages.

`XgrProcessMessages()` calls the CEO function, passing the message arguments, 0 in `r0`, and a copy of `window` or `grid` argument in `r1`. The CEO can examine the message and take whatever action it needs to perform its function. If the CEO returns -1 in `r0`, `XgrProcessMessages()` cancels the message and returns without calling the window function it would otherwise.

Slow Pokes

Some program let users initiate responses that take a *long* time to complete, like seconds instead of milliseconds. Users get antsy after a second or two. They expect programs to be *responsive*.

You'll probably want to let users break out of slow-poke responses. Users sometimes realize they entered a bad starting value and want to change it. Sometimes users didn't realize how long an operation would take and changed their mind. And sometimes they just plain pressed the wrong button!

But your program isn't going to see this "break" message until it returns to the message loop, right? And your program won't return to the message loop until it finishes its slow-poke response.

Wow. What a dilemma. Right?

Actually, it's no problem. Have your slow-poke functions process pending messages occasionally, perhaps this way:

```
XgrProcessMessages ( 0 )  
IF #breakOut THEN RETURN
```

XgrProcessMessages(0) returns immediately if no messages are waiting in the message queue, otherwise it processes one message and returns. If a processed message sets **#breakOut** and your function returns as a result, well, that solves delay and lock-up problems, doesn't it?

Advanced Message Processing

Most applications contain a call to **XgrProcessMessages(1)** in their entry function, and perhaps a couple slow-poke checks like the one described above. But occasionally sophisticated programs need to do something tricky. Fortunately, *GuiDesigner* message loops are accessible and a whole arsenal of message processing functions are available, including:

Message Processing...

```
XuiSendMessage ( grid, message, v0, v1, v2, v3, kid, r1 )  
XuiSendToKid ( grid, message, v0, v1, v2, v3, kid, r1 )  
XuiSendStringMessage ( grid, message$, v0, v1, v2, v3, kid, r1 )
```

Message Queue Processing...

```
XgrAddMessage ( grid, message, v0, v1, v2, v3, r0, r1 )  
XgrDeleteMessages ( count )  
XgrGetMessages ( @count, @messages[] )  
XgrGetMessageType ( message, @messageType )  
XgrJamMessage ( grid, message, v0, v1, v2, v3, r0, r1 )  
XgrMessageNameToNumber ( messageName$, @messageNumber )  
XgrMessageNames ( @count, messages$[] )  
XgrMessageNumberToName ( messageNumber, @messageName$ )  
XgrPeekMessage ( @grid, @message, @v0, @v1, @v2, @v3, @r0, @r1 )  
XgrProcessMessages ( count )  
XgrRegisterMessage ( @messageNumber, messageName$ )
```


Anatomy of Grid Functions

Overview

A *grid function* defines, controls, and supports the *grid type* with the same name. For instance, grid function `XuiLabel()` defines, controls, and supports all grids of `XuiLabel` grid type.

A grid function creates, operates, destroys every grid of its grid type. To send a message to a grid means to call its grid function. The way a grid function responds to the messages it receives determines the character of all grids of its grid type.

Grid Functions and Callback Functions

When you design a window interactively with *GuiDesigner*, then select `WindowToFunction`, you create a new grid type and add a corresponding grid function and callback function to your program.

As created by *GuiDesigner*, the grid function defines and handles the visual aspects of your new grid type and sends callback messages to your callback function. When your callback function receives callback messages it executes whatever code you put there for the occasion, which gives the grids their functionality and character.

Merged Grid Function

When you design a window to be a custom application window, it's more convenient to separate the visual and functional aspects into the grid function and callback function respectively.

But you don't have to. You can put the functional aspects directly in the grid function, and do away with the callback function completely. And if you want to create a new generic grid type you can distribute and/or add to the toolkit, you'll have to merge the functional part into the grid function eventually, or just put it there right from the start.

If you decide to add your code to the grid function, you'll need to understand grid functions in more detail than if you don't.

The rest of this section explain grid functions by discussing a standard grid function already in the toolkit, `XuiDialog2B`.

The code for `XuiDialog2B` is presented first, followed by a detailed line by line description. The numbers on the left of each line are added to facilitate discussion - they are not line numbers!

Grid Function Example

```
0 DECLARE FUNCTION XuiDialog2B (grid, message, v0, v1, v2, v3, r0, ANY)
1 '
2 '
3 ' #####
4 ' ##### XuiDialog2B () #####
5 ' #####
6 '
7 FUNCTION XuiDialog2B (grid, message, v0, v1, v2, v3, r0, (r1, r1$, r1[], r1$[]))
8     STATIC designX, designY, designWidth, designHeight
9     STATIC SUBADDR sub[]
10    STATIC upperMessage
11    STATIC XuiDialog2B
12 '
13    $XuiDialog2B = 0 ' kid 0
14    $Label = 1 ' kid 1
15    $TextLine = 2 ' kid 2
16    $Button0 = 3 ' kid 3
17    $Button1 = 4 ' kid 4
18    $UpperKid = 4 '
19 '
20    IFZ sub[] THEN GOSUB Initialize
21    IF XuiProcessMessage (grid, message, @v0, @v1, @v2, @v3, @r0, @r1, XuiDialog2B) THEN RETURN
22    IF (message <= upperMessage) THEN GOSUB @sub[message]
23    RETURN
24 '
25 '
26 ' ***** Callback ***** message = Callback : r1 = original message
27 '
28 SUB Callback
29     message = r1
30     callback = message
31     IF (message <= upperMessage) THEN GOSUB @sub[message]
32 END SUB
33 '
34 '
35 ' ***** Create ***** v0123 = xywh : r0 = window : r1 = parent
36 '
37 SUB Create
38     IF (v0 <= 0) THEN v0 = 0
39     IF (v1 <= 0) THEN v1 = 0
40     IF (v2 <= 0) THEN v2 = designWidth
41     IF (v3 <= 0) THEN v3 = designHeight
42     XuiCreateGrid (@grid, XuiDialog2B, @v0, @v1, @v2, @v3, r0, r1, &XuiDialog2B())
43     XuiLabel (@g, #Create, 0, 0, 0, 0, r0, grid)
44     XuiTextLine (@g, #Create, 0, 0, 0, 0, r0, grid)
45     XuiSendMessage ( g, #SetCallback, grid, &XuiDialog2B(), -1, -1, $TextLine, grid)
46     XuiPushButton (@g, #Create, 0, 0, 0, 0, r0, grid)
47     XuiSendMessage ( g, #SetCallback, grid, &XuiDialog2B(), -1, -1, $Button0, grid)
48     XuiSendMessage ( g, #SetTextString, 0, 0, 0, 0, 0, @"Enter")
49     XuiPushButton (@g, #Create, 0, 0, 0, 0, r0, grid)
50     XuiSendMessage ( g, #SetCallback, grid, &XuiDialog2B(), -1, -1, $Button1, grid)
51     XuiSendMessage ( g, #SetTextString, 0, 0, 0, 0, 0, @"Cancel")
52     GOSUB Resize
53 END SUB
54 '
55 '
56 ' ***** CreateWindow ***** r0 = windowType : r1$ = display$
57 '
58 SUB CreateWindow
59     IF (v0 = 0) THEN v0 = designX
60     IF (v1 = 0) THEN v1 = designY
61     IF (v2 <= 0) THEN v2 = designWidth
62     IF (v3 <= 0) THEN v3 = designHeight
63     XuiWindow (@window, #WindowCreate, v0, v1, v2, v3, r0, @r1$)
64     v0 = 0 : v1 = 0 : r0 = window : ATTACH r1$ TO display$
65     GOSUB Create
66     r1 = 0 : ATTACH display$ TO r1$
67     XuiWindow (window, #WindowRegister, grid, -1, v2, v3, @r0, @"XuiDialog2B")
68 END SUB
```

```

68 '
69 '
70 ' ***** GetSmallestSize *****
71 '
72 SUB GetSmallestSize
73   XuiSendMessage (grid, #GetBorder, 0, 0, 0, 0, $XuiDialog2B, @bw)
74   XuiSendMessage (grid, #GetSmallestSize, 0, 0, @labelWidth, @labelHeight, $Label, 8)
75   XuiSendMessage (grid, #GetSmallestSize, 0, 0, @textWidth, @textHeight, $TextLine, 8)
76 '
77   FOR i = $Button0 TO $Button1
78     XuiSendMessage (grid, #GetSmallestSize, 0, 0, @width, @height, i, 8)
79     IF (width > buttonWidth) THEN buttonWidth = width
80     IF (height > buttonHeight) THEN buttonHeight = height
81   NEXT i
82 '
83   width = buttonWidth + buttonWidth
84   IF (width < labelWidth) THEN width = labelWidth
85   v2 = width + bw + bw
86   v3 = labelHeight + buttonHeight + textHeight + bw + bw
87 END SUB
88 '
89 '
90 ' ***** Resize *****
91 '
92 SUB Resize
93   vv2 = v2
94   vv3 = v3
95   GOSUB GetSmallestSize      ' returns bw and heights
96   v2 = MAX (vv2, v2)
97   v3 = MAX (vv3, v3)
98 '
99   XuiPositionGrid (grid, @v0, @v1, @v2, @v3)
100 '
101   h = labelHeight + buttonHeight + textHeight + bw + bw
102   IF (v3 >= h + 4) THEN
103     buttonHeight = buttonHeight + 4 : h = h + 4
104     IF (v3 >= h + 4) THEN textHeight = textHeight + 4
105   END IF
106 '
107   labelWidth = v2 - bw - bw
108   labelHeight = v3 - buttonHeight - textHeight - bw - bw
109   buttonWidth = labelWidth >> 1
110   w0 = buttonWidth
111   w1 = labelWidth - w0
112 '
113   x = v0 + bw
114   y = v1 + bw
115   w = labelWidth
116   XuiSendMessage (grid, #Resize, x, y, w, labelHeight, $Label, 0)
117 '
118   y = y + labelHeight
119   XuiSendMessage (grid, #Resize, x, y, w, textHeight, $TextLine, 0)
120 '
121   h = buttonHeight
122   y = y + textHeight
123   XuiSendMessage (grid, #Resize, x, y, w0, h, $Button0, 0) : x = x + w0
124   XuiSendMessage (grid, #Resize, x, y, w1, h, $Button1, 0) : x = x + w1
125   XuiResizeWindowToGrid (grid, #ResizeWindowToGrid, v0, v1, v2, v3, 0, 0)
126 END SUB
127 '
128 '
129 ' ***** Selection *****
130 '
131 SUB Selection
132   SELECT CASE r0
133     CASE $XuiDialog2B :
134     CASE $TextLine :
135     CASE $Button0 :
136     CASE $Button1 :
137   END SELECT
138 END SUB

```

```

139 '
140 '
141 ' ***** Initialize *****
142 '
143 SUB Initialize
144   XuiGetDefaultGridFunctions (@func[])
145   XgrMessageNameToNumber (@"LastMessage", @upperSub)
146 '
147   func[#Callback]           = &XuiCallback ()
148   func[#GetSmallestSize]    = 0
149   func[#Resize]             = 0
150 '
151   DIM sub[upperSub]
152   sub[#Callback]            = SUBADDRESS (Callback)
153   sub[#Create]              = SUBADDRESS (Create)
154   sub[#CreateWindow]        = SUBADDRESS (CreateWindow)
155   sub[#GetSmallestSize]     = SUBADDRESS (GetSmallestSize)
156   sub[#Resize]              = SUBADDRESS (Resize)
157   sub[#Selection]           = SUBADDRESS (Selection)
158 '
159   IF sub[0] THEN PRINT "XuiDialog2B() : Initialize : error ::: (undefined message)"
160   IF func[0] THEN PRINT "XuiDialog2B() : Initialize : error ::: (undefined message)"
161   XuiRegisterGridType (@XuiDialog2B, @"XuiDialog2B", &XuiDialog2B(), @func[], @sub[])
162 '
163   designX = 0
164   designY = 0
165   designWidth = 160
166   designHeight = 68
167 '
168   gridType = XuiDialog2B
169   XuiSetGridTypeValue (gridType, @"xWin",           designX)
170   XuiSetGridTypeValue (gridType, @"yWin",           designY)
171   XuiSetGridTypeValue (gridType, @"width",          designWidth)
172   XuiSetGridTypeValue (gridType, @"height",         designHeight)
173   XuiSetGridTypeValue (gridType, @"minWidth",       designWidth)
174   XuiSetGridTypeValue (gridType, @"minHeight",      designHeight)
175   XuiSetGridTypeValue (gridType, @"maxWidth",       designWidth)
176   XuiSetGridTypeValue (gridType, @"maxHeight",      designHeight)
177   XuiSetGridTypeValue (gridType, @"border",        $$BorderFrame)
178   XuiSetGridTypeValue (gridType, @"can",            $$Focus OR $$Respond OR $$Callback OR $$InputTextString)
179   XuiSetGridTypeValue (gridType, @"focusKid",      $TextLine)
180   XuiSetGridTypeValue (gridType, @"inputTextString", $TextLine)
181   XuiSetGridTypeValue (gridType, @"redrawFlags",    $$RedrawBorder)
182   IFZ message THEN RETURN
183 END SUB
184 END FUNCTION

```

Function Declaration

```
DECLARE FUNCTION XuiDialog2B (grid, message, v0, v1, v2, v3, r0, ANY)
```

Line 0 is the function declaration line from the PROLOG.

It is typical of all grid function declarations, since all grid functions must take the same eight arguments. The last argument is declared as **ANY** type because, depending on **message**, it must contain:

- **XLONG** variable
- **STRING** variable
- **XLONG** array
- **STRING** array

It is very important to pass the correct type in the final argument. Failure to do so can crash your program and sometimes even the program development environment itself. It's easy to avoid such mistakes, however, since the argument is always **XLONG** unless the message explicitly calls for something else, as do the following:

```
STRING          - XuiGetTextString,  XuiGetHelpString
STRING array    - XuiGetTextArray,   XuiSetHelpStrings
XLONG array     - XuiGetKidsArray
```

Function Definition

```
FUNCTION XuiDialog2B (grid, message, v0, v1, v2, v3, r0, (r1, r1$, r1$[]))
```

Line 7 is the function definition line, the line that begins the function. It shows the same eight arguments, except the **ANY** argument expands to **r1,r1\$,r1\$[]** to carry the different type arguments.

These arguments are *aliased*, meaning *assigned the same address*. It's crucially important that your program refer *only* to the one appropriate argument - as determined by the **message** argument.

Variable Declarations

```
STATIC designX, designY, designWidth, designHeight
STATIC SUBADDR sub[]
STATIC upperMessage
STATIC XuiDialog2B
```

Lines 8 - 11 declare static variables. These variables are local to the grid function and they retain their values between subsequent calls.

designX, designY are the x,y position of the design window when it was converted into a grid function by **WindowToFunction**, in display coordinates, with 0,0 in upper-left corner of the display.

designWidth, designHeight are the width and height of the design window in pixels when it was converted into a grid function.

sub[] is an array that contains a subroutine address for every message number. In typical grid functions, most locations contain a zero, so no subroutine is executed for the corresponding message.

upperMessage holds the highest defined message number.

XuiDialog2B contains the grid type number.

Kid Constant Definitions

```
$XuiDialog2B = 0 ' kid 0
$Label       = 1 ' kid 1
$TextLine    = 2 ' kid 2
$Button0     = 3 ' kid 3
$Button1     = 4 ' kid 4
```

Lines 13 - 17 define kid constants to name the grid type components.

As always, the grid itself is always kid 0. The **xuiLabel** grid at the top of **xuiDialog2B** is kid 1, the **xuiTextLine** grid below it is kid 2, and the left and right hand **xuiPushButton** grids are kids 3 & 4.

WindowToFunction numbers kids in the order they are created in the **Create** subroutine. This correspondence is absolutely necessary and must be maintained if you edit the **Create** subroutine.

Initialize - Process Message - RETURN

```
IFZ sub[] THEN GOSUB Initialize
IF XuiProcessMessage (grid, message, @v0, @v1, @v2, @v3, @r0, @r1, XuiDialog2B) THEN RETURN
IF (message <= upperMessage) THEN GOSUB @sub[message]
RETURN
```

Lines 20-23 initialize the grid type and basic grid function variables, invoke the appropriate routines to process the message, and return.

Initialize

```
IFZ sub[] THEN GOSUB Initialize
```

The first time a grid function is entered, it must initialize certain variables and register its grid type. `sub[]` is a local variable and is therefore empty the first time the function is entered. This line calls the `Initialize` subroutine the first time the function is entered, but never again since `sub[]` is dimensioned in the subroutine.

Process Message with Message Processing Function

```
IF XuiProcessMessage (grid, message, @v0, @v1, @v2, @v3, @r0, @r1, XuiDialog2B) THEN RETURN
```

In typical grid functions, most messages are ignored or processed by a *message processing function* called by `XuiProcessMessage()`. The message processing functions for the grid function are established in the `Initialize` subroutine.

If the `r0` argument is not zero, the message is for a kid of the grid, and `XuiProcessMessage()` sends the message on to the kid. The return value of `XuiProcessMessage()` is the kid number the message was sent to, so this line returns from the grid function if the message was sent to a kid of the grid, not the grid itself, thus preventing the grid function from receiving a message intended for a kid and not itself.

Process Message with Message Processing Subroutine

```
GOSUB @sub[message]
```

If a *message processing subroutine* is defined for a particular message, `GOSUB @sub[message]` line calls the subroutine.

Done

```
RETURN
```

Return to the routine that called the grid function.

Callback Subroutine

```
'  
'  
' ***** Callback ***** message = Callback : r1 = original message  
'  
SUB Callback  
  message = r1  
  callback = message  
  IF (message <= upperMessage) THEN GOSUB @sub[message]  
END SUB
```

The `callback` subroutines in grid functions generated by *GuiDesigner* are not executed. That's because *GuiDesigner* puts the following statements near the bottom of the grid function:

```
func[#Callback]          = &XuiCallback ()  
' sub[#Callback]        = SUBADDRESS (Callback)
```

The first statement makes the grid function pass `callback` messages back to the function that set itself as its callback function, while the second statement is commented to disable the `callback` subroutine.

Thus *GuiDesigner* generated grid functions create visual only grids. Any functionality or special purpose is provided by the callback function set for each grid. As a result, every grid can have unique functionality, since each can have its own callback function.

To create grids with special purpose functionality, a grid function must process the callback messages it receives from its kid grids. Comment out the `func[#Callback]` line and enable `sub[#Callback]`. The `callback` subroutine will now execute when the grid function receives `callback` messages.

The lines in the `callback` subroutine transfer the original message from `r1` to `message`, set variable `callback = message`, then call the subroutine that processes the original message. The transfer from `r1` to `message` is necessary because `XuiCallback()` puts the original message in `r1` and a `callback` message in `message`.

Almost all callback messages carry a `selection` message in `r1`. So in most cases, all you have to do to add special purpose functionality to a grid function is add a `SELECT CASE` block to the `selection` subroutine, with one `CASE` entry for each kid constant. In fact, if you've already added the special purpose functionality in a callback function, you can simply copy its `selection` subroutine into the grid function, plus any other code that supports it.

Whenever a function needs to know whether it received a particular message in a callback or not, it can test its local `callback` variable. `callback = 0` means the message is not a callback message, otherwise `callback` contains the callback message number.

Create Subroutine

```
,
,
' ***** Create ***** v0123 = xywh : r0 = window : r1 = parent
,
SUB Create
  IF (v0 <= 0) THEN v0 = 0
  IF (v1 <= 0) THEN v1 = 0
  IF (v2 <= 0) THEN v2 = designWidth
  IF (v3 <= 0) THEN v3 = designHeight
  XuiCreateGrid (@grid, XuiDialog2B, @v0, @v1, @v2, @v3, r0, r1, &XuiDialog2B())
  XuiLabel      (@g, #Create, 0, 0, 0, 0, r0, grid)
  XuiTextLine   (@g, #Create, 0, 0, 0, 0, r0, grid)
  XuiSendMessage ( g, #SetCallback, grid, &XuiDialog2B(), -1, -1, $TextLine, grid)
  XuiPushButton (@g, #Create, 0, 0, 0, 0, r0, grid)
  XuiSendMessage ( g, #SetCallback, grid, &XuiDialog2B(), -1, -1, $Button0, grid)
  XuiSendMessage ( g, #SetTextString, 0, 0, 0, 0, 0, @"Enter")
  XuiPushButton (@g, #Create, 0, 0, 0, 0, r0, grid)
  XuiSendMessage ( g, #SetCallback, grid, &XuiDialog2B(), -1, -1, $Button1, grid)
  XuiSendMessage ( g, #SetTextString, 0, 0, 0, 0, 0, @"Cancel")
  GOSUB Resize
END SUB
```

Every grid function has a **Create** subroutine to process **Create** messages. Each time a grid function receives a **Create** message, it must create a new grid of its own grid type. It does this by calling **XuiCreateGrid()** to create the base grid. Then, if the grid type has kid grids, it calls the appropriate grid functions to create them.

After it creates each kid grid, the grid function sets itself as the callback function for the kid grid - assuming it wants to receive callback messages from the kid. Configuration messages like **SetBorder**, **SetColor**, **SetTextString** must be sent to each kid grid before the next kid grid is created as are the **PushButton** "Enter" and "Cancel" strings in the example above.

After it creates and configures all its kids, the **Create** subroutine must call its **Resize** subroutine, even if its **Resize** subroutine is empty. Any additional code required in the **Create** subroutine, should be carried out in separate subroutines and called immediately before and/or after **GOSUB Resize**.

CreateWindow Subroutine

```
,
,
' ***** CreateWindow ***** r0 = windowType : r1 = &WindowFunc()
,
SUB CreateWindow
  IF (v0 = 0) THEN v0 = designX
  IF (v1 = 0) THEN v1 = designY
  IF (v2 <= 0) THEN v2 = designWidth
  IF (v3 <= 0) THEN v3 = designHeight
  XuiWindow (@window, #WindowCreate, v0, v1, v2, v3, r0, @r1$)
  v0 = 0 : v1 = 0 : r0 = window : ATTACH r1$ TO display$
  GOSUB Create
  r1 = 0 : ATTACH display$ TO r1$
  XuiWindow (window, #WindowRegister, grid, -1, v2, v3, @r0, @"XuiDialog2B")
END SUB
```

Virtually every grid function has a **CreateWindow** subroutine to process **CreateWindow** messages. Each time a grid function receives a **CreateWindow** message, it creates a new grid of its own grid type surrounded by a new window. Since grids must be created in a window, it first creates the window, then calls the **Create** subroutine to create the grid. There are few ways to modify the **CreateWindow** subroutine successfully, so it is rarely if ever attempted.

GetSmallestSize *Subroutine*

```
,
,
, ***** GetSmallestSize *****
,
SUB GetSmallestSize
  XuiSendMessage (grid, #GetBorder, 0, 0, 0, 0, XuiDialog2B, @bw)
  XuiSendMessage (grid, #GetSmallestSize, 0, 0, @labelWidth, @labelHeight, $Label, 8)
  XuiSendMessage (grid, #GetSmallestSize, 0, 0, @textWidth, @textHeight, $TextLine, 8)
,
  FOR i = $Button0 TO $Button1
    XuiSendMessage (grid, #GetSmallestSize, 0, 0, @width, @height, i, 8)
    IF (width > buttonWidth) THEN buttonWidth = width
    IF (height > buttonHeight) THEN buttonHeight = height
  NEXT i
,
  width = buttonWidth + buttonWidth
  IF (width < labelWidth) THEN width = labelWidth
  v2 = width + bw + bw
  v3 = labelHeight + textHeight + buttonHeight + bw + bw
END SUB
```

A `GetSmallestSize` subroutine is needed in any resizable grid. Grid functions generated by *GuiDesigner* are not resizable - they are initially set up to return the fixed design size when they receive a `GetSmallestSize` message.

The `GetSmallestSize` and `Resize` subroutines in this function are a good model, since most grid functions employ the similar methods.

`GetSmallestSize` first sends a `GetBorder` message to itself get its own border width. Next it sends `GetSmallestSize` messages to its kids - the `XuiLabel` at the top, the `XuiTextLine` below it, and the two `XuiPushButtons` at the bottom. It sets the smaller of the buttons equal the larger, sets the overall width `v2` to the largest width plus twice the border width, and sets the overall height `v3` to the sum of the label height, text height, button height, and twice the border width.

Resize Subroutine

```
'
'
' ***** Resize *****
'
SUB Resize
  vv2 = v2
  vv3 = v3
  GOSUB GetSmallestSize      ' returns bw and heights
  v2 = MAX (vv2, v2)
  v3 = MAX (vv3, v3)
'
  XuiPositionGrid (grid, @v0, @v1, @v2, @v3)
'
  h = labelHeight + buttonHeight + textHeight + bw + bw
  IF (v3 >= h + 4) THEN
    buttonHeight = buttonHeight + 4 : h = h + 4
    IF (v3 >= h + 4) THEN textHeight = textHeight + 4
  END IF
'
  labelWidth = v2 - bw - bw
  labelHeight = v3 - buttonHeight - textHeight - bw - bw
  buttonWidth = labelWidth >> 1
  w0 = buttonWidth
  w1 = labelWidth - w0
'
  x = v0 + bw
  y = v1 + bw
  w = labelWidth
  XuiSendMessage (grid, #Resize, x, y, w, labelHeight, $Label, 0)
'
  y = y + labelHeight
  XuiSendMessage (grid, #Resize, x, y, w, textHeight, $TextLine, 0)
'
  h = buttonHeight
  y = y + textHeight
  XuiSendMessage (grid, #Resize, x, y, w0, h, $Button0, 0) : x = x + w0
  XuiSendMessage (grid, #Resize, x, y, w1, h, $Button1, 0) : x = x + w1
  XuiResizeWindowToGrid (grid, #ResizeWindowToGrid, v0, v1, v2, v3, 0, 0)
END SUB
```

The **Resize** subroutine is supposed to change the **x,y,width,height** position and size of the **XuiDialog2B** grid to **v0,v1,v2,v3**.

Resize first it calls its own **GetSmallestSize** subroutine to see if the requested width,height is smaller than the grid can resize. **XuiPositionGrid()** sets the position and size of the **XuiDialog2B** grid in window coordinates.

Taking advantage of several variables that **GetSmallestSize** already computed, the **Resize** subroutine computes the width and height of all the kid grids, and sends a **Resize** message to each kid to set its position and size.

Finally, **Resize** calls **ResizeWindowToGrid()** function, which resizes the window to the **XuiDialog2B** grid if the **XuiDialog2B** grid was created by a **CreateWindow** message. In other words, **ResizeWindowToGrid()** resizes the window only if the grid is supposed to fill the grid.

Selection Subroutine

```
'  
'  
' ***** Selection *****  
'  
SUB Selection  
  PRINT "Selection callback from kid #"; r0  
END SUB
```

`XuiDialog2B` doesn't process callback messages internally because its `Callback` subroutine sends callback messages back to the function that set itself as the `XuiDialog2B` grid.

If you modify the `Callback` subroutine as described previously, however, `Selection` callback messages will be sent to this subroutine. Here you can add functionality code to process callback messages from the `XuiDialog2B` kid grids, and send `Selection` and/or other callback messages back under appropriate circumstances.

As a simple example, if you made the described modifications to the `Callback` subroutine, the following code will convert this function into a simplified name input dialog.

First, add

```
  XuiSendMessage (g, #SetTextString, 0, 0, 0, 0, 0, @"Enter your name below")
```

to the `Create` subroutine after the line that creates the `XuiLabel1` kid to tell the user what to do when the grid appears.

Second, fill the `Selection` subroutine with these lines:

```
SUB Selection  
  SELECT CASE r0  
    CASE $TextLine, $Button0  
      XuiSendMessage (grid, #GetTextString, 0, 0, 0, 0, $TextLine, @name$)  
      XuiSendMessage (grid, #SetTextString, 0, 0, 0, 0, 0, @name$)  
      IF name$ THEN value = 0 ELSE value = -1  
    CASE $Button1  
      value = -1      ' 0 = ok : -1 = cancel  
  END SELECT  
  XuiCallback (grid, #Selection, value, 0, 0, 0, 0, 0)  
END SUB
```

Initialize Subroutine

```
'
'
' ***** Initialize *****
'
SUB Initialize
  XuiGetDefaultGridFunctions (@func[])
  XgrMessageNameToNumber (@"LastMessage", @upperSub)
'
  func[#GetSmallestSize] = 0 ' cancel default standard message function
  func[#Resize]          = 0 ' cancel default standard message function
'
  DIM sub[upperSub]
  sub[#Callback]         = SUBADDRESS (Callback)
  sub[#Create]           = SUBADDRESS (Create)
  sub[#CreateWindow]     = SUBADDRESS (CreateWindow)
  sub[#GetSmallestSize] = SUBADDRESS (GetSmallestSize)
  sub[#Resize]           = SUBADDRESS (Resize)
  sub[#Selection]        = SUBADDRESS (Selection)
'
  IF func[0] THEN PRINT "XuiDialog2B() : Initialize : error ::: (undefined message)"
  IF sub[0] THEN PRINT "XuiDialog2B() : Initialize : error ::: (undefined message)"
  XuiRegisterGridType (@XuiDialog2B, @"XuiDialog2B", &XuiDialog2B(), @func[], @sub[])
'
  designX = 0
  designY = 0
  designWidth = 160
  designHeight = 68
'
  gridType = XuiDialog2B
  XuiSetGridTypeValue (gridType, @"xWin",          designX)
  XuiSetGridTypeValue (gridType, @"yWin",          designY)
  XuiSetGridTypeValue (gridType, @"width",         designWidth)
  XuiSetGridTypeValue (gridType, @"height",        designHeight)
  XuiSetGridTypeValue (gridType, @"minWidth",      64)
  XuiSetGridTypeValue (gridType, @"minHeight",     24)
' XuiSetGridTypeValue (gridType, @"minWidth",      designWidth)
' XuiSetGridTypeValue (gridType, @"minHeight",     designHeight)
' XuiSetGridTypeValue (gridType, @"maxWidth",      designWidth)
' XuiSetGridTypeValue (gridType, @"maxHeight",     designHeight)
  XuiSetGridTypeValue (gridType, @"border",        $$BorderFrame)
  XuiSetGridTypeValue (gridType, @"can",           $$Focus OR $$Respond ...)
  XuiSetGridTypeValue (gridType, @"focusKid",     $TextLine)
  XuiSetGridTypeValue (gridType, @"inputTextString", $TextLine)
  XuiSetGridTypeValue (gridType, @"redrawFlags",   $$RedrawBorder)
  IFZ message THEN RETURN
END SUB
END FUNCTION
```

The `Initialize` subroutine is executed only once - the first time the grid function is called. It establishes the initial or default appearance and behavior of every grid of a grid type by setting its:

- *default message processing functions*
- *default message processing subroutines*
- *default properties*

When a grid is created, it is given the default values of its grid type. After a grid is created, programs can send any number of messages to a grid to change its properties, message processing functions, and message processing subroutines.

Programs often send a few messages to a grid to change properties like grid name, help string, or color. Most programs should rarely if ever change a message processing function or subroutine, however.

Get Default Message Functions

```
,  
,  
, ***** Initialize *****  
,  
SUB Initialize  
  XuiGetDefaultMessageFunctions (@func[])  
  XgrMessageNameToNumber (@"LastMessage", @upperMessage)
```

The `Initialize` subroutine first gets a copy of the default message processing functions in `func[]`, and finds the highest message number, which it will soon need.

The default message processing functions give a grid basic, straightforward functionality by routing common messages to standard message processing functions that handle the messages in an obvious, direct, expected way. For example, `XuiSetColor()` is the standard message processing function for message `SetColor`, and it sets any combination of background, drawing, lowlight, highlight colors for the grid (a `-1` argument means leave unchanged).

Establish Message Functions

```
func[#Callback]      = &XuiCallback()  
func[#GetSmallestSize] = 0  
func[#Resize]        = 0
```

Three message processing functions are changed from the default values established by `XuiGetDefaultMessageFunctions()`:

`func[#Callback] = &XuiCallback` sets the message processing function for `Callback` messages to `&XuiCallback()`. This is normal practice for "visual only grids" - those without internal functionality. Whenever this function receives a `Callback` message, `XuiCallback()` sends the message back to whatever function set itself as the callback function for the `XuiDialog2B` grid whose kid produced the `Callback` message. When *GuiDesigner* creates grid functions, it inserts this line of code. To give a visual grid type functionality, remove this line, enable the `sub[#Callback] = SUBADDRESS (Callback)` line a few lines below, and add code to the `Selection` and/or any other relevant callback message processing subroutines (most callback messages are `Selection` messages).

`func[#GetSmallestSize]=0` cancels default message processing function `XuiGetSmallestSize()` for message `GetSmallestSize`. `XuiGetSmallestSize()` returns the original design size because `minWidth,maxWidth,minHeight,maxHeight` properties are set to the `designWidth,designHeight`. This fixes the size of the grid, while `XuiDialog2B` is supposed to be resizable. An internal `GetSmallestSize` subroutine replaces the message function.

`func[#Resize] = 0` cancels the default message processing function `XuiResizeNot()` for message `Resize`. `XuiResizeNot()` does not resize grids and is therefore designed for fixed-size "non-resizable" grids, while `XuiDialog2B` is resizable. An internal `Resize` subroutine replaces the message function.

Establish Message Subroutines and Register Grid Type

```
DIM sub[upperMessage]
' sub[#Callback]           = SUBADDRESS (Callback)
  sub[#Create]            = SUBADDRESS (Create)
  sub[#CreateWindow]      = SUBADDRESS (CreateWindow)
  sub[#GetSmallestSize]   = SUBADDRESS (GetSmallestSize)
  sub[#Resize]            = SUBADDRESS (Resize)
  sub[#Selection]         = SUBADDRESS (Selection)
'
IF sub[0] THEN PRINT "XuiDialog2B(): Initialize: Error::: (Undefined Message)"
IF func[0] THEN PRINT "XuiDialog2B(): Initialize: Error::: (Undefined Message)"
XuiRegisterGridType (@XuiDialog2B, @"XuiDialog2B", &XuiDialog2B(), @func[], @sub[])
```

The `sub[]` array, which must contain the message processing subroutine addresses for this grid function, is dimensioned large enough to hold subroutine addresses for the highest message number. Then subroutine addresses are then assigned to `sub[]` for those messages to be processed by internal message processing subroutines.

Every grid function *must* have message processing subroutines for `Create` and `CreateWindow` messages, so whenever *GuiDesigner* creates a grid function, it generates message subroutines and corresponding message subroutine address assignments for `Create` and `CreateWindow`.

```
sub[#Create]              = SUBADDRESS (Create)
sub[#CreateWindow]       = SUBADDRESS (CreateWindow)
```

GuiDesigner also creates subroutines and address assignments for `Callback` and `Selection` messages. But the `Callback` address assignment is disabled, since `func[#Callback]=&XuiCallback()` handles callback messages for visual grid types.

To add functionality to a visual only grid type, disable the `func[#Callback]=&XuiCallback()` line, then enable the `sub[#Callback]=SUBADDRESS(Callback)` line and add functionality code to the `Selection` subroutine and/or any other callback relevant callback message processing subroutines.

To make `XuiDialog2B` grids resizable, a programmer provided message subroutines for `GetSmallestSize` and `Resize` messages. To make these subroutines execute when this grid function receives these messages, the following lines were added to `Initialize` :

```
func[#GetSmallestSize]   = 0                ' cancel standard function
func[#Resize]            = 0                ' cancel standard function

sub[#GetSmallestSize]    = SUBADDRESS (GetSmallestSize) ' enable internal subroutine
sub[#Resize]             = SUBADDRESS (Resize)         ' enable internal subroutine
```

The first two lines cancel any default standard message functions that might exist, and the second two lines install the programmer written message subroutines.

Finally, the `XuiDialog2B` grid type is registered with *GraphicsDesigner* and *GuiDesigner* by `XuiRegisterGridType()`.

Establish Grid Type Properties

```
designX = 4
designY = 23
designWidth = 160
designHeight = 68
,
gridType = XuiDialog2B
XuiSetGridTypeValue (gridType, @"xWin", designX)
XuiSetGridTypeValue (gridType, @"yWin", designY)
XuiSetGridTypeValue (gridType, @"width", designWidth)
XuiSetGridTypeValue (gridType, @"height", designHeight)
' XuiSetGridTypeValue (gridType, @"minWidth", designWidth)
' XuiSetGridTypeValue (gridType, @"minHeight", designHeight)
' XuiSetGridTypeValue (gridType, @"maxWidth", designWidth)
' XuiSetGridTypeValue (gridType, @"maxHeight", designHeight)
XuiSetGridTypeValue (gridType, @"border", $$BorderFrame)
XuiSetGridTypeValue (gridType, @"can", $$Focus OR $$Respond OR $$Callback OR $
$InputTextString)
XuiSetGridTypeValue (gridType, @"focusKid", $TextLine)
XuiSetGridTypeValue (gridType, @"inputTextString", $TextLine)
XuiSetGridTypeValue (gridType, @"redrawFlags", $$RedrawBorder)
IFZ message THEN RETURN
END SUB
END FUNCTION
```

The `Initialize` subroutine also establishes basic grid type properties. Whenever a grid of this grid type is created, it is given the properties assigned in this section.

`designX`, `designY`, `designWidth`, `designHeight` are the position and size of the design window *GuiDesigner* created the grid function from. Windows created sending a `CreateWindow` message to this grid with a `-1` in `v0,v1,v2,v3` will position and size the window as designed. Otherwise, it is placed and sized to the values in `v0,v1,v2,v3`.

When `WindowToFunction` translates a design window into a grid function, it sets `minWidth,minHeight,maxWidth,maxHeight` to `designWidth,designHeight`. This makes all grids of this grid type fixed-size and non-resizable. These four lines were disabled in `XuiDialog2B` when `GetSmallestSize` and `Resize` subroutines were added to make the grid resizable.

- align, 6
- appearance, 18
- AppearanceWindow, 5, 6, 18, 22, 24
- argument, 28, 33, 35, 38, 39, 47
- arguments, 11

- BehaviorWindow, 5, 18
- border, 6, 18

- Callback, 19, 50, 54, 56, 57
- callback, 11
- callback function, 11, 16, 17, 19, 20, 40, 43, 50
- callback message, 11, 16, 19, 40, 54, 57
- CEO function, 41
- code function, 16
- color, 6, 18
- convenience functions, 13
- coordinate system, 9
- Create, 15, 20, 22, 48, 51, 54, 57
- CreateWindow, 15, 17, 21, 35, 51, 53, 57, 58
- CreateWindows(), 13, 15, 16, 17

- design mode, 19
- design window, 10, 16, 17, 18, 19, 20, 21, 22, 24, 48, 58
- designHeight, 20, 48, 56, 58
- DesignMode, 3
- designWidth, 20, 48, 56, 58
- designX, 20, 48, 58
- designY, 20, 48, 58
- Destroy, 17
- display, 9
- DisplayWindow, 15, 16

- entry function, 14, 34, 42
- Entry(), 13, 14, 34

- frame, 9
- func[], 56

- GetBorder, 52
- GetSmallestSize, 52, 53, 56, 57, 58
- Graphical User Interface, 1
- graphics, 12
- graphics programs, 12
- GraphicsDesigner, 9, 11, 12, 27, 28, 32, 33, 34, 41
- grid, 9, 12, 21, 28, 38, 39
- grid function, 10, 11, 16, 17, 19, 20, 21, 22, 27, 35, 36, 37, 38, 39, 40, 41, 43, 47, 48, 49, 51, 57, 58
- grid message, 27, 28, 35, 36, 37, 38
- grid name, 6
- grid number, 12, 21, 39
- grid property, 58
- grid type, 10, 18, 37, 43, 49, 58
- GridAppearance, 5, 18
- GridBehavior, 5, 18
- GridDelete, 5
- GridName, 25
- GridSortGrids, 5
- GUI, 1
- GUI components, 9
- GUI programs, 12

- help string, 6
- HelpFile, 24
- HelpString, 24

- indent, 6
- InitGui(), 13, 15, 16, 28
- Initialize subroutine, 49, 55, 56, 58
- InitProgram(), 13, 15, 16
- InstantHelp, 2, 23, 24

- justification, 6

- keyboard, 27, 30
- keyboard focus, 30, 36
- keyboard message, 41
- KeyDown, 27, 30, 36
- KeyUp, 30, 36
- kid, 9, 12, 19, 21, 28, 39, 48
- kid constants, 48
- kid number, 19, 20
- kid number constant, 19
- kid numbers, 9

layout, 18
 maxHeight, 56, 58
 maximize, 9
 maxWidth, 56, 58
 message, 11, 12, 21, 22, 27, 28, 33, 38, 39, 47
 message loop, 15, 34, 42
 message number, 12, 16, 28, 39
 message processing function, 18, 37, 49, 55, 56
 message processing subroutine, 18, 37, 49, 55, 57
 message queue, 33, 42
 minHeight, 56, 58
 minimize, 9
 minWidth, 56, 58
 MonitorContext, 41
 MonitorKeyboard, 41
 MonitorMouse, 41
 mouse, 27, 32
 mouse button, 27
 mouse cursor, 32
 mouse focus, 32
 mouse message, 32
 MouseDown, 11, 27
 MouseMove, 27
 MouseUp, 11
 parent, 9
 process message, 33, 34, 37, 40, 49
 PROLOG, 13
 r0, 12, 19, 28
 r1, 19
 Resize, 20, 51, 52, 53, 56, 57, 58
 ResizeWindowToGrid(), 53
 selected grid, 18
 Selection, 19, 40, 54, 56, 57
 send message, 11, 12, 38
 SetBorder, 17, 51
 SetCallback, 16, 21, 40
 SetColor, 17, 51, 56
 SetTextString, 17, 39, 51
 sub[], 48, 49, 57
 terminateProgram, 15
 test mode, 19
 TestMode, 3
 text array, 6
 text string, 6
 TimeOut, 27
 title-bar, 9
 toolkit, 3, 17, 22
 upperMessage, 48
 window, 9, 12, 28, 35, 38, 51
 window function, 9, 27, 30, 35, 36, 38, 39, 41
 window message, 27, 35, 36, 38
 window number, 12
 window type, 9
 WindowDelete, 4
 WindowFromFunction, 4, 20
 WindowHide, 4
 WindowKeyDown, 27, 30, 36
 WindowKeyUp, 30, 36
 WindowLoad, 4
 WindowMouseDown, 27
 WindowMouseMove, 27
 WindowNew, 4
 WindowSave, 4
 WindowSelected, 27
 WindowToFunction, 4, 10, 19, 22, 43, 48
 wingrid, 28
 XgrProcessMessages(), 15, 29, 32, 33, 34, 35, 41, 42
 Xui(), 15
 XuiCallback(), 50, 56
 XuiCreateGrid(), 51
 XuiGetDefaultMessageFunctions(), 56
 XuiGetSmallestSize(), 56
 XuiPositionGrid(), 53
 XuiProcessMessage(), 49
 XuiResizeNot(), 56
 XuiSendMessage(), 12, 15, 38
 XuiSetColor(), 56
 XuiWindow(), 35