UIM/X

UIM/X Motif Developer’s Guide

ICS

Integrated Computer
Solutions Incorporated

Integrated Computer Solutions, Inc.

54 Middlesex Turnpike, Bedford, MA 01730

Tel: 617.621.0060

Fax: 617.621.9555

E-mail: info@ics.com

WWW: http://www.ics.com

UIM/X Trademarks

Contents

Preface

Overview

This guide helps you to use UIM/X to build Motif interfaces. It explains why the Motif standard is good for programmers to follow, and illustrates the use of the Motif objects that come packaged with UIM/X. It also discusses several programming techniques that you should use when designing Motif applications.

Who Should Use this Guide

This manual assumes you are familiar with the basics of UIM/X. Before using this manual, review the *UIM/X Beginner's Guide* and the *UIM/X User's Guide.*

This manual also assumes that you have some knowledge of programming and a general understanding of the X Window System. You should also know how to use common items such as menus, buttons, and scroll bars. If you are not familiar with these items, you may find it useful to review the *OSF/Motif User's Guide*.

Before you begin, check with your system administrator to ensure that the software has been installed as described in the *UIM/X Installation Guide*.

Before You Read this Guide

This guide makes the following assumptions:

- You are familiar with the basic functions of selecting from menus and dialog boxes; opening, moving, resizing and closing windows, and clicking icons.
- You understand the functions of the three mouse buttons, which this guide refers to as the Select button (left button), the Adjust button (middle), and the Menu button (right). See "Using the Mouse" on page x for more information.
- Either you have enough familiarity with programming to enter your own callback code; or you are using the novice mode of UIM/X to help design user interfaces for which a colleague can provide any code required.

The UIM/X Document Set and Related Books

This section lists the UIM/X document set, and provides a suggested list for further reading.

The following list is the complete UIM/X document set:

- *UIM/X Installation Guide*. Explains how to install and run UIM/X. Includes information on the files provided with UIM/X, backwards compatibility issues, and compiler considerations.
- *UIM/X Beginner's Guide*. Introduces UIM/X by presenting Novice Mode, the simplified Palette that enables new users to be productive immediately. Includes information on a number of important features for creating, testing, and running applications.
- *UIM/X Tutorial Guide***.** A series of step-by-step tutorials, teaching tools and techniques that will greatly assist you in developing your own applications. Features tutorials in Novice Mode, Standard Mode, and on advanced topics.
- *UIM/X User's Guide*. Explores the UIM/X features common to both Motif and cross-platform development. Includes discussions of how to use UIM/ X's editors to set properties, add behavior, etc.
- *UIM/X Motif Developer's Guide*. An in-depth guide to the widgets, features and capabilities of UIM/X as they relate specifically to Motif development.
- *UIM/X Advanced Topics***.** Describes how to customize UIM/X, including integrating new widget and component classes into the executable. Includes reference information of an advanced technical nature.
- *UIM/X Reference Manual*. A comprehensive list of properties, methods, and events, plus more, for Motif development. Designed for the experienced developer.

Suggested Reading

For more information on designing GUIs, see any of the following books:

- *OSF/Motif Style Guide release 1.2* (Prentice Hall, 1993, ISBN 0-13-643123-2)
- *Visual Design with OSF/Motif* (by Shiz Kobara, Addison-Wesley, 1991, ISBN 0-201-56320-7)
- *New Windows Interface: An Application Guide* (Microsoft Corporation, 1994, ISBN 1-55615-679-0)
- *Human Interface Guidelines: The Apple Desktop Interface* (Addison-Wesley, 1987, ISBN 0-201-17753-6)

How this Guide Is Organized

- *Chapter 1, "Overview of Motif Widgets,"* describes each of the objects supported by UIM/X.
- *Chapter 2, "Programming in UIM/X,"* describes how to use the Ux Convenience Library to specify behavior in your interfaces.
- *Chapter 3, "Using the Motif Components,"* describes some of the advanced Motif objects, and discusses some of the issues of Motif programming.
- *Chapter 4, "Controlling Appearance,"* describes how to control the layout and appearance of your interfaces.
- *Chapter 5, "Specifying Application Window Behavior,"* explains how to customize the look and feel of you project by specifying application window behavior.
- *Chapter 6, "Adding Interfaces to Existing Applications,"* explains how to create GUIs for existing programs.
- *Chapter 7, "Generating and Compiling Project Code,"* discusses some of the issues involved in generating code for your projects.
- *Appendix A, "Object Property Values,"* outlines the range of values possible for each property.
- *Appendix B, "Frequently Asked Questions,"* provides answers to some of the most commonly encountered questions about UIM/X.

Some Terms You Should Know

Certain basic terms recur throughout this guide, and it helps to understand them from the outset.

An *object* is a building block you can use to build an interface with UIM/X.

A *Motif widget* is an object whose appearance and behavior precisely follows the *OSF/Motif Style Guide*. The novice mode of UIM/X supports a number of popular Motif widgets, including Push Button, Label, Text Field, and more.

A compound object consists of several Motif widgets combined into one object for your convenience. The novice mode of UIM/X supports a number of compound objects, including Application Window and Group Box, that save you the time you might otherwise spend creating them.

An *interface* is a window or dialog box that you build up from objects with UIM/X. The novice mode of UIM/X supports four different types of interfaces: Application Window, Secondary Window, Message dialog box, and File Selection dialog box. Certain menu options refer to an interface, such as Save Interface; these act only on your selected interface.

A *project* contains all the interfaces (i.e., windows and dialog boxes) and their associated files for a certain GUI you are building with UIM/X. The program can automatically save and generate code for an entire project in one step. Certain menu options refer to a project, such as Save Project; these act on all the windows and dialog boxes in your project.

Conventions Used in this Guide

Typographic Conventions

The following table describes the typographic conventions used in this guide.

Typeface or Symbol	**Meaning**	**Example**
`AaBbCc12`	The names of commands, files, and directories; or on screen output; or user input.	Edit your `.login` file. `%You have mail.` Use `ls -a` to list all the files.
AaBbCc12	A placeholder you replace with your actual value; or words to be emphasized; or book titles.	To delete a file, type `rm` *filename.* You *must* be `root` to do this. See Chapter 6 in the *User's Guide*.
File⇒Open	The Open option in the File menu.	Choose the File⇒Open command.
Alt+F4	Press both Alt and F4 at once.	Press Alt+F4 to exit.
Return	The key on your keyboard marked Enter, Return, or .	Press Return.

Installation Directories

Product installation directories can depend on the platform or the user's preferences. To keep things simple, this guide uses general names for product installation directories. The following table lists the name and the corresponding product installation directory:

Name	Description
uimx_directory	The UIM/X installation directory

Using the Mouse

Before starting the tutorial, take a moment to review the location and usage of your mouse buttons, as illustrated in the Figure P-1and the following table:

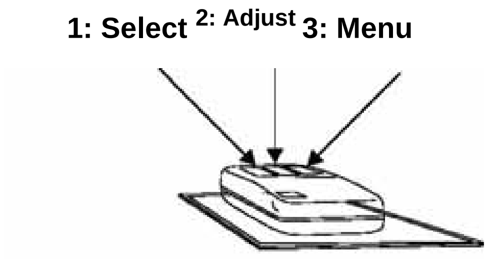


Figure P-1 The Mouse Buttons

Button	Called	Is Used For
1	Select	Selecting objects, menus, toggles, and options.
2	Adjust	Resizing and moving objects.
3	Menu	Displaying popup menus.

Throughout this book, you will use the mouse buttons along with the mouse pointer to make selections, move the input pointer, or position the text insertion point. You can perform any of the following mouse operations.

Operation	Description
Point to	Move the mouse to make the pointer go as directed.
Press	Hold down a mouse button.

Operation	Description
Release	Release a mouse button after pressing it.
Click	Quickly press and release a mouse button without moving the mouse.
Drag	Move the mouse while pressing a mouse button.
Double-click	Click a mouse button twice in rapid succession without moving the mouse pointer.
Triple-click	Click a mouse button three times in rapid succession without moving the mouse pointer

In general, instructions for mouse operations include the name of the mouse button. The exceptions are Click, Double-click, and Drag. These common operations may be described without specifying a mouse button. For example:

- Click on the `applWindow1` icon in the Interfaces Area of the Project Window.
- Drag the Push Button icon from the Palette.

In these cases, use the Select button to click and double-click, and the Adjust button to drag.

Setting Application Defaults

Application Defaults configure the way UIM/X looks and set the default preferences for many of its operations. You can set the Application Defaults for all UIM/X users or for a single user. For more details on setting your Application Defaults see the *UIM/X User's Guide*.

For optimum performance, set the following resources in your Application Defaults:

```
Mwm*autoKeyFocus: false

Mwm*clientAutoPlace: false

Mwm*focusAutoRaise: false

Mwm*focusFollowsPointer: true

Mwm*keyboardFocusPolicy: pointer
```

If you have a gray-scale monitor, you might try the following settings:

```
Mwm*activeBackground: #666666 (gray40)

Mwm*activeForeground: #e5e5e5 (gray90)

Mwm*background: #666666 (gray40)
```

```
Mwm*foreground: #e5e5e5 (gray90)
Uimx3_0*calculatedColors: false
Uimx3_0*background: #ededed (gray93)
Uimx3_0*BottomShadowColor: #000000 (black)
Uimx3_0*foreground: #000000 (black)
Uimx3_0*TopShadowColor: #ffffff (white)
Uimx3_0*XmText.background: #b3b3b3 (gray70)
Uimx3_0*XmTextField.background: #b3b3b3 (gray70)
```

Note: The resources above prefixed with `Mwm` are specific to the Motif Window Manager. If you are using a different window manager, consult your Systems Administrator for the equivalent settings.

Overview of Motif Widgets

1

Overview

Motif is a programming toolkit that provides the look and feel of programs that run on X Windows with graphical user interfaces. Motif was created by the Open Software Foundation (OSF), as a tool that would allow programs to run on computers from a wide range of manufacturers. Motif is based on the X Toolkit Intrinsics (Xt), the usual standard upon which X Windows applications are organized. Xt widgets and gadgets can be assembled to build the visible and behavioral attributes that make up a graphical application. Motif provides a set of widgets and gadgets that are based on Xt widgets and gadgets, but that follow the Motif Style Guide.

Working with the Ux Palette

UIM/X provides a comprehensive palette of objects for building user interfaces. The Ux Palette, shown in Figure 1-1, includes the full Motif widget set. For convenience, the Ux Palette also includes pre-built menus and dialogs, as well as a set of compound objects. The compound objects are pre-configured groups of Motif widgets that are also available in novice mode. (Novice mode is a simplified version of UIM/X intended for newcomers to Motif and UIM/X. For more information on novice mode, see the *UIM/X Beginner's Guide*.) The Ux Palette provides several approaches to creating objects:

- Drag and drop.
- Click, point, and click.
- Click, point, and drag to interactively create and size the object. Holding down the Shift key allows you to create arrays of objects.

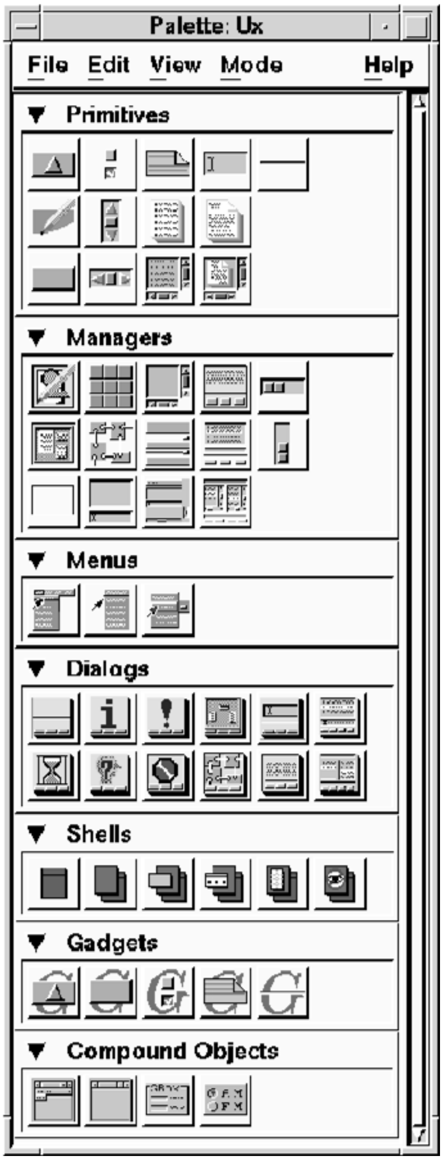


Figure 1-1 Ux Palette

The Primitives Category

The thirteen primitive widgets provided by the Ux Palette are shown in the following table, along with their names, suggested uses, and how the end user activates them in your interface. Primitive widgets can be top-level widgets or the children of Shell, Manager, or Dialog widgets.

Name	Icon	Suggested Uses
Arrow Button	arrow Button	Purpose: To provide directional scrolling. Use the Arrow Button to display an arrow pointing up, down, left, or right and used for much same purpose as a pushbutton. To activate an Arrow Button, click it.
Drawn Button	drawn Button	Purpose: A button widget with functionality similar to a pushbutton, but with an empty widget window. The application provides the graphics to be displayed in the widget. Use a Drawn Button as a pushbutton, with a graphic image instead of a label. To activate a Drawn Button, click it.
Horizontal ScrollBar	horiz ScrollBar	Purpose: Used to implement scrolling in scrolled windows or in other windows you build. Use a Horizontal ScrollBar to bring different areas of a large object into view. To activate a Horizontal ScrollBar, click one of its arrows, drag its indicator to the desired position, or click on the area between the indicator and the arrow.
Label		Purpose: Used to display either text or pixmaps. Use a Label to provide an identifying name or icon to part of your interface. A Label cannot be activated.

Name	Icon	Suggested Uses
List	list	Purpose: Provides a dynamically updated selectable list of items. Use a List to display a collection of text strings, and to make selections from them. To activate a List, click within it.
Push Button	push Button	Purpose: Used to activate an operation. Use a Push Button to activate each of the main functions of your interface. A Push Button can contain a text label or a pixmap. To activate a Push Button, click it.
Scrolled List	scrolled List	Purpose: Provides a scrolled window with a List object work area. Use a Scrolled List to display a collection of text strings that is too large to fit into your interface. To activate a Scrolled List, click on the text strings within it.
Scrolled Text	scrolled Text	Purpose: Provides a scrolled window with a Text object work area. Use a Scrolled Text to display an editable text area that is too large to fit into your interface. To activate a Scrolled Text, click within it and type.
Separator	separator	Purpose: Used to separate child widgets in a Manager widget. Various line styles are available. Use a Separator as a visual design element to help organize your interface. A Separator cannot be activated.

Name	Icon	Suggested Uses
Text		Purpose: Functions as a single-line or multi-line text editor. Use a Text object to accept typed input from the end user. To active a Text object, click within it and type.
Text Field		Purpose: To provide a single-line text editor widget with higher performance than the Text widget. Use a Text Field to accept typed input from the end user. To activate a Text Field, click within it and type.
Toggle Button		Purpose: Used to permit the user to toggle an option between on and off. Contains text or pixmap and an indicator box showing the current status of the widget. Used to display and change the state of a binary variable. To activate a Toggle Button, click it.
Vertical Scroll Bar		Purpose: Used to implement scrolling in scrolled windows or in other windows you build. Use a Vertical ScrollBar to bring different areas of a large object into view. To activate a Vertical ScrollBar, click one of its arrows, drag its indicator to the desired position, or click on the area between the indicator and the arrow.

The Managers Category

The fourteen Manager objects provided by the Ux Palette are shown in the following table, along with their names, suggested uses, and how the end user activates them in your interface. Manager widgets control their children in a particular fashion, according to the type of Manager created. Manager widgets can be top-level widgets or the children of Shell or other Manager widgets.

Name	Icon	Suggested Uses
Bulletin Board		Purpose: A general layout widget used as the foundation for building dialogs. It may arbitrarily have many children and imposes some restrictions on the movement of children. Use a Bulletin Board to build dialogs and as a general container. A Bulletin Board cannot be activated.
Command		Purpose: Provides a command history mechanism including a message area, an input region, a scrolling list of previous commands, and command buttons. Use a Command to provide a command history to the end user. To activate a Command object, click on its scrollbars and type into its text field.
DrawingArea		Purpose: A general layout widget that may have many children and imposes no constraints on the layout of children. Use a Drawing Area to display graphics. The Drawing Area can be activated by its `InputCallback`.

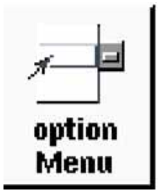
Name	Icon	Suggested Uses
File Selection Box	file SelBox	Purpose: Provides a file selection mechanism including a field for the directory mask, a scrolling list of file names, a scrolling list of directories, an editable input field for the selected file, and command buttons. Use a File Selection Box to allow the end user to select files. To use a File Selection Box, click on the Directories and Files list boxes until you see your desired file, then double-click on the file name, or select the file and click OK.
Form	form	Purpose: A layout widget that may have many children and allows you to specify constraints on the location of children. Use a Form to ensure that a particular layout is proportionately maintained even when the user resizes the window. A Form cannot be activated.
Frame	frame	Purpose: Provides a border to enclose a single child, usually another manager. Use a Frame to provide an etched or three-dimensional border decoration for its work area child. A Frame is useful when you want your work area object to have the same border appearance as the Primitives it contains. A Frame cannot be activated.

Name	Icon	Suggested Uses
Horizontal Scale	horiz Scale	Purpose: Allows the user to select a value from a range of values by positioning a slider inside an elongated rectangular region. Use a Horizontal Scale to indicate a value from within a range of values, and to allow the end user to modify that value. To activate a Horizontal Scale, click and hold its drag button, and drag it left or right.
Main Window	main Window	Purpose: A layout widget that provides a standard layout for the primary window of an application and automatically manages its children. Use a Main Window when you want your primary window to include such items as a menu bar, a command window, a work area, and scroll bars. You act on the Main Window with the Main Window Editor. A Main Window cannot be activated.
Message Box	message Box	Purpose: Provides a means to pass information to the user and includes a symbol, a message area, and command buttons. Use a Message Box to provide information to the end user. To activate a Message Box, click on its pushbuttons.
Paned Window	paned Window	Purpose: A composite widget that lays out children in a vertical column. Control sashes are provided between children. Use a Paned Window when you want the manager's children to appear in a vertical column. To activate a Paned Window, press and hold the left mouse button on the control sash, and move the mouse up or down.

Name	Icon	Suggested Uses
Row Column		Purpose: Provides a general purpose manager that lays out its children in rows and columns. Use the Row Column when you build a matrix of widgets, and as the underlying widget when you build your own menus. To specify a certain cell in a Row Column, select it using the mouse or arrow keys.
Scrolled Window		Purpose: A layout widget that may have one child and may be configured to provide scroll bars when the size of the child exceeds the size of the scrolled window. Use a Scrolled Window when the interface is expected to display a large data display. To activate a Scrolled Window, click on its scrollbars.
Selection Box		Purpose: Provides a means for a user to make a selection from among a number of alternatives. It provides a message area, an editable field, a scrolling list of choices, and command buttons. Use a Selection Box to allow the user to select from a list of alternatives. To activate a Selection Box, click in its text field and type, and click its pushbuttons.
Vertical Scale		Purpose: Allows the user to select a value from a range of values by positioning a slider inside an elongated rectangular region. Use a Vertical Scale to indicate a value from within a range of values, and to allow the end user to modify that value. To activate a Vertical Scale, click and hold its drag button, and drag it up or down.

The Menus Category

The three Menu objects provided by the Ux Palette are shown in the following table, along with their names, suggested uses, and how the end user activates them in your interface.

Name	Icon	Suggested Uses
Menu Bar		Purpose: A horizontal bar across a window, just below the title bar. A menu bar displays a list of menus, such as File, Edit, and Help. Clicking on a menu displays a pulldown menu. Use a Menu Bar to add a menu bar to your interface. To activate a Menu Bar, click it to open one of its menus.
Option Menu		Purpose: Permits the user to select one option from two or more mutually exclusive options. Use an option menu to present a group of choices from which a user can choose only one. To activate an Option Menu, click on it with the left or right mouse button, then click on the menu item you wish to select.
Pop-up Menu		Purpose: A context-sensitive menu that pops up when the user presses the right mouse button over a given area of the user interface. Typically used to provide shortcuts. Use a Pop-up Menu when you want a menu that appears only when the end user needs it. To activate a Pop-up Menu, press and hold the right mouse button over the interface.

The Dialogs Category

The twelve Dialog objects provided by the Ux Palette are shown in the following table, along with their names, suggested uses, and how the end user activates them in your interface. Dialog widgets are a collection of widgets used for file selection or to display warning messages. Dialog widgets are top-level widgets and can never be child widgets, although they can be instances of a component.

Name	Icon	Suggested Uses
Bulletin Board		Purpose: An empty Bulletin Board widget in a Board Dialog Shell widget. Use a Bulletin Board Dialog when you wish to build a custom dialog. A Bulletin Board Dialog cannot be activated.
Error		Purpose: To inform the user of run-time errors. Use an Error Dialog to indicate to the user such errors as typing a non-existent file name. To activate an Error Dialog, click its pushbuttons.
File Selection Box		Purpose: To enable an end user to select a file. Use a File Selection Dialog to enable the end user to select a file to open, view, modify, save, load, or delete. To activate a File Selection Dialog, click on the Directories and Files list boxes until you see your desired file, then double-click on the file name, or select it and click the OK button.
Form		Purpose: A form widget inside a Dialog Shell widget. Use a Form Dialog when building a custom dialog that the user can resize. A Form Dialog cannot be activated.

Name	Icon	Suggested Uses
Information	info Dialog	Purpose: A message box showing the Motif information symbol. Use an Information Dialog to display information to the user. To activate an Information Dialog, click on its pushbuttons.
Message Box	msgBox Dialog	Purpose: A message box with no symbol. Use a Message Box to display a message to the user. To activate a Message Box Dialog, click on its pushbuttons.
Prompt	prompt Dialog	Purpose: A message box showing a prompt symbol and containing a text field. Use a Prompt Dialog to request input from the user. To activate a Prompt Dialog, type into its text field, then click on its pushbuttons.
Question	question Dialog	Purpose: A message box showing a question symbol. Use a Question Dialog to request a simple Yes or No answer from the user. To activate a Question Dialog, click on its pushbuttons.
Selection Box	selBox Dialog	Purpose: A selection box. Use a Selection Box to permit the user to choose from a list of items. To activate a Selection Box Dialog, type into its text field, then click on its pushbuttons.

Name	Icon	Suggested Uses
Template	template Dialog	Purpose: A template that you can use to construct your own dialogs. Use a template Dialog as a base container when building a custom dialog. A Template Dialog cannot be activated.
Warning	warning Dialog	Purpose: A message box showing a warning symbol. Use a Warning Dialog to display a warning message, to prevent the user from accidentally performing destructive actions. To activate a Warning Dialog, click on its pushbuttons.
Working	working Dialog	Purpose: A message box showing a working symbol. Use a Working Dialog to show the user that the application is processing. To activate a Working Dialog, click on its pushbuttons.

The Shells Category

The Shell objects provided by the Ux Palette are shown in the following table, along with their names, suggested uses, and how the end user activates them in your interface. Shell widgets function as top-level interfaces and allow you to dictate features of the widget's interaction with the window manager. Shell widgets can never be children.

Name	Icon	Suggested Uses
Application Shell	appl Shell	Purpose: A shell meant to enclose an application's primary window. Use an Application Shell for an application's primary top-level window. An Application Shell cannot be activated.

Name	Icon	Suggested Uses
Dialog Shell	dialog Shell	Purpose: A shell meant to enclose Dialog widgets. Use a Dialog Shell for Dialog widgets. The Dialog Shell is a subclass of the Transient Shell. A Dialog Shell cannot be activated.
Override Shell	override Shell	Purpose: A shell that is unaffected by the Shell window manager. Use an Override Shell for windows that you wish to bypass the window manager, like pop-up menus. An Override Shell cannot be activated.
TopLevel Shell	topLevel Shell	Purpose: A shell meant to contain a top-level widget. Use a Top-Level Shell for top-level widgets (other than the primary Application Window). A Top-Level Shell can be manipulated and iconified by the window manager. A TopLevel Shell cannot be activated.
Transient Shell	transient Shell	Purpose: A shell you can use for Dialog objects. A Transient Shell can be manipulated by the window manager, but not separately iconified. Use a Transient Shell to enclose a dialog object. A Transient Shell cannot be activated.
Non-Visual Shell	nonVisual Shell	Purpose: A shell you can use to enclose a non-Shell visual object. Use a Non-Visual Shell to enclose a non-visual object. A Non-Visual Shell cannot be activated.

Note: The Non-Visual Shell is not an official Motif widget. See the *UIM/X Tutorial Guide* for an example using a Non-Visual Shell.

The Gadgets Category

Gadgets are somewhat less flexible than Primitive widgets (they have fewer resources associated with them) but they are more efficient. Gadgets must always be the children of Manager or Dialog widgets. They can never be parent widgets.

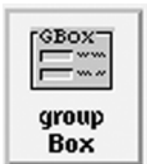
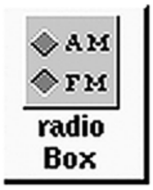
Name	Icon	Suggested Uses
Arrow Button		Purpose: A gadget that displays an arrow pointing up, down, left, or right and used for much the same purpose as a pushbutton. Use an Arrow Button Gadget when you want a directional pushbutton. To activate an Arrow Button Gadget click it.
Label		Purpose: A gadget able to display either text or pixmap. Use a Label Gadget to provide an identifying name or icon to part of your interface. A Label Gadget cannot be activated.
Push Button		Purpose: A pushbutton gadget containing a text label or pixmap. Use a Push Button Gadget to activate each of the main functions of your interface. To activate a Push Button Gadget, click it.
Separator		Purpose: A gadget used to separate items in a display. Various line styles are available. Use a Separator Gadget as a visual design element to help organize your interface. A Separator Gadget cannot be activated.

Name	Icon	Suggested Uses
ToggleButton	toggleB Gadget	Purpose: Used to allow the user to toggle an option between on and off. Use a Toggle Button Gadget to display and change the state of a binary variable. To activate a Toggle Button Gadget, click it.

The Compound Objects Category

The Compound objects provided by the Ux Palette are shown in the following table, along with their names, suggested uses, and how the end user activates them in your interface. Compound objects consist of several Motif widgets combined into one object for your convenience. While these compound objects are not official Motif widgets, they still follow the *OSF/Motif Style Guide*.

Name	Icon	Suggested Uses
Application Window	appl Window	Purpose: A compound object designed to get you started quickly. The window includes a built-in title bar, a menu bar with File and Help pull-down menus, and a work area where you can add more objects. Use an Application Window as the main window in your project. To activate an Application Window, click it to open one of its menus.
Secondary Window	secondary window	Purpose: A window that is transient, and is associated with another, primary window. Use a Secondary Window in the same way as you would use an Application Window. You cannot add a menu bar to a Secondary Window. A Secondary Window cannot be activated.

Name	Icon	Suggested Uses
Group Box		Purpose: A compound object designed to create a visual grouping in your interface. A Group Box includes a label. Use a Group Box to give a name to a group of related objects. A Group Box cannot be activated.
Radio Box		Purpose: A compound object that lets the user select one option among several mutually-exclusive options. Use a radio Box to present a group of choices from which the user can choose only one. To activate a Radio Box, click one of its radio buttons.

Programming in UIM/X 2

Overview

To help you specify Motif behavior, UIM/X comes with a small, easy-to-use convenience library of Ux functions.

The Ux Convenience Library performs such complex tasks as converting resources, allocating colormap entries, automatically managing children of dialog shells, and handling special cases of geometry management. By using the Ux Convenience Library, you do not need to learn the complex Motif Function Library and programming style.

As you develop your application with UIM/X, the Ux Convenience Library performs substantial error checking, notification, and recovery to help in the debugging process.

However, when you link your finished application, you link to a high-performance version of the Ux Convenience Library without error checking. Source to the library is also available for porting.

Full information about the Ux Convenience Library can be found in the *UIM/X Reference Manual*.

Setting Property Values

For each property of each widget class, the Ux Convenience Library provides a `UxPut` function to set the property value and a `UxGet` function to retrieve the property value. The names of these functions are obtained by prefixing `UxPut` or `UxGet` to the property's name. Thus, for the `Width` property, there are two functions: `UxPutWidth()` and `UxGetWidth()`.

The `UxGet` function takes one argument: the name of the widget. It returns the value of the property:

```
int value;
value = UxGetY(form1);
```

The `UxPut function` takes two arguments: the name of the widget, and the new value for the property. For example, the following line sets the `Height` of the `form1` widget to 120:

```
UxPutHeight(form1, 120);
```

Each property has a specific data type, such as `int`, `float`, or `string`. For the data types of particular properties, refer to **Appendix A, "Object Property Values**."

The first argument is of type swidget. For each widget you create, UIM/X maintains a small structure. A swidget is a pointer to this structure. Swidgets are used to identify the widgets in the interface.

The second argument to the `UxPut` function is the value for the property as it would appear in the Property Editor.

Using C++ Bindings

The Ux Convenience Library also includes a set of lightweight Motif wrapper classes. When UIM/X is in C++ mode, and when C++ project code is generated with the "Use Ux C++ Convenience Library bindings" option set, all Motif objects are declared as objects of these wrapper classes. Such objects can be implicitly converted to swidgets, and thus standard Ux Convenience Library functions such as the one below will still work:

```
UxPutWidth(form1, 75);
```

However, the Motif wrapper classes define actual C++ member functions for setting and retrieving property values. This allows you to take advantage of C++ syntax and error checking while manipulating Motif widgets.

For each property of each widget the corresponding wrapper class provides two member functions. A Set function sets the property value and a Get function retrieves the property value. The Set function takes one argument, the property value, and returns `void`. The Get function takes no arguments and returns the property value, as shown in the following example:

```
int value ;

value = form1.GetY();

form1.SetWidth(75);
```

Note: The preceding example illustrates a difference in terminology. The standard Ux Convenience Library functions are named “UxPut...()” while the corresponding member functions of the C++ wrapper classes are named “Set...()”.

Miscellaneous Ux Library Functions

This section describes several Ux Convenience Library functions and features that are useful when creating an interface.

UxPopupInterface(swidget iface, grabtype gtype)

This function makes an interface visible on the screen once it has been created with its Interface Function. Sometimes interfaces pop-up other interfaces, creating a cascade of interfaces. The `gtype` argument allows you to specify how the user interacts within a cascade of interfaces. The grab type can be: `no_grab`, `nonexclusive_grab`, or `exclusive_grab`.

- `no_grab` allows the user to interact with any window on the screen.
- `nonexclusive_grab` allows the user to interact with any widget in the interface cascade, but not with widgets outside the cascading interfaces.
- `exclusive_grab` limits the user to only the interface that you are popping up (for example, a popup menu).

UxPopdownInterface(swidget iface)

This takes a visible interface and removes it from the screen. It also removes the grab that the interface has, if any.

UxDestroyInterface(swidget iface)

This routine destroys an interface. In Test Mode, the Interpreter informs you that the function was called, however the interface is not destroyed.

Map and Unmap Functions

These functions make a widget disappear or reappear.

Function	**Use**
`UxUnmap()`	Removes a widget from the screen.
`UxMap()`	Redisplays a widget on the screen.

Widget Functions

These functions can be used to convert between various representations of a widget: the X widget pointer, the UIM/X swidget pointer, and the widget name (a character string).

Function	**Use**
`UxFindSwidget()`	Given the name of a widget, returns its swidget pointer.
`UxWidgetToSwidget()`	Given an X widget, returns its swidget pointer.
`UxGetWidget()`	Given a swidget, returns its X widget pointer.
`UxGetName()`	Given a swidget, returns its widget name.

Full information about the Ux Convenience Library can be found in the *UIM/X Reference Manual*.

Using the CreateCallback

UIM/X provides a CreateCallback for every widget. The CreateCallback is not a Motif callback (it is not even really a callback). It is a function inserted into the generated code.

The CreateCallback allows you to define a function called immediately after the widget is created, but before it is realized:

```
static swidget _Uxbuild_drawingArea1()
{
/*
 * The Interface Function calls this function to
 * create the swidgets.
 */


   drawingArea1 = UxCreateDrawingArea(
        "drawingArea1", UxParent );
   /* … */
   UxCreateWidget( drawingArea1 );
   /* the CreateCallback */
   createCB_drawingArea1( UxGetWidget(
        drawingArea1 ),(XtPointer)
        UxDrawingArea1Context, (XtPointer) NULL );
   /* … */
   pushButton1 = UxCreatePushButton
        ( "pushButton1", drawingArea1);
   /* … */
   UxCreateWidget( pushButton1 );
   createCB_pushButton1( UxGetWidget( pushButton1
      ),(XtPointer) "CreateCallback ClientData",
      (XtPointer) NULL );
   /* … */
   UxRealizeInterface( drawingArea1 );
}
```

Note that the CreateCallback of a parent should not reference its children, since they have not yet been created.

In UIM/X, the CreateCallback is called only if the Interface Function is explicitly called (for example, if the Interface Function is called from a callback, or evaluated in the Interpreter window).

Using UIM/X Global Variables

For certain X and X Toolkit functions, the following variables may be needed. These are defined globally by UIM/X and do not need to be declared before use:

Global Variable	**Description**
`Display *UxDisplay;`	The X Display.
`int UxScreen;`	The X screen.
`Widget UxTopLevel;`	The top-level widget returned by `XtAppInitialize()`.

Using Xm, Xt, and X Calls

Callbacks can include any Xm, Xt, or X function call—they are already linked into UIM/X. These calls typically take widgets or windows as arguments, but *not* swidgets.

To obtain a pointer to the Motif widget of a swidget in a callback, simply call `UxGetWidget()`:

```
{
  Widget w;
  w = UxGetWidget(mySwidgetName);
  /* — your Xm code — */
}
```

To get the X window of a Motif widget in a callback, use `XtWindow`():

```
{
  Widget w;
  Window xw;
  w=UxGetWidget(mySwidgetName);
  xw=XtWindow(w);
  /* — your Xm and X code — */
}
```

Note: Xm function calls are much less tolerant of errors than Ux Convenience Library calls. In general, Ux Convenience Library calls have better error-checking.

Generating Xt Code

If you intend to generate Xt code, you have to write code that works both in UIM/X and in generated Xt code. What this means is that you have to write code that works with swidgets in UIM/X, but with widgets in the generated code.

This does not mean extra work for you, though. UIM/X does all the work for you. The purpose of this section is to help you understand what UIM/X does to translate swidget code into widget code when you generate Xt code.

Writing Xt Code in UIM/X

No matter what kind of code you generate, UIM/X deals with swidgets, so the code you write in UIM/X has to operate on swidgets. You can write straight Xt code in a UIM/X editor, but you'll have to call `UxGetWidget()` on each swidget you reference in your code:

```
XmString text;
text = XmStringCreateLocalized( "Apply" )
XtVaSetValues( UxGetWidget( pushButton1 ),
   XmNlabelString, text,
   NULL );
XmStringFree( text );
```

In this example, `UxGetWidget()` returns the widget for the swidget `pushButton1`. Only when you generate Xt code does `pushButton1` become a widget. UIM/X handles this by redefining `UxGetWidget()` as a macro:

```
#define UxGetWidget(sw) (sw)
```

So code that works in UIM/X, where everything is a swidget, still works in generated Xt code, where everything is a widget.

Note that when you write callback and action code, you can still use the variable `UxThisWidget` to refer to the swidget. For example, if you were writing a callback for the swidget `pushButton1`, you could simply refer to

`UxThisWidget` instead of calling `UxGetWidget()` on `pushButton1`. In the generated Xt code, `UxThisWidget` is redefined as `UxWidget`, the actual widget passed into the callback:

```
#define UxThisWidget UxWidget
```

Understanding the Generated Xt Code

When you generate Xt code, UIM/X copies several files from *uimx_directory*`/config` into your local directory. Two of these files, `UxXt.h` and `UxXt.c`, provide the necessary support for Xt code generated by UIM/X. You can find out what Ux functions and symbols are available in Xt code by examining both these files.

Each interface header file includes `UxXt.h` (instead of `UxLib.h`, the standard header for the Ux Convenience Library). You can think of `UxXt.h` as an Xt version of `UxLib.h`. `UxXt.h` makes the swidget-oriented code you write in UIM/X compatible with the widget-only generated Xt code.

For example, `UxXt.h` redefines `swidget`, `UxThisWidget`, and `UxGetWidget()`:

```
#define swidget Widget

#define UxThisWidget (UxWidget)

#define UxGetWidget(sw) (sw)
```

The file `UxXt.c` redefines a number of Ux functions so that they no longer depend on the Ux Convenience Library. Many of these functions, such as `UxPutContext()` and `UxGetContext()`, are part of the framework of the generated code, not part of the code you write.

For your convenience, `UxXt.c` defines Xt versions of a number of Ux functions, such as `UxPopupInterface()`, `UxPopdownInterface()`, and `UxDestroyInterface()`.

When you generate C code, UIM/X also copies *uimx_directory*`/config/UxMethods.c` into your local directory. This file provides C-language support for interface methods.

When you generate C++ code, UIM/X copies *uimx_directory*`/config/UxInterf.cc` into your local directory. This file provides the implementation of the base C++ class for all interface classes generated by UIM/X. (This base class is defined in `UxXt.h`.)

Setting Properties

When you generate Xt code, you cannot use the `UxPut`*Property*`()` and `UxGet`*Property*`()` functions provided by the Ux Convenience Library. However, Xt versions of many of the `UxGet` and `UxPut` functions are provided in the directory *uimx_directory*`/contrib/XtCodePuts`. See the `README` in that directory for more information.

The reference page for `UxGet`*Property*`()` lists the properties which have Xt versions of the `UxGet` and `UxPut` functions. See the *UIM/X Reference Manual*.

Using the Motif Components

3

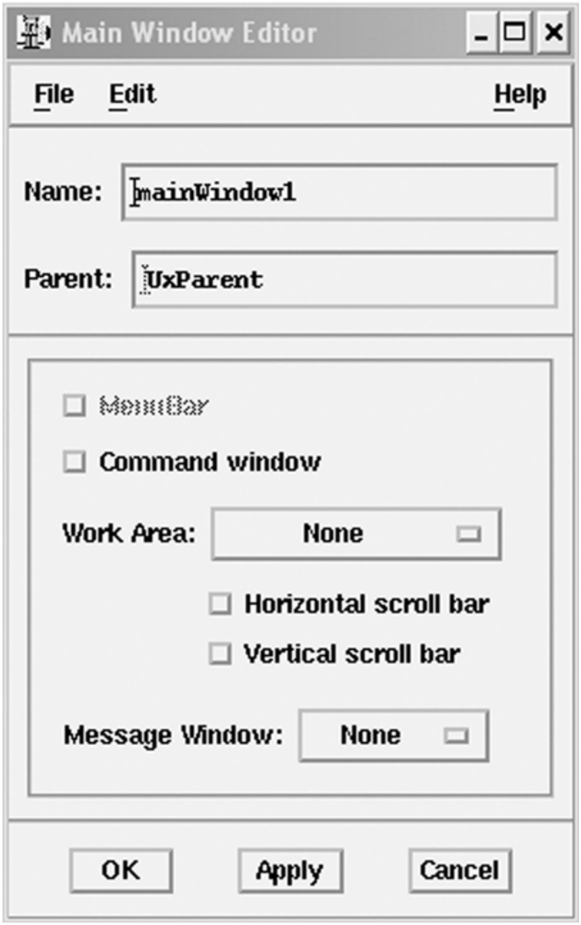
Overview

The Ux Palette includes a set of objects for Motif development. Motif objects are abstractions of common user interface objects, such as dialogs and option buttons.

This chapter describes some of the advanced Motif objects, the Main Window, the List, and the Radio Box. It discusses some of the issues involved in building menus. It explains how to use convenience dialogs, and how to establish a default button. Finally, it describes some of the techniques you use to communicate with the window manager.

Building Main Windows

Most applications have a main window with a pulldown menu and a work area. The OSF/Motif Toolkit provides a small set of convenience functions for creating and working with main windows. Within UIM/X, you need not worry about these functions because the Main Window Editor provides the means to develop main windows.



Name: mainWindow1
Parent: UxParent
MenuBar
Command window
Work Area: None
Horizontal scroll bar
Vertical scroll bar
Message Window: None

Figure 3-1 Main Window Editor

The purpose of the Main Window Editor is similar to that of the Motif function `XmMainWindowSetAreas()`. It identifies the objects that become key parts of the main window. You use toggle buttons and option menus to choose the elements that you want added to the main window.

Opening the Main Window Editor

There are several ways to open the Main Window Editor:

1. Choose Create⇒Managers⇒Main Window from the Project Window. The Main Window Editor appears automatically.

Alternatively, you can open the Main Window Editor for a Main Window that already exists:

1. Create a Main Window from the Palette.
2. Double-click on the Main Window object.
   OR
3. Choose Selected Objects⇒Tools Main⇒Window Editor.
   OR
4. Choose Tools⇒Main Window Editor from the Project Window. The Main Window Editor appears.

Working with a Main Window

1. Create a Main Window object and open the Main Window Editor.
2. Select the elements to make up the main window by clicking on the appropriate toggle buttons with the Select mouse button. Until you have created a menu bar and pulldown menus using the Menu Editor, the MenuBar toggle button is insensitive.
3. Having made your selections, click OK to apply them and close the Main Window Editor. Click Apply if you want to apply the changes without closing the Main Window Editor. Click Cancel to close the Main Window Editor and cancel your selections.

Modifying an Existing Main Window

To modify an existing Main Window, you must display the Main Window Editor.

Modifications are made in much the same way as the main window is created: by selecting toggle buttons, selecting a work area object, scroll bars, or a message window object, adding a menu bar and pulldown menus, and so on. Note that modifications to the menu bar are made through the Menu Editor. The Main Window Editor can only delete a menu bar.

Changes to a main window usually involve the deletion of the objects originally included in the Main Window object. For example, you might change the object originally selected as a work area, choosing a different one and thereby destroying its predecessor.

Alternatively, you might choose to delete a now unwanted work area object by specifying None. Again, the predecessor would be destroyed.

When you click OK or Apply to apply any changes made to an existing main window, a dialog box prompts you to confirm the changes. The dialog box allows you to confirm all changes that destroy existing objects in the Main Window. The dialog box is not displayed if you add a work area or message window object, displacing only the value None.

When you confirm the changes by clicking OK in the dialog box, the main window is redrawn according to your changes.

Accessing the Property Editor

You can use the Property Editor to change the properties of the MainWindow object itself or any child object of the MainWindow.

To Load a MainWindow into the Property Editor

With most interfaces, all you have to do is double-click on an object and the Property Editor appears, loaded with that object. When you double-click on a Main Window or one of its children, the Main Window Editor appears instead. However, the Main Window Editor provides an alternate way to load a Main Window into the Property Editor:

1. Open the Main Window Editor.
2. Choose Edit⇒Properties from the Main Window Editor.

   The Property Editor is displayed, loaded with the Main Window object.

Double-clicking on a Main Window’s child opens the Main Window Editor instead of the Property Editor. To load a Main Window’s child into the Property editor, open the Property Editor first, and then use any of the other loading methods. To learn the basics of how to use the Property Editor, see the *UIM/X User’s Guide* and the *UIM/X Beginner’s Guide*.

Using the motifMain.prj Example

If your application needs a main window, you might consider using the file *uimx_directory*/`contrib/MotifMain/draw_start.prj` as a starting point. This file contains an example main window interface with the following features:

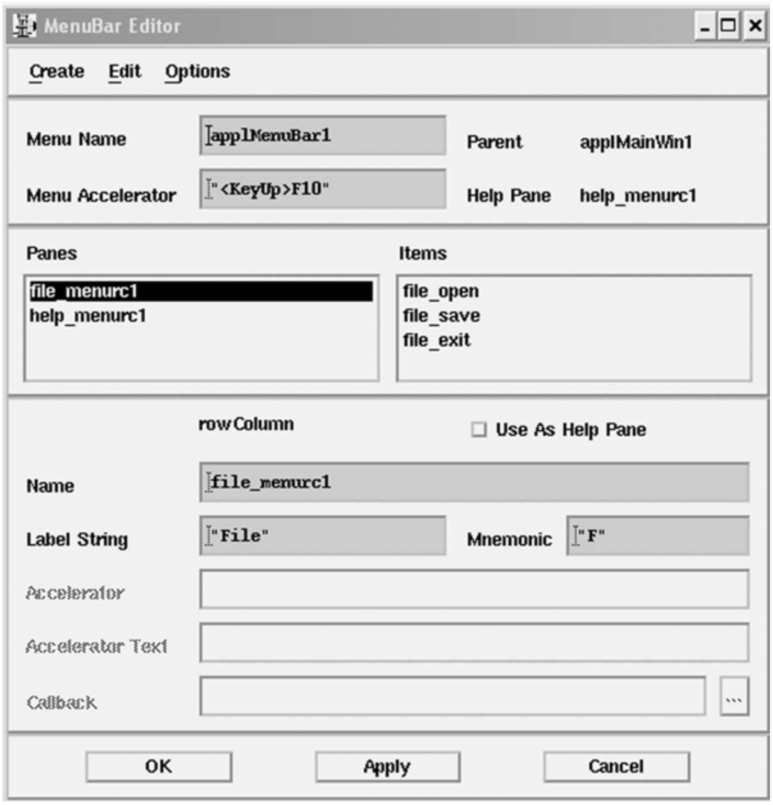
- A menu bar featuring the following:
  - All the pulldown menu commands described in the *OSF/Motif Style Guide*.
  - A View menu that demonstrates how to use radio buttons in menus. Radio buttons ensure only one selection can be made at a time. This pane achieves the radio button behavior by setting the `RadioBehavior` property of the menu pane to `true`.
  - An Option menu that demonstrates how to mix a set of radio buttons with a set of check boxes in the same menu pane. Here, the mutually exclusive behavior of the radio buttons is accomplished within the callbacks for the toggle buttons.
  - Keyboard mnemonics for all menu commands. These are displayed as underlined characters within each menu command.
- A work area. If you want to add something to the work area, create a manager object and make it a child of the form. If you don't want these existing managers, simply delete them and add your own work area object.
- Two application-modal dialogs:
  - A file selection box that is displayed by choosing either File⇒Open or File⇒Save As from the menu bar. If you select a file while using this interface in Test mode, the file name is echoed to standard output using `printf()`.
  - An exit dialog that is displayed by choosing Exit from the File menu, or Close from the window menu (provided by the window manager). If you click OK, the `exit()` function is called. In Test mode, the Interpreter catches this function and displays a message.

    To demonstrate the modality of these dialogs, you must switch to Test Mode, then execute the main window's Interface Function, `popup_mainWS()` in the Interpreter. Choose the appropriate commands from the menus. You cannot display both dialogs at once.

Building Menus

The Menu Editor, shown in Figure 3-2, allows you to build pulldown, option, and pop-up menus. Menus are composed of panes and items. A pane is a RowColumn object containing a list of items, and items are the selections offered. An item can either perform an operation or display a cascading menu (a subordinate pane).

The basics of building menus with the Menu Editor are covered in the *UIM/X Beginner's Guide*.



Menu Name: applMenuBar1
Parent: applMainWin1
Menu Accelerator: "<KeyUp>F10"
Help Pane: help_menurc1

Panes: file_menurc1, help_menurc1
Items: file_open, file_save, file_exit

Name: file_menurc1
Label String: "File"
Mnemonic: "F"

Figure 3-2 Menu Editor

Pulldown Menus

Menu bars are composed of several pulldown menus arranged horizontally across a top-level or manager object. The pulldown menus in a menu bar typically offer global operations or operations on selected objects. Pulldown menus are accessed with the Select mouse button.

Pulldown menus can only be the children of top-level or manager objects (for example, a Form object).

The menu bar of the Project Window provides several examples of pulldown menus. The File, Create, Edit, View, Options, Mode, Tools, and Help selections each drop a submenu, each with its own set of operations.

Option Menus

Option menus are composed of a *single principal pane* containing a single list of one or more mutually exclusive items. Note that an Option menu cannot contain submenus. The list of items on an option menu is accessed by the Select mouse button.

Option menus can only be the children of top-level or manager objects.

The Category menu in the Property Editor is an example of an option menu. The Core, Specific, Constraints, All Resources, Behavior, Compound, Declaration, and All items in the menu determine which group of properties is displayed in the Property Editor.

Pop-up Menus

Pop-up menus also consist of a single principal pane, but differ from Option menus by allowing cascade menus. Unlike the other types of menus, a pop-up is only visible when the user presses the Menu mouse button over the object on which the pop-up was created. An object can only have one pop-up menu. Pressing the Menu mouse button on a child of the object also displays the pop-up menu (unless the child has its own pop-up menu).

The Selected Interfaces menu in the Interfaces Area of the Project Window is an example of a pop-up menu.

Menus and Panes

Every menu, whatever its type, begins with a RowColumn object. Beyond that, pulldown, pop-up, and option menus differ slightly in their object hierarchy.

In pulldown menus, the RowColumn object manages the horizontal arrangement of the panes in a menu bar. When adding a pane to a menu and applying the change, UIM/X automatically creates a RowColumn object to hold the pulldown menu pane.

When you define a pane in the Menu Editor and apply your changes, UIM/X automatically creates a cascade button on the menu bar for the pane. Selecting this cascade button displays the pane. The panes and cascade buttons in the menu bar are children of the menu object. Menu items are children of the pane.

When creating an Option menu, a pane is automatically added to manage the arrangement (vertical by default) of the items and display the current selection. This menu type can only contain one pane (that is, no cascade menus are allowed). The pane is the child of the menu object. The menu items are children of the pane.

Note: The OSF/Motif menu shell, present in pulldown and pop-up menus, is transparent in UIM/X. Its resources cannot be accessed.

The pop-up menus also begin with a single top pane (a RowColumn menu object). This RowColumn object serves as the menu's principal pane and manages the arrangement (vertical by default) of the menu items, the children of that object. This menu can contain other panes, but they must be children of the single top pane.

Menu Items

The following object and gadget items can be added to a pane:

Object/Gadget	Pulldown	Pop-up	Option
Push Button	x	x	x
Cascade Button	x	x	
Toggle Button	x	x	
Label	x	x	x
Separator	x	x	x

The function of most items should be clear. However, cascade buttons deserve further discussion. A cascade button is used exclusively to display a subordinate pane, in other words, a cascade menu. For example, in the Project Window, the Create pane offers three cascade menus: Shells, Managers, and Dialogs. A cascade menu may also contain cascade buttons, thus creating a menu hierarchy.

When creating a cascade menu, a Label object is created as a child of the RowColumn if the pane's LabelString field is not the null string ("") when applying changes to the menu. This object provides the title of the cascade menu when it is displayed.

Note: A cascade button can only display one pane. No two cascade buttons may be used to display the same pane.

Reordering Panes and Items

To Reorder Panes (in Pulldown Menus Only) or Items

1. Select the pane or item you want to reorder.
2. Choose either Edit⇒Exchange Before or Edit ⇒Exchange After from the Menu Editor.

   Repeat the process until the pane or item is positioned where desired.

Reparenting Menus and Panes

Any menu object can be reparented through the Property Editor.

To Reparent a Menu Object Using the Property Editor

1. Load the menu object into the Property Editor.
2. Select Declaration from the Property Editor's Category option menu.
3. In the Parent field, enter the name of the new parent object.

To Reparent a Pane in a Menu

1. Create a cascade button in another pane.
2. Add the name of the pane to the Next Pane field.

   The object representing the pane in which the cascade button appears becomes the new parent.

   To reparent a subordinate pane in a pulldown menu to the menu bar, delete the cascade button that calls it.

   In pop-up menus, the first pane cannot be reparented.

Note: Menu items cannot be reparented. Menu objects cannot be promoted to top-level status.

Setting Menu Properties

The Property Editor for a menu, pane, or item can be accessed using the Edit menu in the Menu Editor. However, the menu, pane, or item must exist: click on the Apply button first to create the menu, pane, or item. For example, to open the Property Editor for a pane, create the pane, then choose Edit⇒Properties⇒Pane.

Setting Menu Connections

You can open the Connection Editor from the Menu Editor, with one of your menu objects already loaded as either the Source or the Target object. For example, suppose you want to open the Connection Editor with a menu pane loaded as the Target object:

1. In the Menu Editor’s Panes list, select a pane by clicking on it.
2. Choose Edit⇒Connection To⇒Pane from the Menu Editor.

   The Connection Editor appears, with your selected menu pane loaded as the Target object.

Choosing Edit⇒Connection From works in a similar fashion, except that the selected object will be loaded as the Source object.

Creating a Help Menu Entry

The *OSF/Motif Style Guide* specifies that the help menu selection of the main pulldown menu should appear at the far right of the menu bar, with the other menu selections grouped at the left of the menu bar.

To achieve this behavior, simply select the menu pane you want as your help menu in the Menu Editor, and click on the Use As Help Pane toggle button.

The menu bar should appear at the top of its window, and completely fill the width of the window. To achieve this behavior, a menu bar is typically created as a child of a Main Window or Form object. When you put a menu bar on a Form, you set the `LeftAttachment` and `RightAttachment` constraints to `attach_form`.

Using Convenience Dialogs

Three of the most common techniques for dialog boxes are:

- Making a dialog more flexible by adding parameters to its Interface Function.
- Customizing a dialog by working with the dialog's children objects.
- Reparenting a dialog to make it *application modal*.

Adding Parameters to an Interface Function to Make it Flexible

It is often efficient to use a single interface for several parts of an application. This is especially true for dialogs. For example, if your application opens *and* saves files, you could use the same File Selection box for both operations. The difference between the two operations could be reflected in the dialog's title and selection string.

The following example demonstrates this technique. The Interface Function for the message dialog is declared like this:

```
swidget popup_messageDialog(char *messageString,
   char *title)
```

Within the dialog's Property Editor, these two parameters are used as property values. For example, if you create an instance of this dialog using this call:

```
popup_messageDialog("Hello Joe!", "This is a
   dialog");
```

the dialog is displayed with "Hello Joe!" as a message and "This is a dialog" as a title.

Unmanaging Some Children of Dialogs

Since children of a dialog are created automatically, you cannot access them using the Selected Objects popup menu. However, you can access these objects using functions provided by the Motif Toolkit. Since most dialogs are built using subclasses of XmMessageBox or XmSelectionBox, you can get the widget ID for most children of convenience dialogs using these calls:

- XmMessageBoxGetChild()
- XmSelectionBoxGetChild()

Once you have a widget ID for the child you wish to manipulate, you can use standard Xm and Xt calls. For example, to remove the Help button from a file selection dialog, you could enter this call in the dialog's Final Code section:

```
XtUnmanageChild(XmSelectionBoxGetChild(UxGetWidge
    t(rtrn),XmDIALOG_HELP_BUTTON));
```

Making Dialogs Application Modal

During a session with your application, you may want to force the end-user to respond to a dialog before continuing. This is accomplished by setting the dialog style to application modal. Whenever an application modal dialog is displayed, input is rejected by every other window in that application.

To make a dialog operate this way, you must do two things:

- Change the dialog's parent from `UxParent` to the name of the top-level shell for the application's main window.
- Change the dialog's `DialogStyle` property to `dialog_primary_application_modal`.

The `draw_start.prj` example interface demonstrates this technique with two dialogs. Both are children of `mainWindowShell`, which is the application shell parent of the main window. When either of these dialogs is displayed, input to the main window is refused. To see how this example works, follow these steps:

1. Load *uimx_directory*`/contrib/MotifMain/draw_start.prj` into UIM/X. It may take a few moments to load this interface file.
2. Switch to Test mode.
3. Evaluate this function call in the Interpreter Work Area:

   ```
   popup_mainWS();
   ```

   The three interface windows will disappear (as they are recreated) and then the main window is redisplayed.
4. Choose File⇒Open in the sample main window, not in the Project Window. The File Selection box is displayed, ready for input.
5. While the File Selection box is displayed, try using the File menu again in the sample main window. It should not work. In fact, if your terminal's beeper is active, you should hear a beep each time you click the mouse in the main window.
6. Click OK to close the File Selection box.
7. Switch back to Design mode.

Using System Modal Dialogs

You can also make a dialog system modal, which prevents *all* other windows on the display from receiving input while the dialog is displayed. However, this function is generally advised only for very special applications such as a window manager.

If your application does have a special need to provide system modal dialogs, you should note that the `dialog_system_modal` value for the `DialogStyle` property is ignored in Test mode. This prevents you from locking up your computer with a partially-complete interface. To test system modal behavior you must generate and compile the source code for the interface.

Using the List Object

The List object is used extensively in many applications. It is particularly suited for managing textual lists of related information.

Typically, items in a List represent data that your application is managing. Since the items within a List are usually based on run-time data, List objects are generally left empty during construction. It is up to your application's callback procedures and other functions to add, delete, and manipulate items in a List.

Using the List Convenience Functions

To help manage List objects, the Motif Toolkit provides a number of convenience functions. Before using the List object, you should study the properties, callbacks, and convenience functions described in the *OSF/Motif Programmer's Guide* and the *OSF/Motif Programmer's Reference Manual*.

Since most lists may potentially have too many items to display all at once, the Motif Toolkit provides a scrolled list, which is a list object inside a scrolled window. Unless you have special requirements, you should use scrolled lists for all the lists in your application. A special requirement might be scrolling two lists with a single scrollbar. In this case, it is up to your application to create and handle all of the scrolling.

The following are the Motif List functions:

Function	Description
XmListAddItem()	Adds an item to the List (possibly selected).
XmListAddItemUnselected()	Adds an unselected item to the List.
XmListDeleteItem()	Deletes an item from the List.
XmListDeletePos()	Deletes an item from the List using position number.
XmListDeselectItem()	Deselects the specified item.
XmListDeselectPos()	Deselects the item at the specified position.
XmListDeselectAllItems()	Deselects all items in the List.
XmListItemExists()	Determines if the specified item is in the List.
XmListSelectItem()	Selects an item in the List.
XmListSelectPos()	Selects an item in the List at the specified position.
XmListSetBottomItem()	Puts the specified item at the bottom of the List window.
XmListSetBottomPos()	Puts the specified position at the bottom of the List window.
XmListSetHorizPos()	Sets the position of the horizontal scrollbar (if any).
XmListSetItem()	Puts the specified item at the top of the List window.
XmListSetPos()	Puts the specified position at the top of the List window.

The Connection Editor provides the following List swidget methods:

Method	Description
GetItemCount	Retrieves the total quantity of items in the List.
GetSelectedItemCount	Retrieves the total quantity of selected items in the List.
GetItems	Retrieves all items from the List.
GetSelectedItems	Retrieves the selected items from the List.
SetItems	Sets the items in a List.
SelectItems	Selects the specified set of items in the List.
SelectAllItems	Selects all items in the List.

Method	Description
DeselectItems	Deselects the specified set of items in the List.
DeselectAllItems	Deselects all items in the List.
AddItemsToBeginning	Adds the specified items to the beginning of the List.
AddItemsToEnd	Adds the specified items to the end of the List.
DeleteItems	Deletes the specified set of items from the List.
DeleteItemsAtBeginning	Deletes the specified number of items from the beginning of the List.
DeleteItemsAtEnd	Deletes the specified number of items from the end of the List.
DeleteSelectedItems	Deletes the selected items from the List.
DeleteAllItems	Deletes all items from the List.
ReplaceItems	Replaces a specified set of items with a second set of specified items.
ReplaceSelectedItems	Replaces the selected items with the specified set of items.
GoToItem	Scrolls the List so that the specified item is visible.
GoToBeginning	Scrolls the List so that the first item is visible.
GoToEnd	Scrolls the List so that the last item is visible.

Note: For all swidgets that accept an Itemlist: Itemlist is a comma-separated list of items. Any item which contains a comma or a backslash must have that character preceded by a backslash. The same is true for all swidget methods that return a list of items. Note also that all backslashes must be protected from the C preprocessor. Thus, as an extreme example, to specify an item with the name "\", the argument to the swidget must be specified as "\\\\\\".

The following example routine, `AddThreeItems()`, shows how to add an item to a List. The parameter `list` is presumed to be a list shadow widget created by UIM/X. The `UxGetWidget()` function is used to determine the widget ID (type `Widget`) expected by the `XmListAddItem()` function.

```
/* This function demonstrates how to add items  */
/* to a list. The list is passed as a parameter */
/* to the function as a shadow widget.          */


void AddThreeItems(swidget list) { /* Declare a  */
temporary variable for creating /* the XmString  */
for each item.         XmString item;


/* Add the string "First Item" to the list /* and     */
then free the XmString.                                */

item = (XmString)XmStringCreateLtoR ("First
Item",
   XmSTRING_DEFAULT_CHARSET);
XmListAddItem(UxGetWidget(list), item, 0);
XmStringFree(item);


/* Add the string "Second Item" to the list*/


/* and then free the XmString.             */
item = (XmString)XmStringCreateLtoR
("Second Item",XmSTRING_DEFAULT_CHARSET);
XmListAddItem(UxGetWidget(list), item, 0);
XmStringFree(item);

/* Add the string "Third Item" to the list */


/* and then free the XmString              */
item = (XmString)XmStringCreateLtoR ("Third
Item", XmSTRING_DEFAULT_CHARSET);

XmListAddItem(UxGetWidget(list), item,0);
XmStringFree(item);

}
```

Using Sample Data to Test a List

Since List items are normally based on run-time data, to test your List interface you should have some test data in place and a callback routine or other function that can read the data and create the List items.

Try the `FileToList.i` example interface, demonstrating this technique:

1. Create a working directory to hold the `Lists` contrib:

   ```
   mkdir Lists
   ```
2. Change to the working directory you just created:

   ```
   cd Lists
   ```
3. Copy the contents of the `Lists` contrib into your working directory:

   `cp` *`uimx_directory`*`/contrib/Lists/* .`
4. Start UIM/X. If you already have UIM/X running, save your work and then restart UIM/X by choosing Reset from the File menu.
5. Load the following sample interface file: `FileToList.i`
6. Change to Test mode.
7. Click on the Load List button in the interface. The application loads items into the List from a file named `TestData` stored in the same directory with the interface file.

The Load List button has the following connections:

```
ActivateCallback--->applicationShell1::LoadList("
   TestData")

ActivateCallback--->scrolledList1::DeselectAllIte
   ms()
```

To clear the List, click the Clear List button. This button has the following connection:

```
ActivateCallback--->scrolledList1::DeleteAllItems
   ()
```

The Add To List Text Field allows you to type in entries separated by commas. When this Text Field is active, pressing Return adds the new entries to the bottom of the List. This Text Field has the following connections:

```
ActivateCallback--->textField1::getText()

ActivateCallback--->scrolledList1::AddItemsToEnd(
   item list)
```

After testing a List interface, choose Edit⇒Other⇒Recreate from the Project Window to return it to its initial conditions. To reset an entire interface, choose Selected Objects⇒Other⇒Recreate. The Recreate command causes an existing object and all of its children to be destroyed and then recreated using initial values specified in the Property Editor.

Creating a Radio Box

A radio box is a special configuration of Toggle Buttons within a RowColumn manager object. The RowColumn manager can take on the burden of enforcing radio behavior, which ensures that only one Toggle Button can be selected at a time.

To create a radio box, you create a RowColumn manager object and set its `RadioBehavior` property to `true`. You then create a Toggle Button for each option in the radio box. Refer to the Motif documentation for more information about the RowColumn object.

If you want one toggle to be selected by default when the radio box is first created, you should change its `Set` property to `true`.

Adding Behavior to Each Toggle Button

Within a radio box, the RowColumn manager takes care of the behavior that unselects one Toggle Button at the selection of another. Any additional behavior that should occur when a Toggle Button is selected should be added using the toggle’s `ValueChangedCallback` behavior property.

A Toggle Button’s `ValueChangedCallback` is called whenever the state of the Toggle Button changes. *Be careful*: When you select a Toggle Button in a radio box, *two* `ValueChangedCallback` callbacks are called—first for the toggle being automatically unselected, then for the toggle that was selected.

Therefore, within a value changed callback, it is usually important to determine if the toggle is selected. This is accomplished using convenience functions provided by the Motif Toolkit. For example, in this ValueChangedCallback, the appropriate `printf()` statement is executed based on a test of the Toggle Button using `XmToggleButtonGadgetGetState()` like this:

```
{

  /* This is the ValueChangedCallback for the        */
  /* option1 toggle button gadget. Within this       */
  /* callback, the option1 button can be referenced */
  /* using the built-in local variable UxWidget      */
  /* which is type Widget.                           */

  if (XmToggleButtonGadgetGetState(UxWidget))
  printf("option1 selected!\n");
  else
  printf("option1 unselected!\n");
}
```

This example assumes that `option1` is a gadget. If the toggle used is not a gadget, use `XmToggleButtonGetState()` instead.

If you create a radio box with only two Toggle Buttons, you should not have to add behavior to both buttons' ValueChangedCallback. Rather, add the behavior to one Toggle Button and include a test like the one above that determines whether the toggle was selected or unselected.

Determining Which Toggle Button Was Last Selected

Every time a Toggle Button or Push Button within a RowColumn manager is used, the RowColumn manager updates its own `MenuHistory` property with the widget ID of that child button. Although this feature is primarily intended for use in option menus, you can use the `MenuHistory` property to determine which Toggle Button within a radio box was last selected.

Establishing a Default Button

It is often useful to have a button within an interface that is activated when the user presses Return. The Motif Toolkit supports this idea with the bulletin board `DefaultButton` property. To establish a default button, simply set the `DefaultButton` property to the widget ID of the desired Push Button object or gadget.

Installing Special Accelerators

If the parent of the bulletin board is not a dialog shell, you may have to install accelerators for some of its children so the default button will work. For example, an OK button is added as an accelerator to a Text object by calling this function:

```
void SetupAccelerators(void)
{
   Widget buttonWidget;
   buttonWidget = UxGetWidget(ok_Button);
   XtInstallAccelerators(UxGetWidget(text1),butto
      nWidget);
}
```

Providing a Visual Cue to the Default Button

By convention, default buttons are visually identified with an additional three dimensional frame around the button. The thickness of this frame is determined by the button's `ShowAsDefault` property.

Changing the `ShowAsDefault` property alters *only* the appearance of a button, not its behavior.

Communicating with the Window Manager

Your application's communication with the window manager should be designed to comply with the *Inter-Client Communications Conventions Manual* (ICCCM). The conventions in this book outline the standard methods for X clients to communicate with one another. One special relationship is the one between applications like yours and the window manager.

The general philosophy behind communications with a window manager is that the application cannot assume that any of its requests will be honored.

Therefore, an application's requests to the window manager are often referred to as hints. It is considered bad programming practice to base any critical part of your application's appearance or behavior on window manager functions.

The techniques in this section demonstrate the mechanisms built into the Motif widget set for sending commonly-used hints to the Motif Window Manager (`mwm`). You will learn how to:

- Request specific window manager decorations for an interface window.
- Request specific commands to be included in the window menu.
- Add a callback to detect when the window manager's close command is invoked from the window menu.

If you use these techniques in your applications, always include the `MwmUtil.h` header file (`#include <X11/MwmUtil.h>`).

Requesting Window Manager Decoration

The `VendorShell` widget property `MwmDecorations` is used to set window manager hints for particular window decorations. This property is supported by all of the shell objects in UIM/X, except the Override Shell object.

The `MwmDecorations` property expects an integer value. This value is used by the window manager to determine which decorations it should add to the window. Seven integer values for various decorations are defined in `MwmUtil.h`:

Use This Value:	**For:**
MWM_DECOR_ALL	All window decorations.
MWM_DECOR_BORDER	A border without resize handles.
MWM_DECOR_RESIZE	A border with resize handles.
MWM_DECOR_TITLE	A title bar.
MWM_DECOR_MENU	A window menu button.
MWM_DECOR_MINIMIZE	A minimize button.
MWM_DECOR_MAXIMIZE	A maximize button.

These values are defined so that you can logically OR them together to get the decoration you want. For example, suppose your application has a dialog box that should never be resized, minimized, or maximized by the user. You could set the MwmDecorations property for the dialog's shell to this value:

```
MWM_DECOR_BORDER | MWM_DECOR_TITLE |
    MWM_DECOR_MENU
```

This specifies the decoration to be a border, a title bar, and a window menu button. Remember, these values are integers, so *don't* add any quotation marks.

Requesting Window Manager Commands

Your application can also request changes to the window menu for each interface window. It does so by setting the `MwmFunctions` property. The file `MwmUtil.h` defines the following integer values that you can use to request particular functions in the window menu.

Use This Value:	**For:**
MWM_FUNC_ALL	All window menu commands.
MWM_RUNC_RESIZE	The Resize command.
MWM_FUNC_MOVE	The Move command.
MWM_DECOR_MINIMIZE	The minimize command.
MWM_DECOR_MAXIMIZE	The maximize command.
MWM_FUNC_CLOSE	The Close command.

You logically OR them together just like the decoration values previously shown.

Detecting the Window Menu Close Command

If exiting your application might result in lost data, you should always confirm the user's intention to exit. For example, you might want to display an exit dialog when the user chooses File⇒Exit from the Project Window. The application is not terminated unless the OK button in the exit dialog is pressed.

However, an exit dialog alone is not sufficient. It does not protect the user from closing the window through the window manager. If the user chooses Close from the window menu, your application is immediately terminated unless you make these changes:

- Set the `DeleteResponse` shell property on your application's main window to `do_nothing`.
- Add a protocol callback that detects when the Close command is activated by the user.

Setting the Delete Response

The `VendorShell` widget property `DeleteResponse` determines what action the shell takes when it receives a `WM_DELETE_WINDOW` message from the window manager. This message is sent to a window whenever you choose the Close command from the window menu.

Setting the `DeleteResponse` to `do_nothing` ensures that the shell will not immediately destroy itself. The default value for this property is `destroy`.

Adding a Protocol Callback

You can detect when a particular message is sent to an application's window by setting the protocol callback. To establish a callback that is invoked when the Close command is activated on your application's main window, you must set up the callback to watch for a `WM_DELETE_WINDOW` message.

In the `draw_start.prj` example, the main window establishes a protocol callback by calling an auxiliary function named `CreateWindowManagerProtocols()`. This function is called in the interface's Final Code area as follows:

```
/* Final Code */

CreateWindowManagerProtocols(UxGetWidget(rtrn));
```

The local variable `rtrn` contains the shadow widget structure for the main window's application shell. The widget ID (type `Widget`) is passed to this function as a parameter, so the protocol callback can be established. Here is how the function is declared:

```
/* Auxiliary Functions */

void CreateWindowManagerProtocols(Widget shell)

{

    Atom xa_WM_DELETE_WINDOW;

    /* Intern the "delete window" atom.*/

    xa_WM_DELETE_WINDOW = XInternAtom
        (UxDisplay,"WM_DELETE_WINDOW",False);

    /* Add the window manager protocol callback.*/

    XmAddWMProtocolCallback (shell,
        xa_WM_DELETE_WINDOW,ExitCB, NULL);

}
```

The protocol callback list for the `WM_DELETE_WINDOW` message now includes a call to `ExitCB()`. This function is also entered into the Auxiliary Functions area:

```
/* This function pops up the Exit dialog box.*/

void ExitCB(Widget w, XtPointer
   client_data,XtPointer call_data)

{

   mainWS_Exit(UxWidgetToSwidget(w), &UxEnv);

}
```

The Exit command in the application’s File menu also calls the exit method to display the exit dialog. Therefore, once the protocol callback is established, the Close command in the window menu is functionally equivalent to the Exit command in the File menu.

Controlling Appearance 4

Overview

The concepts and techniques presented in this chapter show you how to control particular visual aspects of your application's user interface. Techniques for controlling layout using manager objects and constraint properties are also included.

As you work on your application's user interface, you should make note of the visual characteristics that should be customized by the end-user. For these characteristics—such as fonts and colors—you should provide acceptable default values, but not interfere with the user's ability to change them.

Note: Several examples are provided with the UIM/X software. Most have their own subdirectories containing one or more files needed to explore a particular topic. If you just want to explore the on-line examples, look in the *uimx_directory*/`contrib` directory.

Polishing the Layout

The interactive layout features of UIM/X help you to quickly overcome some of the most difficult aspects of programming with objects. Here are a few tips to help you get the most out of these layout features:

Avoiding Absolute Coordinates

When you create a new object and place it on the screen, UIM/X uses the box you draw to set the object's location and size properties (`X`, `Y`, `Width`, and `Height`). Initially these properties are set to Private, which means that the object may not be able to resize itself accordingly if it needs to.

If you are using a manager object, such as a bulletin board, that does not impose any geometry management on its children, you may want to reset some or all of each object's layout properties back to default.

Planning Ahead for End-User Customization

As you arrange the objects in an interface, you should keep in mind that many users may want to change some visual characteristics such as fonts. A change in font size could render your interface useless or impaired if it is unable to adjust to accommodate the font. You should also consider what happens to the layout of an interface if the user resizes it. For instance:

- Should some elements of the interface remain a constant size while others automatically adjust to fit the remaining space?
- Is there a minimum or a maximum useful size?
- Should the interface maintain a certain width-to-height ratio?
- Should the interface never be resized by the user?

The answers to such questions may help you choose particular shell and manager objects. You may also discover some properties that help implement the desired behavior.

Choosing the Right Manager Objects

Choosing the right manager object often simplifies your layout work. Before you begin constructing an interface, you should understand which manager objects are available, how they control the layout of their children, and which one best meets your needs. Refer to the *OSF/Motif Programmer's Guide* and *OSF/Motif Programmer's Reference Manual* for more information on manager objects.

If the children of the manager object are created dynamically while your application is running, you should place the manager object inside a Scrolled Window object. The Scrolled Window is used when a portion of an interface may be too large to display it in its entirety.

The Form object is perhaps the most useful object for controlling the layout of its children. The Form object uses *constraint* properties to create dependencies between objects and properties. Using forms, you can construct interfaces that respond as the user expects when they are resized.

The RowColumn object is designed to manage its children in rows and columns. This function makes it particularly useful for arranging collections of similar or identical objects. For example, a file manager application could use a RowColumn object to order file and directory icons.

The best way to manage resize behavior is to use the Constraint Editor. This chapter includes a discussion on how to use the Constraint Editor.

Customizing Colors and Fonts

Users, especially those familiar with the X Window System, expect to be able to customize the colors and fonts for each application they use. While building the interfaces for your application you should keep these recommendations in mind:

- If you specify any color or font property in your application's interface, be sure to make them Public. If you don't, end-users will not be able to customize these properties. Later, when you generate the source code for your interfaces, Public resource files are created. You can combine the properties in these files to make an application defaults file for your application.

- Carefully consider how the user's choice of a different font might affect the layout of the interface you designed. For example, if the user chooses a larger font, does the interface automatically make itself large enough? Here are some tips for working with objects and interfaces whose size may be determined by the current font:
    - Be sure the `RecomputeSize` property for Label objects (and subclasses of Label) is set to `true`. When set to `true`, this property causes the object to resize itself whenever any of its visual characteristics, including fonts, change.
    - Choose manager objects that respond positively when their children request a new size. The Form object is popular for this type of layout because it allows you to create layout relationships between its children. If you want the form to grow and shrink as needed, you should attach its children to the right and bottom edges (or another pair of adjacent objects) of the form.
    - Object properties that control an object's size and location, such as `Width`, `Height`, `X`, and `Y`, should be set to their default values. Use of the Bulletin Board object is generally discouraged because it relies on these properties to layout its children. As you create and place objects during construction, UIM/X sets these values to Private. Depending on how the object's managers use these properties, you may have to specifically set them to Default.
    - Set the `AllowShellResize` property for each top-level interface to `true.` This allows the manager object within the shell to specify a new window size.

If your application has special characteristics that depend on screen resolution or the availability of colors, you can query the X server to obtain a lot of valuable information. Using this information, your application could determine the best default sizes and colors for some of its interfaces.

Here are some of the Xlib functions that you can use to get information from the server:

```
BlackPixel()          DefaultDepth()
DefaultRootWindow()   DefaultVisual()
DisplayCells()        DisplayWidth()
DisplayWidthMM()      WhitePixel()
DisplayHeight()       DisplayHeightMM()
```

Using the Form Object

The Form object uses constraint properties which allow you to create layout dependencies among its children. To see how constraint properties are used to layout an interface, consider a form and its four children as shown in Figure 4-1:

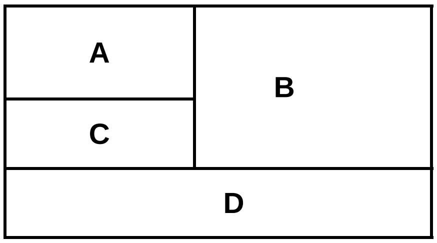


Figure 4-1 A Form With Children

The four objects are arranged on the form according to these design specifications:

- Object A should appear in the upper-left corner of the form and may specify
- its own size at creation. Object A's size should not change when the form is resized.
- Object B should fill the form to the right of object A. Object B's height is supposed to be 70% of the form's height.
- Object C should fill the rectangular space below object A and to the left of object B.
- Object D fills the bottom 30% of the form.

To construct this interface, you first create the form and its first child, object A. You attach A to the top and left edges of the form by setting two constraint properties:

TopAttachment:	`"attach_form"`
LeftAttachment:	`"attach_form"`

Next, you create object B. To meet the design specifications, you set constraints as follows:

TopAttachment:	`"attach_form"`
LeftAttachment:	`"attach_widget"`
LeftWidget:	`widgetA`
RightAttachment:	`"attach_form"`
BottomAttachment:	`"attach_position"`
BottomPosition:	`70`

These values constrain the width of object B to the width of the form minus the width of object A. The `BottomPosition` property is set to 70, which means that the bottom edge of object B should be 70% from the top of the form. Since the top of object B is attached to the form, object B’s height is always 70% of the form’s height.

To attach object C so that it fills the rectangle below object A and to the left of object B, you set constraints as follows:

TopAttachment:	`"attach_widget"`
TopWidget:	`widgetA`
LeftAttachment:	`"attach_form"`
RightAttachment:	`"attach_opposite_widget"`
RightWidget:	`widgetA`
BottomAttachment:	`"attach_opposite_widget"`
BottomWidget:	`widgetB`

You use `attach_opposite_widget` when you want an edge of an object to be flush with another. Here, the right edge of object C is made flush with the right edge of object A. Similarly, its bottom edge is made flush with object B.

Finally you set the constraints for object D as follows:

TopAttachment:	`"attach_widget"`
TopWidget:	`widgetB`
LeftAttachment:	`"attach_form"`
RightAttachment:	`"attach_form"`
BottomAttachment:	`"attach_form"`

Object D is attached to the sides and bottom of the form and to the bottom of object B.

Note: When setting form constraints for using an object's Property Editor, the default values provided by the form may be confusing. If *all* attachment properties are set to default, the default values for `TopAttachment` and `LeftAttachment` are `attach_form`, and `RightAttachment` and `BottomAttachment` default to `attach_none`. However, if you change the constraints for `RightAttachment` or `BottomAttachment`, the default values for `TopAttachment` and `LeftAttachment` become `attach_none`.

Guidelines for Form Constraints

Here are a few guidelines for using Form object constraints:

- Avoid attaching several objects to the same position. If your design calls for a particular position, attach one object to that position and then attach others to that object.

  In the example above you could attach the bottom of C and the top of D to the 70% position. However, if your design specifications for this position change (say to 60%), you will have to update the position properties for B, C, and D. Using the method shown, you have to change the position constraint only on object B—objects C and D automatically adjust because they are attached to object B.

- Use offsets to provide space between an object and whatever it is attached to. Offsets are specified using one or more of these constraint properties: `TopOffset`, `LeftOffset`, `RightOffset`, and `BottomOffset`.

- Consider how the form should respond if its children determine their sizes dynamically at creation time. As mentioned earlier, if you are constructing an interface that should determine its own size based on the current font or other visual properties, you should not attach objects to the right and bottom edges of the form. For the example in this section, that means that the `RightAttachment` properties of objects B and D, and the `BottomAttachment` property of object D should be left as their default values (`attach_none`).

Using the Constraint Editor

The Constraint Editor is a valuable tool that allows you to attach constraints to the objects in your interface. The Constraint Editor is shown in Figure 4-2. With the Constraint Editor’s easy-to-use graphical interface, you can selectively apply constraints that make your interface and the objects within it maintain proportion when they undergo a resize event.

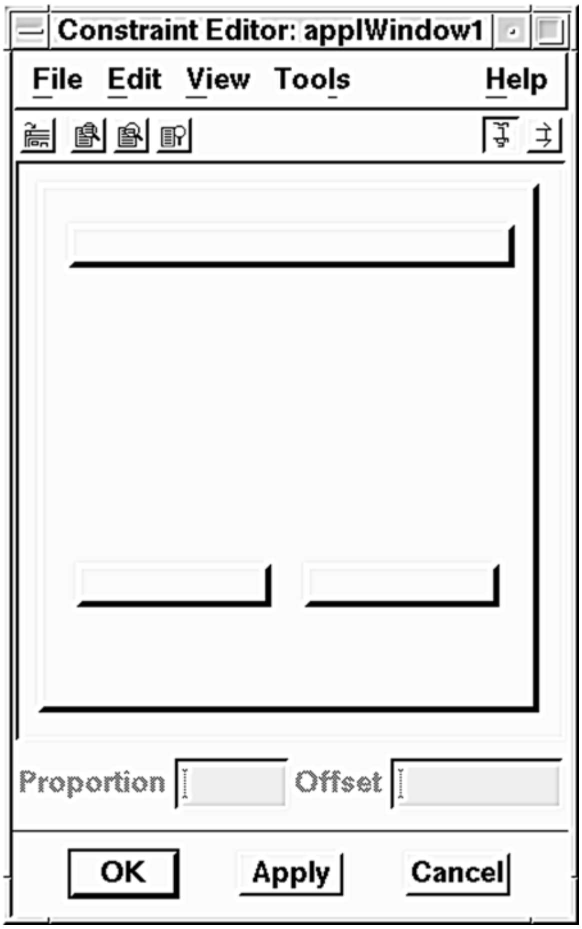


Figure 4-2 Constraint Editor

Opening the Constraint Editor

To open the Constraint Editor:

1. Select an object in your form.

2. Choose Tools⇒Constraint Editor from the Project Window, or choose Selected Objects⇒Tools⇒Constraint Editor.

   The Constraint Editor appears, displaying the selected form.

   To load a form into a Constraint Editor that is already showing, choose File⇒Load from the Constraint Editor, or click on the Load icon in the Constraint Editor's icon bar.

To Use the Zoom Feature

In order to better see your form in the Constraint Editor, you can zoom in, zoom out, or revert to the form's actual size.

To zoom in and magnify the size of your form, click on the Zoom In icon in the Constraint Editor's icon bar, or choose View⇒Zoom In from the Constraint Editor.

To zoom out and shrink the size of your form, click on the Zoom Out icon in the Constraint Editor's icon bar, or choose View⇒Zoom Out from the Constraint Editor.

To revert to the original size of your form, click on the Actual Size icon in the Constraint Editor's icon bar, or choose View⇒Actual Size from the Constraint Editor.

Specifying Constraints

With the Constraint Editor, you can set constraints between children on a form or between a child and its parent. The first step in setting a constraint is to select a tool. The tool you choose determines the type of constraint that will be applied to the object. The Constraint Editor lets you choose from three tools, as described below.

Bolt

The Bolt constraint allows you to anchor one object to another so that an absolute distance is maintained between the objects during subsequent move and resize operations.

To impose a Bolt constraint:

1. Click on the Bolt icon in the Constraint Editor's icon bar.

OR

1. Select Tools⇒Bolt from the Constraint Editor.
2. Position the pointer on an edge of the selected object, then press and hold the Select mouse button.

3. Drag the mouse pointer to a valid destination and release the mouse button.

The applied constraint is depicted graphically by a bolt symbol. The caption of the right-hand text field in the Constraint Editor changes from ‘‘Offset’’ to ‘‘Length.’’ The text field displays the length to be maintained between the two objects during move and resize operations.

To change the value of Length, type a new value into the text field and click on Apply. The graphics in the form area are updated to reflect the new value.

Note: The Bolt constraint maps to the `attach_form` and `attach_widget` properties. If you wish, you can load an object into the Property Editor, choose the Constraints category, and see how the Constraint properties change as you apply constraints with the Constraint Editor.

Dimension

The Dimension constraint allows you to anchor an object so that during subsequent move and resize operations, the position of the selected edge remains fixed at a proportionate distance from the left (y-axis) or bottom (x-axis) edge of the form.

Applying a Dimension constraint to the upper or lower edge of an object imposes a y-axis constraint. Applying a Dimension constraint to the left or right edge of an object imposes an x-axis constraint. Once such a constraint is applied, you cannot move an object from its position on that axis.

To impose a Dimension constraint:

1. Click on the Dimension icon in the Constraint Editor’s icon bar.

OR

1. Select Tools⇒Dimension from the Constraint Editor.
2. Position the pointer on an edge of the selected object, and press the Select mouse button.

The applied constraint is depicted graphically by an arrow, linking the selected object to the parent. The Proportion text field displays the percentage of the form (in the appropriate dimension) that is taken up by the Dimension arrow. The Offset text field displays the number of pixels to offset the edge from its base position.

To change the values of Proportion or Offset, type new values into their respective text fields and click Apply. The graphics in the form area are updated to reflect the new values.

Note: The Dimension constraint maps to the `attach_position` property.

PushPin

The Pushpin constraint allows you to constrain an object so that it maintains its absolute position along the x-axis, the y-axis, or both during subsequent resize operations.

To impose a Pushpin constraint:

1. Click on the Pushpin icon in the Constraint Editor's Icon Bar.

OR

1. Select Tools⇒Pushpin from the Constraint Editor.
2. Position the pointer on an edge of the selected object, and press the Select mouse button.

The applied constraint is depicted graphically by a pushpin symbol.

The Location text field displays the location on the x-axis or y-axis at which the selected object is anchored. To change the Location value, type a new value into the text field and click Apply. The graphics in the form area are updated to reflect the new value.

Note: The Pushpin constraint maps to the `attach_self` property.

Editing Constraints

The Constraint Editor allows you to easily select, deselect, and delete constraints, either one at a time or all at once. You can also recreate your form within the Constraint Editor.

Selecting Constraints

To select a single constraint, click on its symbol in the Constraint Editor. To select more than one constraint, hold down the Control key while clicking on their symbols. To select all of the constraints, choose Edit⇒Select All from the Constraint Editor, or choose Selected Objects⇒Select All.

Deselecting Constraints

To deselect constraints, choose Edit⇒Deselect All from the Constraint Editor, or choose Selected Objects⇒Deselect All in the Constraint Editor, or click somewhere other than the constraints' symbols, either in the actual interface, in its representation in the Constraint Editor, or in the Interfaces area of the Project Window.

Deleting Constraints

You can delete all previously applied constraints simultaneously, or you can delete individually selected constraints. To delete a constraint:

1. Select the constraint or constraints you wish to delete.
2. Choose Edit⇒Delete from the Constraint Editor.

OR

2. Choose Selected Objects⇒Delete in the Constraint Editor.

To delete all constraints simultaneously:

1. Choose Edit⇒Select All from the Constraint Editor.
2. Choose Edit⇒Delete from the Constraint Editor.

OR

2. Choose Selected Objects⇒Delete.

Recreating Your Form

To recreate your form with the Constraint Editor, choose Edit⇒Recreate Form from the Constraint Editor, or choose Selected Objects⇒Recreate Form in the Constraint Editor.

Handling Absolute Coordinates

The Constraints that you set with the Constraint Editor may conflict with the absolute coordinates specified by the `X`, `Y`, `Width`, and `Height` properties. To avoid any confusion, load the objects in your interface into the Property Editor, and set the Source option menu to Default for each of these properties.

Specifying Application Window Behavior

5

Overview

You customize the look and feel of your application by specifying the behavior of each application window in your project. For example, a mouse pointer in a text editor window will most likely behave differently from one in a drawing application window. In a text editor window, pressing a mouse button and dragging the mouse pointer might select text. In a drawing application window, those operations might draw an object. In UIM/X, you regulate application window behavior by assigning *translation tables*. You use translation tables to map events and event sequences into window-specific actions.

In UIM/X, working with translation tables is made easier with the support of a Translation Table Editor, an Action Table Editor, and an Event Editor. By maintaining separate action tables, you can assign the same actions to many widgets without typing the code over again.

Note: This chapter focuses on the use of translation tables to add behavior to application windows. Translation tables can also be used to override a widget's built-in behavior. For example, clicking on a pushbutton changes the graphical representation of the widget. Assigning a translation table to the pushbutton can alter that behavior as desired.

The procedure for specifying behavior to application windows is as follows:

- Create the translation table with events and actions as required.
- Assign the translation table to the application window using the Property Editor.

Action Tables and Events

In the Event Editor, you can graphically specify the events that will trigger a response in the application window. UIM/X's Event Editor lists common events, such as mouse button presses and releases. Nevertheless, you can choose from any X Toolkit event—not just those presented graphically.

Similarly, in the Action Table Editor you define the *response* to the event. The action's code can refer to the variable `UxThisWidget`, and to the standard four parameters of an action function. (These standard arguments are a widget, an event, an array of strings containing any arguments specified for the action in the translation table, and the number of arguments in the string.)

The Translation Table Editor

Connecting events and actions is done in a Translation Table Editor, after which you apply the translation table to an object. At this stage, you can choose from all three X Toolkit modes for modifying object behavior. The built-in object behavior can be augmented, overridden, or replaced by the new behavior you define. Once the translation table is installed on an object, the named action will be invoked each time the event sequence occurs in the object.

This chapter gives details on each of the main steps in creating a translation. A special section explains the code generated by UIM/X from your action function and how to pass parameters to your action function.

Defining the Events: Using the Event Editor

The UIM/X Event Editor (shown in Figure 5-1) allows you to define events interactively, for use in the Translation Table Editor. When you choose events via the Event Editor, the correct code is automatically presented in the Event String field. You can then transfer the event to the Translation Table Editor by applying the change.

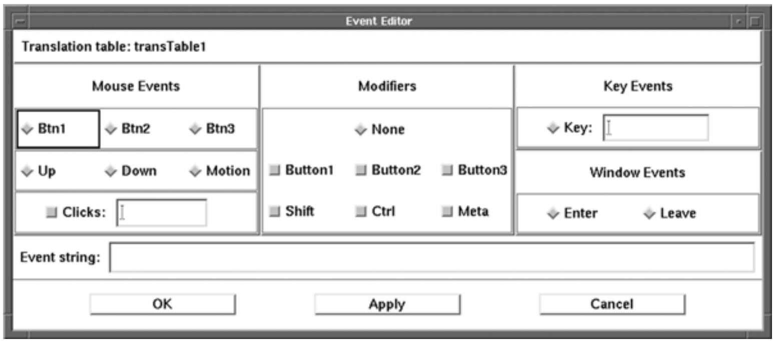


Figure 5-1 Event Editor

Opening the Event Editor

The Event Editor is opened via the Translation Table Editor. The Translation Table editor is opened via the Translation Table List.

To Open the Translation Table List

1. Select any object in your interface.
2. Choose Tools⇒Translation Table List from the Project Window.

OR

2. Choose Selected Objects⇒Tools⇒Translation Table List. The Translation Table List appears.

Alternatively, you can open the Translation Table List by selecting your interface.

1. Select your interface by clicking on it, or by clicking on its icon in the Interfaces Area of the Project Window.
2. Choose Tools⇒Translation Table List from the Project Window.

OR

2. Choose Selected Interfaces⇒Tools⇒Translation Table List. The Translation Table List appears, as shown in Figure 5-2.

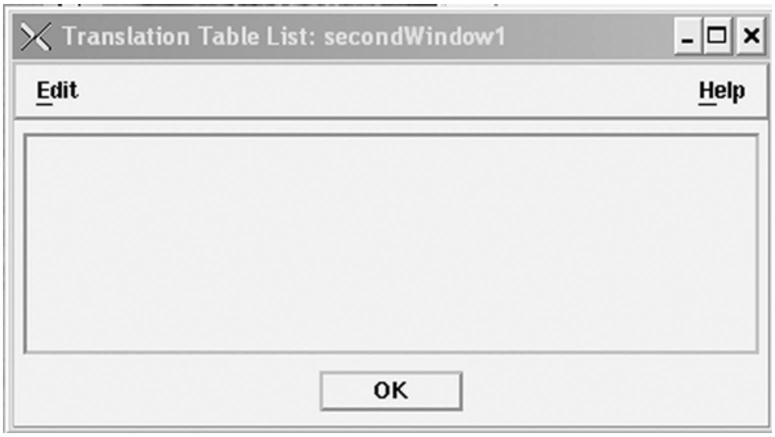


Figure 5-2 Translation Table List

To Open the Translation Table Editor for a New Translation Table

1. Choose Edit⇒Add from the Translation Table List.

OR

1. Choose Selected Entry⇒Add.

The Translation Table Editor appears, with a new translation table loaded, as shown in Figure 5-3.

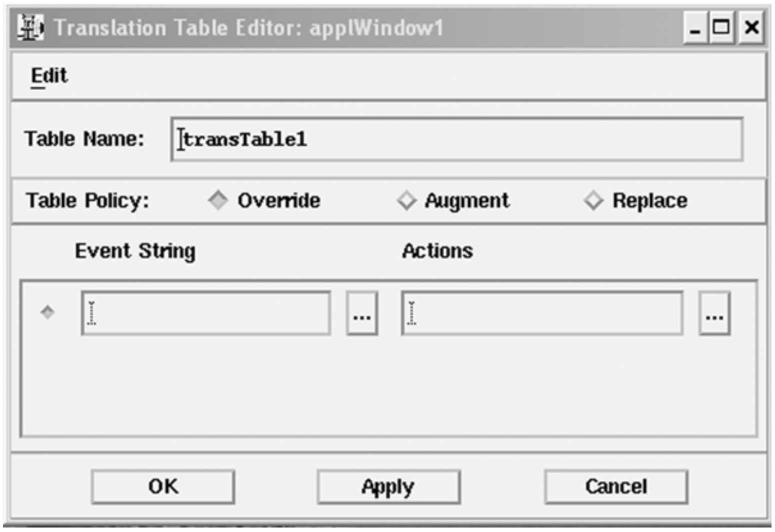


Figure 5-3 Translation Table Editor

To Open the Translation Table Editor for an Existing Translation Table

1. Double-click on the icon of the Translation Table you want to edit.

OR

1. In the Translation Table List, click on the icon of the Translation Table you want to edit.
2. Choose Edit⇒Editor from the Translation Table List.

OR

2. Choose Selected Entry⇒Editor in the Translation Table List. The Translation Table Editor appears, containing your chosen translation table.

Note: A complete description of the Translation Table Editor is given in “Connecting Events and Actions: Translation Tables” on page 74.

To Open the Event Editor

1. Choose Edit⇒Event Editor in the Translation Table Editor.

OR

1. Choose Selected Entry⇒Event Editor in the Translation Table Editor.

The Event Editor appears, loaded with your selected translation table.

Defining Events

Once the Event Editor is displayed you are ready to define an event.

To Define an Event

1. By clicking on its toggle button, select the Event String field on the Translation Table Editor where you want to define the event.

   Initially there will be only one Event String field, but you can add others. To do this, choose Edit⇒Add from the Translation Table Editor. An empty Event String and Action row appears.
2. In the Event Editor, use the Select mouse button to choose the mouse buttons, modifiers, keys, and window events that make up the event.

   As you click on the events, the correct code is automatically displayed in the Event String field of the Event Editor.

   UIM/X supports all X Toolkit events, not just those listed in the table. If the event you want to match is not listed, type it directly into the Event String area. You can also type it directly into the Event String area of the Translation Table Editor.
3. Apply the changes by clicking on OK or Apply in the Event Editor. The event will be displayed in the active Event String field of the Translation Table Editor.

Note: It is not possible to assign the event to the translation table itself until you have provided a corresponding action name in the Translation Table Editor.

Responding to Events: Defining the Actions

The Action Table Editor (shown in Figure 5-4) is where you define the action part of the event-action sequence. Here you define the action names that can be referred to in both the Translation Table Editor (where events and actions are linked) and the action’s code.

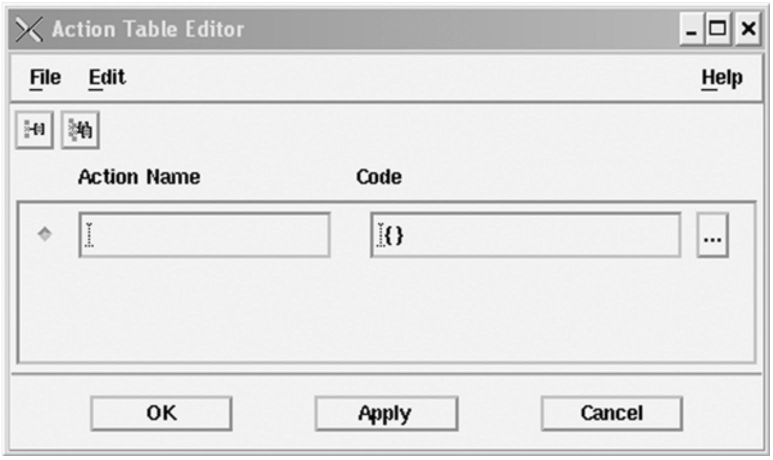


Figure 5-4 Action Table Editor

There is only one action table per project. The actions defined in it are available to all interfaces in the project.

Note that actions that are part of an augmented UIM/X are already defined and do not need to be re-defined in the Action Table Editor. They can be referred to directly in the Translation Table Editor. (Refer to the *X Toolkit Intrinsics Programming Manual* for details on the format of action functions.)

In addition to the above action type, there are four parameters declared implicitly by UIM/X for each action you define. You can refer to these in your action's code without re-declaring them. In addition, one of the parameters can be used as an argument, to pass strings into your action. Details on this are given in “Advanced Usage: Passing Parameters to Your Actions” on page 79.

Interface-specific variables cannot be referenced in the Action Table Editor. Actions should call a method or function using the action function arguments `UxThisWidget`, `UxWidget`, `UxEvent`, `UxParams`, and `UxNumParams`. (Note that `UxNumParams` is an `int`, not a pointer to an `int`. UIM/X dereferences the pointer passed in by Xt.)

Opening the Action Table Editor

To open the Action Table Editor choose Tools Action Table Editor from the Project Window.

Working with Actions

To Add an Action

1. Choose Edit⇒Add from the Action Table Editor.

OR

1. Choose Selected Entry⇒Add from the Action Table Editor.

   A new action row appears in the Action Table Editor and becomes the selected action.

To Select an Action

Move the pointer to the toggle on the left of the action and press the Select mouse button. To unselect that action, select another action.

To Duplicate an Action

1. Select the action you want to duplicate.
2. Choose Edit⇒Duplicate from the Action Table Editor.

OR

2. Choose Selected Entry⇒Duplicate from the Action Table Editor.

To Delete an Action

1. Select the action you want to delete.
2. Choose Edit⇒Delete from the Action Table Editor.

OR

2. Choose Selected Entry⇒Delete from the Action Table Editor.

To Create an Action

1. Enter the Action Name.
2. Enter the code. To access the Text Editor for code having more than one line, choose the Text Editor (…) button.
3. Apply the changes to the action table by clicking Apply or OK in the Action Table Editor.

Connecting Events and Actions: Translation Tables

The actions defined in the Action Table Editor can be used with any object in your application. To connect actions with the events that trigger them, and then use these actions with specific objects, you use translation tables.

Translation tables connect events to the actions defined in the Action Table Editor. Objects have a translation property that allow you to associate a translation table with a given object.

In UIM/X, each interface can have any number of translation tables. The Translation Table List (shown in Figure 5-5) manages the translation tables associated with an interface. The Translation Table List allows you to view, add, edit, and delete Translation Tables.

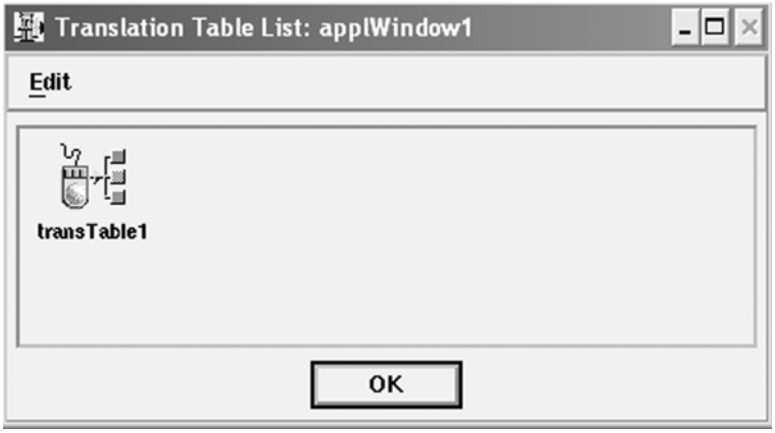


Figure 5-5 Translation Table List

Working with the Translation Table List

To learn how to open the Translation Table List, see “To Open the Translation Table List” on page 69.

To Add a Translation Table

1. Choose Edit⇒Add from the Translation Table List.

OR

1. Choose Selected Entry⇒Add from the Translation Table List.

An icon representing the translation table appears on the Translation Table List. The editor for that translation table is automatically displayed.

To Select a Translation Table

1. Position the pointer on the corresponding icon in the Translation Table List.
2. Click the Select mouse button.
3. To select additional translation tables, hold down the Control key while clicking the Select mouse button on the icons.

To Duplicate a Translation Table

1. Select the translation table you want to duplicate by clicking on it.
2. Choose Edit⇒Duplicate from the Translation Table List.

OR

2. Choose Selected Entry⇒Duplicate from the Translation Table List.

To Delete a Translation Table

1. Select the translation table you want to delete by clicking on it.
2. Choose Edit⇒Delete from the Translation Table List.

OR

2. Choose Selected Entry⇒Delete from the Translation Table List.

Working with the Translation Table Editor

To learn how to open the Translation Table Editor, see "To Open the Translation Table Editor for a new Translation Table" on page 70.

To Add a Translation

1. Choose Edit⇒Add from the Translation Table Editor.

OR

1. Choose Selected Entry⇒Add from the Translation Table Editor.

A new translation row appears in the Translation Table Editor, and is automatically selected.

To Select a Translation

Move the pointer to the toggle on the left of the Event String and press the Select mouse button. To unselect that translation, select another Event String.

To Duplicate a Translation

1. Select the translation to be duplicated by clicking the Select mouse button on the toggle beside the Event String.
2. Choose Edit⇒Duplicate from the Translation Table Editor.

OR

2. Choose Selected Entry⇒Duplicate from the Translation Table Editor.

A copy of the selected translation is added to the Translation Table Editor. It is automatically selected.

To Delete a Translation

1. Select the translation to be deleted by clicking the Select mouse button on the toggle beside the Event String.
2. Choose Edit⇒Delete from the Translation Table Editor.

OR

2. Choose Selected Entry⇒Delete from the Translation Table Editor.

The translation is removed from the Translation Table Editor.

Specifying the Table Policy

Objects respond to certain events automatically because they have translations built-in. For example, clicking on a pushbutton automatically changes the pushbutton's shading. When you assign a new translation to an object, you must decide what policy it will have regarding an object's already existing translations.

The Translation Table Editor supports all three X Toolkit policies: override, augment, and replace. Override replaces the object's translations with those you define in the Translation Table Editor. That is, when the event is one to which the object already responds, the new action replaces the old one. Override also adds new translations to the object when there is no conflict. Augment adds new translations *only* when there is no conflict with the object's already existing translations. If the object already responds to an event, that response remains in force. Replace removes all built-in translations and replaces them with those you define in the Translation Table Editor.

To change the table policy, use the Select mouse button to press the appropriate toggle in the Table Policy area of the Translation Table Editor.

Events for Accelerator Tables

When building events for Accelerator Tables, note that the X Toolkit doesn't allow event abbreviations. Instead of `<Btn1Down>`, for example, you must type the long form of the event, `<ButtonPress>Button1`.

For more details on the long forms of events, refer to X Toolkit programming manuals.

Attaching the Translation Table to a Widget

Before an object can make use of the events and actions you define in a Translation Table Editor, you must attach the table to it. Translation tables can be attached to an object in the Property Editor, or dynamically using `UxPutTranslations()`.

Note that many objects in an interface can share the same translation table. Each object, however, must have the translation table attached to it.

Attaching a Translation Table in the Property Editor

To Attach a Translation Table Using the Property Editor

1. Bring up the object's Property Editor.
2. Each object has a Core property called Translations. Display this property and type the translation table name in the field without quotation marks.
3. Apply the change to the object.

Note: If the Translations property is set to Public, augment and override modes in the Translation may not work properly. If translations have been specified in a resource file, the object does not install the default translations. This is due to a bug in Motif. Note also that Public resources may not be visible, depending on the setting of the Hide Source toggle in the View menu of the Property Editor.

Dynamically Attaching a Translation Table: UxPutTranslations

It is often necessary to dynamically change the behavior of an object. For example, in a drawing program you may want to draw lines one moment, then choose a palette entry and draw rectangles. This would mean changing the application window's translation table so the same mouse event—pressing and dragging—would produce a different result.

You can assign a translation table to an object dynamically using `UxPutTranslations()`. For example, the callbacks for palette entries for lines and rectangles respectively might contain the following C code:

```
{
   /* Draw lines. */
   UxPutTranslations(drawingWindow,
      line_trans_table);
}
{
   /* Draw rectangles. */
   UxPutTranslations(drawingWindow,rect_trans_tab
      le);
}
```

Create the translation table as you normally would.

Note: When assigning translation tables dynamically using `UxPutTranslations()`, take care to ensure you get the behavior you expect. If your new table has its policy set to *augment*, your new translations may not get assigned. Ensure the new behavior is assigned to the object by *overriding* the previous translations.

Advanced Usage: Passing Parameters to Your Actions

When you call an action in the Translation Table Editor, UIM/X allows you to pass strings to the action function. To understand why parameters can be passed, it is first necessary to understand the action function generated by UIM/X from the code you type in. In addition to the automatically declared parameter used to pass in strings, there are three other parameters available for use in your action code. Finally, there is also the variable `UxThisWidget`, that your code can refer to without explicitly defining it.

The Action Function Generated by UIM/X

When you type action code between the curly braces in the Action Table Editor, UIM/X creates a function. For example, suppose your interface is called `drawingArea1`. UIM/X generates a function as follows:

```
static void action_actionName(Widget UxWidget,
   XEvent *UxEvent, String *UxParams, int
   *p_UxNumParams)

{

   int UxNumParams=*p_UxNumParams;

   _UxCdrawingArea1 *UxSaveCtx, *UxContext;

   swidget UxThisWidget;

   UxThisWidget = UxWidgetToSwidget(UxWidget);

   UxSaveCtx = UxDrawingArea1Context;

   UxDrawingArea1Context = UxContext =

      (_UxCdrawingArea1 *)
         UxGetContext(UxThisWidget);

   /* Code entered in the Action Table Editor */

   {UxPutBackGround(UxThisWidget, "red");} /*
```

```
         action code*/
      UxDrawingArea1Context = UxSaveCtx;
   }
```

As you can see, each action has four arguments. These are `UxWidget`, `UxEvent`, `UxParams`, and `p_UxNumParams`. These are the four standard X Toolkit arguments for an action. `UxWidget` is the X widget for which the action was called. `UxEvent` is the event that triggers the action. `UxParams` is an array of strings. It is this array that holds the parameters you pass to the action. `p_UxNumParams` is a pointer to the number of elements in the array. UIM/X dereferences the pointer and stores the actual value in `UxNumParams`.

You can use `UxWidget`, `UxEvent`, `UxParams`, and `UxNumParams` in the code you write in the Action Table Editor without re-declaring them. In addition, (as in callback functions) the variable `UxThisWidget` can be used. As shown in the sample, it identifies the swidget associated with `UxWidget`, for which the action was called.

Adding Arguments to Action Calls

To use parameters when making an action call, append the parameter list to the name of the action in the Translation Table Editor. The argument list must be a series of words separated by commas, typed between the parentheses that follow the action. UIM/X takes these arguments and passes them into the action in the third argument, `UxParams`, which is an a array of strings—like `argv` in a C main function. At the same time it updates `UxNumParams` to reflect the number of arguments passed in—analogous to `argc`.

As the arguments are passed as strings, it is not possible to pass variables to an Action. However, you can pass numerical values, since a numerical value can be retrieved from a string.

Example

Consider an action called `act1`, whose body is declared in the Action Table Editor as follows:

```
{
   int   i;
   for (i=0; i < UxNumParams; i++)
   {
   printf("Parameter %d = %s\n", i, UxParams[i]);
   }
}
```

As noted above, neither `UxNumParams`nor `UxParams`are declared explicitly in the Action Table Editor. However, as UIM/X declares them automatically, you can use them in action code without re-declaring them.

In the Translation Table Editor, this action could be called as follows:

```
act1("Greetings, Leif!", "3.14")
```

When an event triggers `act1`, the following is printed to `stdout` (in Test mode, it is printed to the Project Window Messages Area):

```
Parameter 0 = Greetings, Leif!
Parameter 1 = 3.14
```

Adding Interfaces to Existing Applications

6

Overview

Using UIM/X, you can create GUIs for already existing applications. For command-line applications, you can create an interface that communicates directly with the application: it is not necessary to modify the command-line application. For applications where some restructuring is possible, another technique can be used. This chapter discusses both these techniques.

Adding an Interface to a Command-Line Application

The technique of creating a GUI for a command-line driven application is known as subprocess control.

Note: To use subprocess control, add `#include "UxSubproc.h"` to the Includes, Defines, Global Variables section of the Declarations Editor.

Functions For Controlling Subprocesses

UIM/X provides a number of functions for controlling the execution of the application, passing command-line strings to the application, and handling output from the application.

Function	**Use**
UxCreateSubproc()	Specifies an application to be executed as a subprocess of UIM/X.
UxRunSubproc()	Begins execution of the subprocess. If the subprocess is already running, it returns an error.
UxExecSubproc()	Begins execution of the subprocess. If the subprocess is already running, it terminates the running subprocess and restarts it.
UxSetSubprocClosure()	Specifies data that is to be passed to the output handler function for a given subprocess. It identifies where the output is sent.
UxSetSubprocEcho()	Turns echoing of input on or off for a subprocess.
UxSetSubprocFunction()	Specifies the function that handles output from the subprocess.
UxSetSubprocExitCallback()	Specifies a function to be called when the subprocess is terminated or stopped.
UxSendSubproc()	Sends a command string to the subprocess.
UxExitSubproc()	Terminates execution of a subprocess.
UxDeleteSubproc()	Terminates execution of a subprocess, and deletes all related data.
UxGetSubprocPid()	Determines the process id of a running subprocess.

Function	Use
UxAppendTo()	Allows subprocesses to write to a Text object.
UxTransferToBuffer()	Copies a 2048 byte block of output from a subprocess to a buffer, returning a pointer to the buffer.

Command-Line Example: Database Application

The following examples use an imaginary database application to demonstrate UIM/X subprocess features. Suppose that the imaginary application accesses a database of countries and their populations and that its executable is named `population_db`. Further, suppose that the application accepts commands from a terminal and prints information back to the terminal. It is, therefore, eligible for treatment using the UIM/X subprocess features.

Suppose that the application accepts the following commands:

Command	Response
`pop` *name of country*	Returns the population of a country.
quit	Exits.

There are two basic scenarios for handling output from the application: output can be sent directly to a Text object or to a text buffer for processing.

Sending Output to a Text Object:

In the first scenario, output is to be sent to a Text object. Consider the interface in Figure 6-1.

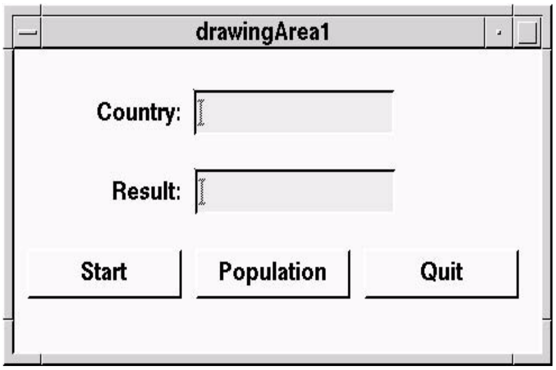


Figure 6-1 Text Object Interface Sketch

The user presses the Start button to initialize the application and the Quit button to exit. To find the population of a country, the name of the country is entered in the `text1` object and the Population button is pressed. The result appears in the `text2` object. The following paragraphs describe how to implement this interface.

Step 1: Initializing the Application

1. The following code goes in the Include, Defines, Global Variables section of the Declaration Editor:

```
#include "UxSubproc.h"
handle h;
```

2. To initialize the application, the following code would be entered in the `ActivateCallback` property for the Start Push Button:

```
h = UxCreateSubproc("population_db", NULL,
   UxAppendTo);

if (ERROR == UxSetSubprocClosure
   (h,UxGetWidget(text2))) {printf("Cannot set
   subproc closure\n");return;

}

if (ERROR == UxRunSubproc(h, NULL)) {

   printf("Cannot start the application\n");

   return;

}
```

The variable h is used by UIM/X to identify the subprocess. The UxCreateSubproc() function call initializes UIM/X's mechanism for handling the subprocess, but does not start the subprocess itself. The first argument is the name of the application. The second argument is the default arguments that are to be passed to the application. Because this application requires no arguments, the value of the second argument of UxCreateSubproc() is NULL. The third argument is a function pointer to a function which handles output from the subprocess. In this case, the output is to be sent directly to a Text object. The Ux Convenience Library function UxAppendTo() can be used for this purpose.

The second function call, `UxSetSubprocClosure()`, identifies the Text object to which `UxAppendTo()` will send output from the subprocess. The first argument is the handle of the subprocess returned by `UxCreateSubproc()`, and the second argument is the X widget pointer of the Text object.

The third function call, `UxRunSubproc()`, starts the subprocess. Its arguments are the subprocess handle and the arguments to pass to the subprocess. If the second argument is `NULL`, the default arguments specified in the call to `UxCreateSubproc()` are passed as arguments to the subprocess.

The function `UxExecSubproc()` can be used in place of `UxRunSubproc()`. The difference is that if the subprocess is already running, `UxRunSubproc()` returns an error; `UxExecSubproc()` first terminates and then restarts the subprocess.

Note: You can create multiple subprocesses, but if you don't need to, it is recommended to create a single subprocess by calling `UxCreateSubproc()` then `UxRunSubproc()`. When the child subprocess terminates, it is enough to run the subprocess again using `UxRunSubproc()`. It is up to you to manage each spawned subprocess. To clean up the subprocess structure, use `UxDeleteSubproc()`.

If you wish to have the subprocess started when the interface is first created, you would enter the code above in the Final Code area of the Declaration Editor for the interface. In this case, a Start button would not be needed.

Step 2: Implementing the Population Command

To implement the population command, you would enter the following code in the `ActivateCallback` property of the Population button:

```
char s[128];

sprintf(s, "pop %s", UxGetText(text1));

UxSendSubproc(h, s);
```

The `UxSendSubproc()` function is used to send a command string to the running subprocess. In this case, the string sent is formed by the command `pop` followed by the name of the country, which is read from the Text object, `text1`. Output would be handled by `UxAppendTo()`, which would display the string in the Text object, `text2`.

Final Step: Implementing the Quit Command

To implement the quit command, you would enter the following code in the ActivateCallback property of the Quit button:

```
UxDeleteSubproc(h);
```

For applications that might be terminated and restarted many times from the same interface, `UxExitSubproc()` should be called instead. In this case, only a call to `UxRunSubproc()` is needed to restart it.

Sending Output to a Text Buffer

In the second scenario, the output is sent to a text buffer for processing. Consider the interface shown in Figure 6-2.

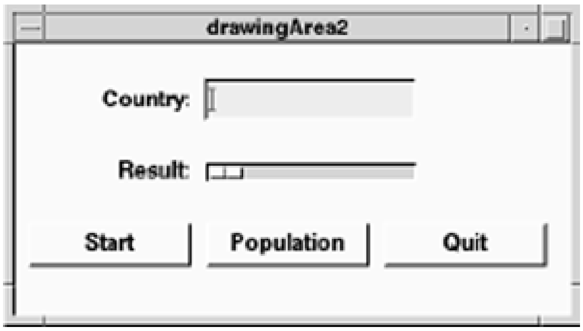


Figure 6-2 Text Buffer Interface Sketch

A Horizontal Scale object displays the population value graphically. It replaces the Text object that displayed the population in the previous example. This example also uses a method called `_set_ScaleValue` that sets the value of the Horizontal Scale object. Otherwise, the interfaces are identical.

1. The following code goes in the Include, Defines, Global Variables section of the Declaration Editor:

   ```
   #include "UxSubproc.h"
   handle h;
   ```

2. For this case, it is necessary to write a customized output handler to replace `UxAppendTo()`. This function would normally be entered into the Auxiliary Functions section of the Declarations Editor and would have the form:

   ```
   void LaurasOutputHandler(int fd, char *data)

   /* fd is the file descriptor for output from the
      subprocess */

   /* data is set by UxSetSubprocClosure()      */

   {

      int status;

      char *s;

      s = UxTransferToBuffer(fd, &status);

      drawingArea2_set_ScaleValue((swidget) data,
   ```

```
            atoi(s)/100, &UxEnv);
    }
```

The function `UxTransferToBuffer()` copies a block of 2048 bytes of output from the subprocess to a buffer and returns a pointer to it. In this case, we would convert the result to integer and use the interface's `_set_ScaleValue` method to set the position of the indicator in the Horizontal Scale object. The status and data parameters are described later.

3. The code for the Start Push Button's `ActivateCallback` property would be:

```
handle h;

h = UxCreateSubproc("population_db",
   NULL,LaurasOutputHandler);

UxSetSubprocClosure(h,(char *)drawingArea2);

if (ERROR == UxRunSubproc(h, NULL))

{
   printf("Cannot start the application\n");
   return;

}
```

The third argument to `UxCreateSubproc` now specifies the custom output handler `LaurasOutputHandler()` rather than `UxAppendTo()`. In addition, the call to `UxSetSubprocClosure()` passes in the interface, enabling it to be used to call the _set_ScaleValue method, as seen in Step 2. This function is used to set the second argument, which is passed to `LaurasOutputHandler()` each time it is called. A typical application would be to distinguish among multiple processes running simultaneously, all using the same output handler.

The `UxSendSubproc()` and `UxDeleteSubproc()` calls are handled the same way in the two scenarios.

The `status` variable is non-zero (`1`) if there is more output data present. It is up to your output handler function to read from the buffer until `status` is `0`.

The function `UxSetSubprocFunction()` can be used to modify the output handler function for a running subprocess without having to terminate it.

Determining When the Subprocess Exits or is Stopped

1. The function `UxSetSubprocExitCallback()` is provided to handle cases when the subprocess exits or is stopped. The developer must write an exit handler, which takes two integer parameters: the process id of the exiting subprocess and a status parameter. This function is entered into the Auxiliary Functions section of the Declarations Editor. A typical exit callback is:

```
void my_exit_callback(int pid, int status, handle
sub_handle)

{

   printf("subprocess %d terminated with exit code
      %d\n",pid, (status & 0xffff) >> 8);

}
```

The status parameter will contain the value set by the `wait()` system call. The example is rather primitive and does not handle cases when the subprocess is stopped or killed.

2. To set the exit callback function, use `UxSetSubprocExitCallback()` after creating but before running the subprocess:

```
handle h;

h = UxCreateSubproc("population_db", NULL,
   UxAppendTo);

if (ERROR == UxSetSubprocClosure(h,
   UxGetWidget(text2))) {printf("Cannot set
   subproc closure\n");return;

}

if (ERROR == UxSetSubprocExitCallback

   (h, my_exit_callback)) {

   printf ("Cannot set subproc exit callback\n");

   return;

}

if (ERROR == UxRunSubproc(h, NULL)) {

   printf("Cannot start the
   application\n");return;

}
```

Note: `UxCreateSubproc()` creates two queues on the system. These queues are automatically cleared when the subprocess terminates. If the process can't run, or if it terminates abnormally, the queues may still be present.

To print information about the currently active queues, run the command `ipcs`. To remove the queues after an abnormal termination, use the command `ipcrm -q` *id*, where *id* is the id of the queue (given by `ipcs`.)

Intermediate Restructuring of an Existing Application

When it is possible to modify the code, but the effort to restructure it for the asynchronous event-loop style of programming is too great, UIM/X provides an intermediate mechanism to add an iconic interface to an application.

The typical situation is one where deep in existing code there is an I/O call to read input (for example, a color) and the process does not continue until input is received. UIM/X provides two function calls: `UxWaitForNotify()` and `UxNotify()` to handle such cases. Here, you would create an interface—a color editor, for example—and modify the Interface Function to return a color. In addition, you would place a call to `UxWaitForNotify()` after the `UxPopupInterface()` call in the Final Code section of the Declarations Editor and before the `return` statement. In the OK button callback of the color editor, put a call to `UxNotify()`. Finally, the original I/O call is replaced by a call to the Interface Function.

When the Interface Function is called, the color editor pops up and UIM/X enters the `UxWaitForNotify()` call. `UxWaitForNotify()` processes events so the user can use the interface to choose a color. However it does not return until `UxNotify()` is called when the OK button is clicked. Control then returns to the application. The interface is still modal, but development cost to create a usable product is minimal.

Generating and Compiling Project Code 7

Overview

When you use UIM/X objects to build a project, you can generate ANSI C, K&R C, or C++ code. For a list of supported compilers, refer to the *UIM/X Installation Guide*.

This chapter discusses the files that are created when you generate code for your project, and explains how to use the Program Layout Editor. It shows you how to manage your files, and provides a breakdown of the structure of the generated code.

Generating Resource Files

When an interface contains a property where the Source is set to Public, UIM/X creates a resource file when generating the interface's code file (The name of the file is that of the code file, with the extension `.rf`). This file can be edited by the end user, and is automatically merged with the resource database when the interface is created.

By setting properties in an interface to Public, you allow end-users to customize the application. By modifying the resulting resource file, end-users can customize the appearance and behavior of the generated application.

Generating Message Catalogs

By default, message strings entered in the Property Editor appear directly in the generated code. For example, if you set the LabelString property of a Label, the text is hard-coded in a call to `UxPutLabelString()`:

```
UxPutLabelString( label, "Label:" )
```

When you generate a message catalog for a project, the values of properties such as LabelString are put in the message catalog. UIM/X generates message catalog entries for all Private properties of type `XmString`, `XmString*`, `String`, or `String*`.

Generating a message catalog separates message text from program code, and makes it easier to localize the application. When you use a message catalog, UIM/X replaces references to message text with the macro `UxCATGETS()`.

```
UxPutLabelString( label,UxCATGETS(
    UXMC_ROWCOLUMN1, 1, UXDS_ROWCOLUMN1_1 )),
```

The macro `UxCATGETS()` retrieves the message from the project message catalog. (See "Retrieving Messages" on page 100 for more information on the `UxCATGETS()` macro.) The generated main program file takes care of opening and closing the project message catalog.

The following table describes the different files used to build the project message catalog.

File	Description
Project`.mk`	The makefile generated by UIM/X. It contains the rules for building the project message catalog from the message text source files generated by UIM/X.
Project`.cat`	The message catalog for the project.
Project`_cat.h`	Defines the message set constants used as the first argument to `UxCATGETS()`. In the message text source file generated for an interface, a constant is used for the message set number: $set UXMC_*INTERFACE* The makefile generates a `#define` for each message set constant and puts it in the *Project*`_cat.h` file. The *Project*`_cat.h` file is included by the interface source and header files.
Interface`.msg`	The message text source file for an interface. Contains one message set. This file is generated by UIM/X. The makefile builds the project message catalog from the interface `.msg` files.
Interface`_ds.h`	Defines constants for the default strings. This file is generated by UIM/X and is included by the interface source and header files.

Adding Messages to the Project Catalog

You can use the Program Layout Editor to add your own messages to the project message catalog. In the Ux (or Xt) Makefile, use the macro `APPL_MSG_FILES` to list the message files you want to add to the project message catalog:

```
APPL_MSG_FILES =

MSG_FILES = $(INTERFACES:$PJ_SOURCE_SUFFIX =
    .msg)$(APPL_MSG_FILES)

MSG_FILES_STRIP = $(INTERFACES:$PJ_SOURCE_SUFFIX
    =)$(APPL_MSG_FILES:.msg=)

MSG_CATALOG = $(EXECUTABLE).cat

MSG_HEADER = $(EXECUTABLE)_cat.h

MSG_CPP = $PJ_CATALOG_CPP

MSG_DEPEND = $PJ_CATALOG
```

When you run the makefile, your message files will be merged into the project message catalog.

Note: Your message files must follow the X/Open standard for message files. As well, they must use symbolic constants for message set numbers:

```
$quote"
$set MyMSGSET
1 "Message: "
...
```

The makefile replaces the symbolic constant `MyMSGSET` with a unique integer when it builds the project message catalog. The makefile also generates a `#define` for `MyMSGSET` in the *Project*`_cat.h` file. This allows you to use the symbolic constant to retrieve messages.

The makefile replaces the symbolic constant `MyMSGSET` with a unique integer when it builds the project message catalog. The makefile also generates a `#define` for `MyMSGSET` in the *Project*`_cat.h` file. This allows you to use the symbolic constant to retrieve messages.

Retrieving Messages

To retrieve messages from your own message files, you use the macro `UxCATGETS()`. In UIM/X, this macro always returns the default string:

```
    #define UxCATGETS( setId, msgId, ds ) (ds)
```

The first argument to `UxCATGETS()` is the message set number. This should be the message set constant you put in your message text source file (for example, `MyMSGSET`). The second argument is the message number, and the third argument is the default string.

As long as you use the correct message set constant and the correct message number, `UxCATGETS()` will retrieve the proper message at run time:

```
    UxCATGETS( MyMSGSET, 1, "MyDefaultString" )
```

It is not generally a good idea to use `UxCATGETS()` to retrieve messages generated by UIM/X. As you edit interfaces and set properties, the message numbers may change each time you generate the interface message files.

The run time versions of `UxCATGETS()` are defined in *uimx_directory*`/include/UxLib.h` (for code that uses the Ux Convenience Library) and *uimx_directory*`/config/UxXt.h` (for Xt code).

At run time, `UxCATGETS()` retrieves the message from the catalog:

```
#ifdef UX_CATALOG
#define UxCATGETS( setId, msgId, ds ) \c
atgets( UxMsgCatalog, (setId), (msgId), (ds) )
#else
#define UxCATGETS( setId, msgId, ds ) (ds)
#endif /* UX_CATALOG */
```

The constant `UX_CATALOG` is defined by the generated makefile using the `-D` compiler option. This constant controls whether or not a generated application uses a project message catalog.

`UxMsgCatalog` is the catalog descriptor for the project message catalog. It is set in the main program file when the project message catalog is opened:

```
nl_catd UxMsgCatalog;
UxMsgCatalog = UxCATOPEN( UX_CATALOG_NAME, 0 );
```

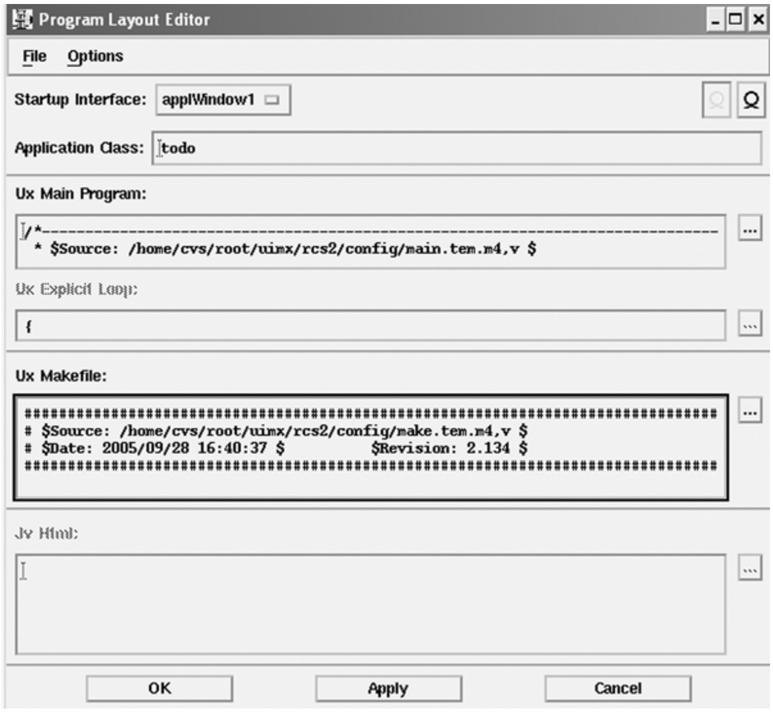
The macro `UxCATOPEN()` expands to a call to `catopen()`. The constant `UX_CATALOG_NAME` is the name of the project catalog file. It is defined in the makefile by appending `_cat.h` to the name of the executable.

In UIM/X, a project consists of one or more interfaces and, in some cases, one or more palettes. Interfaces may contain Action Tables and Translation Tables. In addition, when you generate project code, you get a main program file, a makefile, a project options file, a message catalog, and resource files for each interface.

You use the Program Layout Editor to customize the main program file and the makefile. The Property Editor controls the contents of the generated message catalog and resource files.

Using the Program Layout Editor

The Program Layout Editor, shown in Figure 7-1, allows you to modify the main program file, the explicit event loop, and the makefile automatically generated when the code is written for a project.



Startup Interface: applWindow1

Application Class: todo

```
/*---------------------------------------------------------------------------
 * $Source: /home/cvs/root/uimx/rcs2/config/main.tem.m4,v $
```

```
{
```

```
###################################################################################
# $Source: /home/cvs/root/uimx/rcs2/config/make.tem.m4,v $
# $Date: 2005/09/28 16:40:37 $         $Revision: 2.134 $
###################################################################################
```

Figure 7-1 The Program Layout Editor

To Use the Program Layout Editor

1. Open the Program Layout Editor by choosing Tools⇒Program Layout Editor from the Project Window.
2. Enter the application class name in the corresponding text field. The application class name is recognized as the name of the application resource file in which you establish application-wide resource settings. UIM/X automatically uses the application class name specified here in the `XtAppInitialize()` and `UxAppInitialize()` functions in the main program file.

   The default application class name is the name of the project.
3. To change the start-up interface, select the interface icon on the Project Window representing the desired start-up interface, then choose File Load Startup Interface from the Program Layout Editor.

4. To change the Main Program File, click on the corresponding Text Editor (…) button to access the Text Editor. When finished adding code at the appropriate places, click on the Text Editor’s OK button.

Note: *Do not* modify any sections preceded by `$`. UIM/X automatically replaces these sections with the appropriate code when the main program is generated.

---

5. To change the loop type, choose Options⇒Loop⇒Explicit Loop or Options⇒Loop⇒Implicit Loop from the Program Layout Editor. If you choose Explicit Loop, the Ux Explicit Loop area becomes sensitive.
6. To add code to the Explicit Loop, or to the Makefile, click on the appropriated Text Editor (…) button. When finished, click on the Text Editor’s OK button.
7. Click on the Apply button on the Program Layout Editor.

You can now automatically generate code and the corresponding main program file and makefile for the project by choosing File⇒Generate Project Code from the Project Window.

Customizing a Main Program File

The code generated by UIM/X contains, for every interface, an interface function to create and possibly pop-up the interface. UIM/X generates the function name by prepending `create_` or `popup_` to the name of the top-level widget. The Interface Function can be either a `create` or a `popup` function. You specify one or the other through the Interface Function Type selection of

the Options menu in the Project Window. Both create the interface—a `popup` function additionally pops up the interface.

However, to make an executable from the generated C code, a main function is needed. This main function performs various initialization tasks and then calls the Interface Function of the application’s start-up interface. If the Interface Function is a `create` function, the main function includes a call to `VisualInterface_Manage()` to pop up the interface. Finally, the main function starts an event loop.

Note: If your Interface Function is a `popup` function and you edit the final code and remove the call to `VisualInterface_Manage()`, the Interface Function is still defined as a `popup` function, and the final code is assumed to be responsible for popping up the interface. As a result, the interface will not appear.

UIM/X automatically generates a main program file for your application. This main function has the following form:

```
#ifdef XOPEN_CATALOG
#include <locale.h>
#endif
#include <UxLib.h>
#include <X11/Xlib.h>
#ifdef _NO_PROTO
int main(argc, argv)
   int argc;
   char *argv[];
#else
int main(int argc, char *argv[])
#endif
{
   swidget mainIface;
   swidget create_bulletinBoard1(swidget _V_UxParent);
   swidget UxParent = NULL;
#ifdef XOPEN_CATALOG
   setlocale(LC_ALL, "");
   if (XSupportsLocale()) {
   XtSetLanguageProc(NULL, (XtLanguageProc) NULL,
      NULL);
   }
#endif
   (void) UxInitCat();
```

```
    UxTopLevel = XtAppInitialize(&UxAppContext,
       app_class_name, NULL, 0,
       &argc, argv, NULL, NULL, 0);
    UxAppInitialize(app_class_name, &argc, argv);
    mainIface = create_bulletinBoard1(UxParent);
    VisualInterface_Manage(mainIface, &UxEnv);
    UxMainLoop();
    /*NOTREACHED*/
    return 0;
}
```

Here it is assumed that the Interface Function is a `create` function, and that it is called `create_bulletinBoard1()`. The call to `UxInitCat()` initializes the message catalog. The call to `XtAppInitialize()` initializes the X Toolkit and returns `UxTopLevel`, which is the Widget ID of the root widget. `UxAppInitialize()` performs application-specific initialization. Next, the interface is created by calling `create_bulletinBoard1()` and then popped up by `VisualInterface_Manage()`. Finally, `UxMainLoop()` enters the X event loop. It is similar to `XtMainLoop()`, but correctly handles the freeing of the context structure when an interface is deleted.

Alternatively, the following code can be used instead of `UxMainLoop()`:

```
for (;;)
{
    XEvent event;
    UxNextEvent(&event);
    switch (event.type)
    {
/*---------------------------------------------*/
/* Insert code to handle any events that you do */
/* not wish to be handled by the interface     */
/*---------------------------------------------*/
    default:
            UxDispatchEvent(&event);
            break;
        }
    }
```

`UxNextEvent()` obtains the next event from the event queue. Depending on the type of the event, you can insert code to have the application handle the event, or pass it to one interface by calling `UxDispatchEvent()`.

Adding New Input Sources to the Event Loop

You can add new sources of input to the Xt event loop using the procedure `XtAppAddInput()`:

```
XtAppAddInput (XtAppContext app_context, int
   source, XtPointer condition,
   XtInputCallbackProc proc, XtPointer
   client_data)
```

The parameter `source` is an open file descriptor that can be any source or sink of data, such as a regular file or a socket. The parameter `condition` specifies the condition for which an event should be generated: when `source` has pending input, pending output, or a pending exception. Whenever that condition arises, the callback `proc` is called.

`XtAppAddInput()` may be called at anytime after `UxAppInitialize()` in your application program.

In the procedure `proc`, non-blocking I/O should be used to read from or write to `source`. This ensures that the procedure reads the data available from source without blocking to wait for more, so the application can return quickly to the main processing loop. Otherwise, the processing of window system events will be impaired, making your application less responsive.

`XtAppAddInput()` and the procedure type `XtInputCallbackProc` are documented in the *X Toolkit Intrinsics Reference Manual*. For discussion of blocking and non-blocking I/O and the meaning of the event conditions, the `select` system call is a good place to start.

There is another way to input sources to your application. You can use an explicit event loop and add your own input checks to the loop. For example, the following code will work, but at a high price:

```
/* Bad example of multiple input handling */
for (;;)
{
   if (XtPending())
   {
      UxNextEvent(&event);
      UxDispatchEvent(&event);
   }
   else if (YourApplicationPending())
      YourProcessInput();
}
```

This code is expensive because your application will cycle through this loop testing for input when it should be idle. At the application level of event processing, it is difficult to handle multiple input sources without creating such a busy wait situation.

To avoid a busy wait, you can read one of the input sources (usually the X event stream) with a blocking I/O call, servicing the others between events on the first:

```
/* Another bad example of multiple input
handling*/
for (;;)
{
   UxNextEvent(&event);
   UxDispatchEvent(&event);
   if (YourApplicationPending())
      YourProcessInput();
}
```

This is also unsatisfactory because the application input sources are only processed while there is Xt event activity.

The preferred way to add input sources to an application, without busy waiting and without any input stream blocking others, is to use `XtAppAddInput()`.

Managing Resource Files

When you generate code, UIM/X generates a resource file for each interface with Public properties. In the generated code, UIM/X adds a call to the Ux Convenience Library function `UxLoadResources()`. This function ensures that the resource file (of the same name as the interface) is merged with the database.

When generating Xt code UIM/X also adds a call to `UxLoadResources()`. However, in Xt code, `UxLoadResources()` is a stub function in `UxXt.c`, and you must fill in the body. If you do not, the `.rf` file will not be read, and the Public resources are ignored.

While UIM/X generates a separate resource file for each interface in the project, you may want to merge these resource files. By combining resource files you conveniently limit the number of places end-users have to go to modify widget properties.

For example, you may want to merge the resource files together and store them in the `app-defaults` directory. Additionally giving the file the application class name ensures the resource values will be merged with the resource database at program initialization.

If you store the file under a different name, or in a different directory, you will have to modify the search path.

The X-Compliant Default Search Path

When a generated application runs, it calls `XtAppInitialize()`, which causes the X resource manager to search a predefined path for resource definitions. When several definitions of the same resources are found, the later definitions override the earlier. The sources searched for resource definitions are the following (in the order presented):

1. A file with the same name as the application, in the directory `/usr/lib/X11/app-defaults.`
2. Files in the directory given by the environment variable `XAPPLRESDIR`, or, if the variable is not set, in the end-user's home directory, with the name *class*, where *class* is the class name of the application.
3. Resources loaded into the resource database manager. If no resources have been loaded this way, the resource manager looks for a `.Xdefaults` file in the end-user's home directory.

4. A file specified by the shell environment variable `XENVIRONMENT`.

   If this variable is not defined, the resource manager looks for a file named `.Xdefaults-`*hostname* in the end-user's home directory. Here, *hostname* is the name of the host where the application is running.
5. Any values specified on the command line using `-xrm`.
6. Finally, values specified on the command-line (other than with the `-xrm`). These final values will override those specified by resource defaults, regardless of their source.

Modifying the Search Path

If you move the resource files, you may need to modify the search path. A predefined variable `UxResourcePath` (of type `pathlist)` is an ordered list containing the search path used in locating resource files:

1. `/usr/lib/X11/app-defaults`
2. `/usr/lib/X11/app-defaults/`*AppClassName*
3. `/usr/lib/X11/app-defaults/`*AppClassName*`/`*screentype*`/`
4. `/usr/lib/X11/app-defaults/`*AppClassName*`/`*screentype*`/`*resolution*
5. If `XAPPLRESDIR` is set:
   - `$XAPPLRESDIR/`*AppClassName*
   - `$XAPPLRESDIR/`*AppClassName*`/`*screentype*
   - `$XAPPLRESDIR/`*AppClassName*`/`*screentype*`/`*resolution*

   Else:
   - `$HOME/`*AppClassName*
   - `$HOME/`*AppClassName*`/`*screentype*
   - `$HOME/`*AppClassName*`/`*screentype*`/`*resolution*

For compatibility with previous versions of UIM/X, `UxResourcePath` also includes the following paths:

- `./`*AppClassName*`/`
- `./`*AppClassName*`/`*color*`/`
- `./`*AppClassName*`/`*color*`/`*resolution*

UIM/X itself does not use `UxResourcePath`. You can use this path list in a generated application to load a resource file that is not in the X-compliant default search path:

```
UxLoadResources( UxExpandResourceFilename(
    filename ) );
```

The call to `UxExpandResourceFilename()` uses `UxResourcePath` to expand a file name into a full path name. By default, `UxLoadResources()` only looks in the current directory for the file.

The Ux Convenience Library provides a set of functions (see below) for working with path lists such as `UxResourcePath`.

Ux Convenience Library and Resource Files

The Ux Convenience Library contains a number of functions for dealing with resource files. These functions allow you to modify the search path, obtain the value of a resource, and load resource files into the resource database.

The following tables provide brief descriptions of each function—for details, refer to the UIM/X Reference manual.

Modifying the Search Path

When you write code to modify the search path, use the Program Layout Editor to add your code to the main program.

The path list `UxResourcePath` is initialized by `UxAppInitialize()`. If you want to append paths to the default list, add your code after the call to `UxAppInitialize()`. Adding paths to `UxResourcePath` before `UxAppInitialize()` is called overrides the default search paths.

Function	Description
UxFileExists()	Checks if a file exists in the current path.
UxInitPath()	Initializes a search path.
UxAddPath()	Adds a directory to the search path.
UxResetPath()	Replaces a search path.
UxFreePath()	Frees the memory related to a search path.
UxGetPath()	Returns the search path.
UxExpandFilename()	Given a file name, finds the first occurrence of the file in the search path and returns the file name prefixed by its directory path.
UxExpandEnv()	Expands all environment variables in a string.

Function	Description
UxExpandResourceFilename()	Given a file name, finds the first occurrence of the file in the `UxResourcePath()` search path and returns the file name prefixed by its directory path.

Obtaining Resource Values

Use the following functions to obtain the default value of a resource:

Function	Description
UxGetResource()	Gets the value of a resource, given the program and resource name. Returns `NULL` if the resource is not found.
UxGetDefault()	Same as `UxGetResource()`, except you also supply a default value that is returned if the resource is not found.
UxGetAppResource()	Gets the value of a resource, given the resource name and returns `NULL` if the resource is not found. The program name is taken as the first argument to `UxInitialize()`.
UxGetAppDefault()	Same as `UxGetAppResource()`, except you also supply a default value that is returned if the resource is not found.

Setting Resource Values

Use the following functions to merge or overwrite values into the resource database:

Function	Description
UxLoadResources()	Merges a resource file into the database. Does not overwrite values of the same name.
UxOverrideResources()	Loads a resource file into the database. Overwrites values of the same name.

Generating Code

The final step in creating a user interface for an application is to generate the code. UIM/X allows you to generate code for the entire project or for selected interfaces.

When generating code, UIM/X may also generate resource files and include files. This section describes the roles of these files and the circumstances under which they are produced.

UIM/X uses a utility program, called `uxcgen`, to generate interface code from interface files.

UIM/X also includes a contributed utility script that generates project code from a project file. You can find this utility in *uimx_directory*`/contrib/PrjGen`.

Generating Code for a Project

When you generate code for a project, UIM/X generates code for every interface in your project. If any widget's properties have been set to Public, UIM/X automatically generates a resource file for the interface in which the widget resides. All Translation Table and Action Table code related to the project will also be generated.

Code generation conforms to the options you specify through the Project Window's Options menu. There, you choose K&R C, ANSI C, or C++, instruct UIM/X to also generate UIL code, or instruct UIM/X to make the Ux Convenience Library or the Ux Convenience Library C++ Bindings accessible.

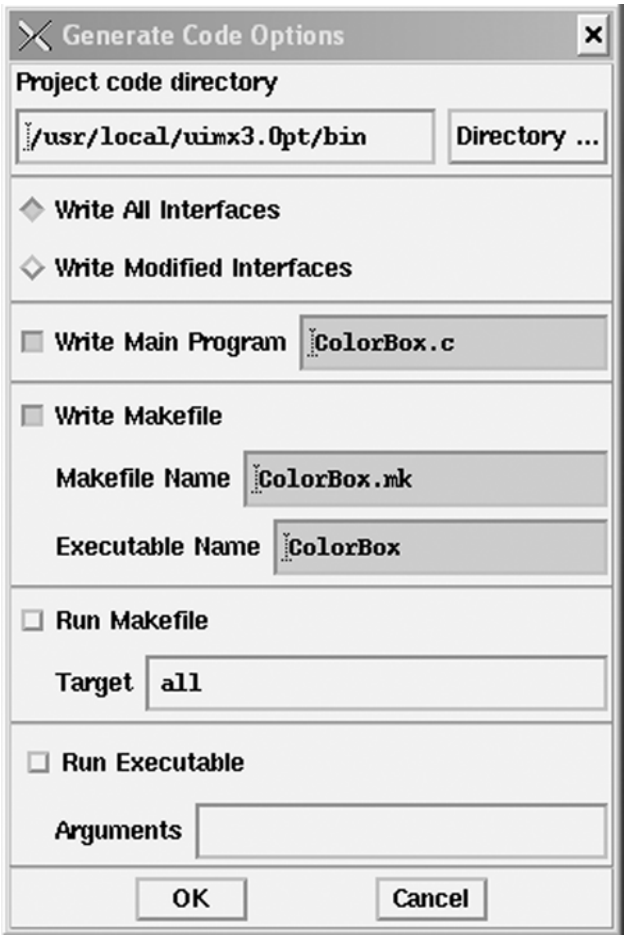
You also instruct UIM/X to generate Include files, message catalogs, and to support multiple copies of the same interface. You can also specify the suffixes for source files and header files.

Code generation also conforms to any instructions specified through the Program Layout Editor. You can indicate the application class, the start-up interface, and whether the event loop is to be explicit or implicit. You can also specify application-specific modifications to be incorporated in the main program file or in the makefile when these are generated.

To Generate Project Code

1. Choose File⇒Generate Project Code As from the Project Window.

   The Generate Code Options dialog box appears, as shown in Figure 7-2.



Generate Code Options

Project code directory: /usr/local/uimx3.0pt/bin

Write All Interfaces

Write Modified Interfaces

Write Main Program: ColorBox.c

Write Makefile

Makefile Name: ColorBox.mk

Executable Name: ColorBox

Run Makefile

Target: all

Run Executable

Arguments:

Figure 7-2 Generate Code Options Dialog Box

2. Set the desired options.
3. Click on OK to generate the code, or on Cancel to cancel code generation.

Quick Code Generation for Projects

If you have already generated code for your project, you might prefer to choose File⇒Generate Project Code from the Project Window.

Generate Project Code performs the same functions but does not present the Generate Code Options dialog box if the code generation for the project has already taken place. All files are automatically written under their current names and all options currently in force are respected.

However, if you select Generate Project Code when generating code for a project for the first time, the Generate Code Options dialog is presented.

Another method of quick code generation is to use Run Mode. Simply click on the Run icon in the Project Window’s icon bar, or choose Mode Run from the Project Window. Your code is generated and the compiled program runs on your screen.

Generating Code for Selected Interfaces

UIM/X also allows you to generate code for specific interfaces. You can choose any interface icon or icons in the Project Window. Code is generated for those interfaces only. No main program file or makefile is generated.

To Generate Code for Selected Interfaces in the Project

1. In the Project Window, select the interface icons that represent the interfaces for which code is to be generated.
2. Press and hold the Menu mouse button. The Selected Interfaces popup menu appears.
3. Choose Selected Interfaces⇒Generate Code As.
4. A File Selection box appears. Here, you can override the default filename offered for the interface file and indicate the directory in which the file is to be stored.

   If you do change the file name and neglect to specify a suffix, for example: `bulletinBoard`, UIM/X automatically appends to the filename the suffix specified in the Code Generation Options.
5. Click on OK. To cancel this function, click on Cancel.

   After you click OK or Cancel, UIM/X displays the File Selection box for the next interface you selected.

Quick Code Generation for Selected Interfaces

If you have already generated code for selected interfaces, you might prefer to choose Selected Interfaces⇒Generate Code.

Generate Code performs the same function as Generate Code As, but does not present the File Selection box unless you are generating code for the first time for the selected interfaces.

However, if you select Generate Code when you are generating code for the first time for any interface, the File Selection box is presented.

The Structure of the Generated Code

The generated file has a number of components, many of which you can define or modify via the Declaration Editor. The standard parts of a generated source file are listed below, in the order in which they are generated.

- Header files/Includes
- Global Variables
- Context Structure
- Translations
- Auxiliary Code
- Actions
- Callbacks
- Utility functions
- Interface Function

In a source file, the context structure enables UIM/X to determine the correct interface to act upon when a callback function or action function is called.

The standard parts of a generated C++ source file are similar, except that there is no context structure, because interface specific variables become members of the class.

Generating Header Files

Setting the Include File toggle in the Code Generation Options dialog causes UIM/X to generate an include file for each interface in the project. A generated include file contains the definition of the interface’s context structure. It also contains declarations of any method calls associated with the interface. When you generate an include file, the generated code file will `#include` it. Otherwise the code file contains an explicit definition of the context structure.

Using Include Files to Query Properties

You can use an include file to query an interface's properties from outside the interface. Consider an interface called `bulletinBoard1`, containing a Toggle Button called `toggleButton1`. Any of the following approaches can be used to determine (from outside the interface), if the Toggle Button is set:

- Defining a method for the Toggle Button's interface. This approach is the most robust, as it will function when there are several copies of the interface.
- Monitoring the state of the Toggle Button in the Auxiliary Code section of the interface.
- Setting a global variable each time the button is toggled.
- Accessing the Toggle Button directly.

Each of these approaches is discussed below.

Using Methods

One approach uses methods. First, define a method on `bulletinBoard1`:

```
int bulletinBoard1_toggleButton1IsSet(swidget
   UxThis, Environment *pEnv)
{
return strcmp(UxGetSet(toggleButton1), "true") ==
   0;
}
```

When `bulletinBoard1` is created, save the return value from the popup or create function:

```
swidget save_bulletinBoard1;
save_bulletinBoard1 = popup_bulletinBoard1();
```

To query the state of `toggleButton1`, call the method:

```
toggleButtonState =
   bulletinBoard1_toggleButtonIsSet(save_bulletin
      Board1,&UxEnv);
```

Note: Auxiliary functions cannot access data members in C++.

Monitoring the State of the Toggle Button in Auxiliary Functions

Another approach is to put a function in the Auxiliary Functions section of the Declaration Editor for the interface. This function gets the state of the Toggle Button:

```
int toggleButton1IsSet()
{
   return (strcmp(UxGetSet( toggleButton1 ),
      "true") == 0);
}
```

Setting a Global Variable

The third solution is to define a variable—`toggleButton1_state`, for example—and set it whenever the `toggleButton1`'s select or release callback is executed. For example, the `ArmCallback` might be:

```
{ extern int toggleButton1_state;
   toggleButton1_state = 1; }
```

Accessing the Toggle Directly

A fourth approach is to set `toggleButton1`'s `Name` property to Global in the Declaration properties of the Property Editor. You can then declare it as an external variable in other files, and query its state using `UxGetSet`():

```
extern swidget toggleButton1;

toggleButton1_state =
   (strcmp(UxGetSet(toggleButton1),"true") == 0);
```

Development Environment and Executable Code Differences

When using a development tool such as UIM/X, it is inevitable that certain differences will exist between the development environment and compiled code. These differences are briefly discussed below.

The Main Event Loop

UIM/X does not use the same event loop as the executable code. UIM/X has to steal and process events in Design mode. In Test mode all events are passed to the widgets.

Translations

In Design mode, if a Translation Table is being edited and has been put in the Translations property of a widget, when the Translation Table is modified, the widget will be recreated so that it reflects the new Translation Table. This doesn't happen in compiled code.

Multiple Copies of an Interface

Multiple copies of an interface are always supported in Test mode. In compiled code, you can choose whether or not to support multiple copies of an interface by setting the Context Support toggle in the Code Generation Options dialog.

Error Checking

During development, the Ux Convenience Library performs substantial error checking and recovery. The version compiled without `DESIGN_TIME` set does not and has much higher performance. Within the makefiles meant to generate an augmented version of UIM/X, `DESIGN_TIME` must be defined in order to compile and link your interfaces.

UxDestroyInterface() and UxDestroySwidget()

`UxDestroyInterface()` does not destroy interactively created interfaces during development—it unmaps them. `UxDestroySwidget()` pops up a confirmation dialog and does nothing.

X Toolkit Grabs

It is extremely dangerous to permit X Toolkit grabs during development. Most grabs such as `UxPopupInterface(I, exclusive_grab)` and `dialog_system_modal` are ignored.

abort(), assert(), exec(), exit(), fork(), wait()

These calls are trapped and a dialog appears.

Public Resources

At design time, properties set to Public are set using an `XtVaSetValues()` call rather than being merged into the resource database.

Problems with Your Compiled Application

The following topics describe some problems that you may encounter after compiling an application interface that has been built with UIM/X.

Differences Between Test Mode and Compiled Application

The two most common reasons for differences between an interface's behavior in Test mode and its behavior as a compiled application are:

- During development, your application's interface inherits property values from the Application Defaults.

  Your compiled application will probably reference a different resource file with different settings. That resource file is referenced by the `XtAppInitialize()` and the `UxAppInitialize()` functions in the Main Program File.

- During development, Shell widgets may not keep track of their size if you use the window manager to resize an interface window. To avoid this problem, resize interface shells by choosing Selected Widgets⇒Other⇒Resize.

A Debuggable UIM/X Library

If you suspect that a problem with some UIM/X-generated code is caused by the Ux Convenience Library, you may want to test your application using a debuggable library. This requires that you have the source code to the Ux Convenience Library.

exclusive_grab and nonexclusive_grab

In Test mode, the values `exclusive_grab` and `nonexclusive_grab` are mapped to `no_grab`. This prevents you from locking up your computer. To test `exclusive_grab`, you must generate and compile the source code for the interface.

Object Property Values A

Overview

The Property Editor is used to set and modify the values of properties for the objects of an interface. For efficient use, you need to know what values are admissible for each property. In many cases, this is clear from the value initially shown for a property. For example: the `BorderWidth` property accepts an `integer` value. In some cases, however, properties accept a limited number of pre-assigned strings. For example: the `EditMode` property accepts either `"single_line_edit"` or `"multi_line_edit"`.

This appendix outlines the range of values possible for each property. These are the values that can be used in the Property Editor.

They are also the values that can be used in the `UxPut` functions, which you may wish to include in callback or action code, or in the Declaration Editor.

Further, they are the values returned by the `UxGet` functions. In general, a value returned by a `UxGet` function will be an `integer` or a `char *`. In the case of a `char *`, a `strcmp` is sometimes necessary to determine which of the several possibilities is being returned. There are no `UxPut` or `UxGet` functions for callbacks.

Both the `UxPut` and `UxGet` functions are described in the *UIM/X Reference Manual*.

The following table lists all properties and their types. Properties of type `string` should be enclosed in quotes when entered in the Property Editor. Certain properties are of type `color`; colors can be specified as follows:

1. You may enter any color name defined in `/usr/lib/X11/rgb.txt,` for example, `"blue"`.
2. You may enter the `rgb` components of the color using three, six, nine, or twelve hexadecimal digits. In this case, the digits are preceded by a `#` character and enclosed in quotes. Each component is specified by means of one, two, three, or four digits respectively, for example, `"#00f"`, `"#0000ff"`,`"#000000fff"` and `"#00000000ffff"`.

Object Property	**Values**
Accelerator	string
AcceleratorText	string
Accelerators	translation table name
ActivateCallback	code
AdjustLast	“true”, “false”
Adjust Margin	“true”, “false”
Alignment	“alignment_center”, “alignment_end”, “alignment_beginning”
AllowOverlap	“true”, “false”
AllowResize	“true”, “false”
AllowShellResize	“true”, “false”
AncestorSensitive	“true”, “false”
ApplyCallback	code
ApplyLabelString	string
Argc	integer
Argv	char *
ArmCallback	code
ArmColor	color
ArmPixmap	bitmap or pixmap filename enclosed in quotes
ArrowDirection	“arrow_up”, “arrow_down”, “arrow_left”, “arrow_right”
AudibleWarning	“bell”, “none”
AutoShowCursorPosition	“true”, “false”
AutoUnmanage	“true”, “false”
AutomaticSelection	“true”, “false”
Background	color
BackgroundPixmap	filename of a bitmap file enclosed in quotes
BaseHeight	integer
BaseWidth	integer
BlinkRate	integer
BorderColor	color

Object Property	Values
BorderPixmap	filename of a bitmap file enclosed in quotes
BorderWidth	short
BottomAttachment	"attach_none", "attach_form", "attach_opposite_form", "attach_widget", "attach_opposite_widget", "attach_position", "attach_self"
BottomOffset	integer
BottomPosition	integer
BottomShadowColor	color
BottomShadowPixmap	bitmap or pixmap filename enclosed in quotes
BottomWidget	widget name enclosed in quotes
BrowseSelectionCallback	code
ButtonFontList	font name enclosed in quotes
CanBeTopLevel	"true", "false"
CancelButton	widget name enclosed in quotes
Cancelcallback	code
CancelLabelString	string
CanHaveChildren	"true", "false"
CascadePixmap	bitmap or pixmap filename enclosed in quotes
CascadingCallback	code
ChildHorizontalAlignment	"alignment_center", "alignment_end", "alignment_beginning"
ChildHorizontalSpacing	integer
ChildPlacement	"place_above_selection", "place_below"selection", "place_top"
Children	widget name enclosed in quotes
ChildType	"frame_title_child", "frame_workarea_child", "frame_generic_child"
ChildVerticalAlignment	"alignment_baseline_bottom", "alignment_baseline_top", "alignment_widget_top", "alignment_center", "alignment_widget_bottom"
ClipboardOps	"true", "false"
Colormap	string
Columns	integer
Command	string
CommandChangedCallback	code

Object Property	Values
CommandEnteredCallback	code
CommandWindowLocation	“command_above_workspace”, “command_below_workspace”
CompoundEditorName	char *
CompoundIcon	bitmap or pixmap filename enclose in quotes
CompoundName	widget name enclosed in quotes
CompoundResourceSet	char *
CompoundSwidgetMethodSet	char *
CreateCallback	code
CreateManaged	“true”, “false”
CreatePopupChildProc	code
CursorPosition	integer
CursorPositionVisible	“true”, “false”
DecimalPoints	short
DecrementCallback	code
DefaultActionCallback	code
DefaultButton	widget name enclose in quotes
DefaultButtonShadowThickness	integer
DefaultButtonType	“dialog_cancel_button”, “dialog_ok_button”, “dialog_help_button”
DefaultFontList	font name enclosed in quotes
DefaultPosition	“true”, “false”
DeleteResponse	string
Depth	integer
DestroyCallback	code
DialogStyle	“dialog_system_modal”, “dialog_primary_application_modal”, “dialog_modeless”, “dialog_work_area”, “dialog_full_application_modal”
DialogTitle	string
DialogType	“dialog_command”, “dialog_file_selection”, “dialog_prompt”, “dialog_selection”, “dialog_work_area”
Directory	string
DirectoryValid	“true”, “false”
DirListItemCount	integer
DirListItems	string
DirListLabelString	string
DirMask	string
DirSearchProc	proc

Object Property	Values
DirSpec	string
DisarmCallback	code
DoubleClickInterval	integer
DragCallback	code
DragRecursion	“none”, “up”
EditMode	“single_line_edit”, “multi_line_edit”
Editable	“true”, “false”
Editor	code
EntryAlignment	“alignment_beginning”, “alignment_center”, “alignment_end”
EntryBorder	short
EntryCallback	code
EntryClass	string
EntryVerticalAlignment	“alignment_baseline_buttom”, “alignment_baseline_top”, “alignment_contents_bottom”, “alignment_center”, “alignment_contents_top”
ExposeCallback	code
ExtendedSelectionCallback	code
FileListItemCount	integer
FileListItems	string
FileListLabelString	string
FileSearchProc	char *
FileTypeMask	char *
FillOnArm	“true”, “false”
FillOnSelect	“true”, “false”
FilterLabelString	string
FocusCallback	code
FontList	font name enclosed in quotes
Foreground	color
FractionBase	integer
GainPrimaryCallback	callback list
Geometry	string
Height	short
HeightInc	integer
HelpCallback	code
HelpLabelString	string
HighlightColor	color
HighlightOnEnter	“true”, “false”
Highlight Pixmap	bitmap or pixmap filename enclosed in quotes
HighlightThickness	short

Object Property	Values
HistoryItemCount	integer
HistoryItems	string
HistoryMaxItems	integer
HistoryVisibleItemCount	integer
HorizontalScrollBar	widget name enclosed in quotes
HorizontalSpacing	short
IconMask	string
IconName	string
IconNameEncoding	atom
IconPixmap	string
IconWindow	string
IconX	integer
IconY	integer
Iconic	“true”, “false”
Increment	integer
IncrementCallback	code
IndicatorOn	“true”, “false”
IndicatorSize	short
IndicatorType	“one_of_many”, “n_of_many”
InitialResourcesPersistent	“true”, “false”
InitialDelay	integer
InitialFocus	string
InitialState	integer
Input	“true”, “false”
InputCallback	code
InputMethod	string
InsertPosition	code
IsAlignable	“true”, “false”
IsAligned	“true”, “false”
IsAreaSelectable	“true”, “false”
IsArrangeable	“true”, “false”
IsCompound	“true”, “false”
IsDeletable	“true”, “false”
IsDraggable	“true”, “false”
IsDuplicatable	“true”, “false”
IsHomogeneous	“true”, “false”
IsInCompound	“true”, “false”
IsMovable	“true”, “false”
IsNovice	“true”, “false”
IsRecreatable	“true”, “false”
IsRegion	“true”, “false”
IsReorderable	“true”, “false”
IsReparentable	“true”, “false”
IsResizable	“true”, “false”
IsSelectable	“true”, “false”

Object Property	**Values**
ItemCount	`integer`
Items	`string`
KeyboardFocusPolicy	`string`
LabelFontList	`font name enclosed in quotes`
LabelInsensitivePixmap	`bitmap or pixmap filename enclosed in quotes`
LabelPixmap	`bitmap or pixmap filename enclosed in quotes`
LabelString	`string`
LabelType	`“string”, “pixmap”`
LeftAttachment	`“attach_none”, “attach_form”, “attach_opposite_form”, “attach_widget”, “attach_opposite_widget”, “attach_position”, “attach_self”`
LeftOffset	`integer`
LeftPosition	`integer`
LeftWidget	`widget name enclose in quotes`
ListItemCount	`integer`
ListItems	`string`
ListLabelString	`string`
ListMarginHeight	`short`
ListMarginWidth	`short`
ListSizePolicy	`“constant”, “variable”, “resize_if_possible”`
ListSpacing	`short`
ListUpdated	`“true”, “false”`
ListVisibleItemCount	`integer`
LosePrimaryCallback	`callback list`
LosingFocusCallback	`code`
MainWindowMarginHeight	`short`
MainWindowMarginWidth	`short`
MapCallback	`code`
MappedWhenManaged	`“true”, “false”`
MappingDelay	`integer`
Margin	`short`
MarginBottom	`short`
MarginHeight	`short`
MarginLeft	`short`
MarginRight	`short`
MarginTop	`short`
MarginWidth	`short`
MaxAspectX	`integer`
MaxAspectY	`integer`

Object Property	Values
MaxHeight	integer
MaxLength	integer
MaxWidth	integer
Maximum	integer
MenuAccelerator	“true”, “false”
MenuHelpWidget	widget name enclosed in quotes
MenuHistory	widget name enclosed in quotes
MenuPost	string
MessageAlignment	“alignment_beginning”, “alignment_center”, “alignmnet_end”
MessageString	string
MessageWindow	widget name enclosed in quotes
MinAspectX	integer
MinAspectY	integer
MinHeight	integer
MinWidth	integer
MinimizeButtons	“true”, “false”
Minimum	integer
Mnemonic	KeySym
MnemonicCharSet	string
ModifyVerifyCallback	code
MotionVerifyCallback	code
MsgDialogType	“dialog_error”, “dialog_information”, “dialog_message”, “dialog_question”, “dialog_template”, “dialog_warning”, “dialog_working”
MultiClick	“multiclick_discard”, “multiclick_keep”
MultipleSelectionCallback	code
MustMatch	“true”, “false”
MwmDocorations[1]	integer
MwmFunctions[1]	integer
MwmInputMode	“mwm_input_modeless”, “mwm_input_system_modal”, “mwm_input_full_application_modal”, “mwm_input_primary_application_modal”
MwmMenu	string
Name	widget name
NavigationType	“none”, “tab_group”, “sticky_tab_group”, “exclusive_tab_group”

Object Property	**Values**
NoMatchCallback	code
NoMatchString	string
NoResize	“true”, “false”
NumChildren	integer
NumColumns	short
OkCallback	code
OkLabelString	string
Orientation	“vertical”, “horizontal”
OverrideRedirect	“true”, “false”
Packing	“pack_tight”, “pack_column”, “pack_none”
PageDecrementCallback	code
PageIncrement	integer
PageIncrementCallback	code
PaneMaximum	integer
Paneminimum	integer
Parent	widget name
Pattern	string
PendingDelete	“true”, “false”
PopdownCallback	code
PopupCallback	code
PopupEnabled	“true”, “false”
PositionIndex	integer
PreeditType	char *
ProcessingDirection	“max_on_top”, “max_on_bottom”, “max_on_left”, “max_on_right”
PromptString	string
PushButtonEnable	“true”, “false”
QualitySearchDataProc	proc
RadioAlwaysOne	“true”, “false”
RadioBehavior	“true”, “false”
RecomputeSize	“true”, “false”
RefigureMode	“true”, “false”
RepeatDelay	integer
Resizable	“true”, “false”
ResizeCallback	code
ResizeHeight	“true”, “false”
ResizePolicy	“resize_none”, “resize_any”, “resize_grow”
ResizeRecursion	“none”, “up”, “down”
ResizeWidth	“true”, “false”

Object Property	Values
RightAttachment	"attach_none", "attach_form", "attach_opposite_form", "attach_widget", "attach_opposite_widget", "attach_position", "attach_self"
RightOffset	integer
RightPosition	integer
RightWidget	widget name enclosed in quotes
RowColumnType	"work_area", "menu_bar", "menu_pulldown", "menu_popup", "menu_option"
Rows	integer
RubberPositioning	"true", "false"
SashHeight	short
SashIndent	short
SashShadowThickness	short
SashWidth	short
SaveUnder	"true", "false"
ScaleHeight	short
ScaleMultiple	short
ScaleWidth	short
ScrollBarDisplayPolicy	"as_needed", "static"
ScrollBarPlacement	"top_left", "bottom_left", "top_right", "bottom_right"
ScrollHorizontal	"true", "false"
ScrollLeftSide	"true", "false"
ScrollTopSide	"true", "false"
ScrollVertical	"true", "false"
ScrolledWindowMarginHeight	short
ScrolledWindowMarginWidth	short
ScrollingPolicy	"automatic", "application_defined"
SelectColor	color
SelectInsensitivePixmap	bitmap or pixmap filename enclosed in quotes
SelectPixmap	bitmap or pixmap filename enclose in quotes
SelectThreshold	integer
SelectedItemCount	integer
SelectedItems	string
SelectionArray	char *
SelectionArrayCount	integer
SelectionLabelString	string
SelectionPolicy	"single_select", "multiple_select", "extended_select", "browse_select"

Object Property	Values
Sensitive	"true", "false"
SeparatorOn	integer
SeparatorType	"single_line", "double_line", "single_dashed_line", "double_dashed_line", "no_line", "shadow_etched_in", "shadow_etched_out"
Set	"true", "false"
ShadowThickness	short
ShadowType	"shadow_in", "shadow_out", "shadow_etched_in", "shadow_etched_out"
ShellUnitType	string
ShowArrows	"true", "false"
ShowAsDefault	short
ShowInBrowser	"true", "false"
ShowSeparator	"true", "false"
ShowValue	"true", "false"
SingleSelectionCallback	code
SkipAdjust	"true", "false"
SliderSize	integer
Source	text source
Spacing	short
StringDirection	"string_direction_l_to_r", "string_direction_r_to_l", "string_direction_revert"
SubMenuId	widget name enclosed in quotes
SymbolPixmap	bitmap or pixmap filename enclosed in quotes
TearOffModel	"tear_off_enable", "tear_off_disabled"
Text	string
TextAccelerators	translation table name
TextColumns	integer
TextFontList	XmFontList
TextString	string
TextTranslations	translation table name
Title	char *
TitleEncoding	atom
TitleString	string
ToBottomCallback	code
ToTopCallback	code

Object Property	Values
TopAttachment	“attach_none”, “attach_form”, “attach_opposite_form”, “attach_widget”, “attach_opposite_widget”, “attach_position”, “attach_self”
TopCharacter	integer
TopItemPosition	integer
TopOffset	integer
TopPosition	integer
TopShadowColor	color
TopShadowPixmap	bitmap or pixmap filename enclosed in quotes
TopWidget	widget name enclosed in quotes
Transient	“true”, “false”
TransientFor	widget name enclosed in quotes
Translations	translation table name
TraversalOn	“true”, “false”
TroughColor	color
UnitType	“pixels”, “100th_millimeters”, “100th_inches”, “100th_points”, “100th_font_units”
UnmapCallback	code
UseAsyncGeometry	“true”, “false”
UsePropEditor	“true”, “false”
UserData	char *
Value	integer (Note: for text value use property name “text”)
ValueChangedCallback	code
ValueWcs	string
VerifyBell	“true”, “false”
VerticalScrollBar	widget name enclosed in quotes
VerticalSpacing	short
VisibleItemCount	integer
VisibleWhenOFf	“true”, “false”
Visual	visual
VisualPolicy	“variable”, “constant”
WaitForWm	“true”, “false”
WhichButton	Button1, Button2, Button3
Width	short
WidthInc	integer
WindowGroup	string
WinGravity	integer
WmTimeout	integer
WordWrap	“true”, “false”

Object Property	Values
X	`short`
Y	`short`

[1] The Property Editor does not have option menus for these two resources. You must enter their values directly.

The OSF/Motif 1.2 header file `Xm/MwmUtil.h` defines macros for the valid values of these resources (except for the default value, which is `-1`). You may want to use these macros, rather than integer constants, to set the resource values.

Frequently Asked Questions

B

Overview

This appendix contains frequently asked questions (FAQs) extracted from customer feedback of UIM/X. This information is provided in the interest of enhancing user comprehension of UIM/X, helping the user avoid commonly made errors, and addressing work-around solutions to specific problems.

1. Why don't the children of convenience dialogs appear in the Browser?

Since the children of a convenience dialog are created dynamically by the Motif Toolkit, UIM/X has no way of managing them directly. However, you can access these children using the Motif *GetChild* convenience functions (such as `XmMessageBoxGetChild`). For example, to unmanage the Cancel button in a message box, you could use this function in the interface's Final Code:

```
XtUnmanageChild(XmMessageBoxGetChild(UxGetWidget(dia
    logBox), XmDIALOG_CANCEL_BUTTON));
```

2. How do I pass client-specific data to a callback?

In the Property Editor, you see a client data property for each callback:

- ActivateCallback
- ActivateClientData

The ActivateClientData property can be specified as any C pointer expression. The value of that expression is passed to the callback as client data. It is available in `Static` callback code in the variable `UxClientData`. If you use `Extern` callbacks, the client data property is passed as the client data argument to the callback function whose name you specify.

3. Is the Duplicate command on the Menu Editor different from the Copy command on the Selected Objects popup menu?

Yes. The Duplicate command on a Menu Editor does not copy the entire object structure. It only duplicates the type of object, its label string, and callback or next pane. If you have modified any other properties using the Property Editor, the new values for these objects are not copied to the new object.

4. When I add a child to a Paned Window, it completely covers the Paned Window. How do I then add additional children?

There are three ways to resolve this problem.

- If the children of the Paned Window are of the same type, create the first child and then copy it using Selected Objects Duplicate.
- If any of the Paned Window's children are not valid parents, you can place new objects on them. Since they cannot be parents, an object placed on one of them automatically becomes a child of the Paned Window.

- Before adding any children to a Paned Window, set its `RefigureMode` property to `false`. This prevents the Paned Window from resizing its children automatically. After adding all of the children, return `RefigureMode` to `true`.

5. Why is the AllowShellResize property sometimes listed in the Core properties?

When you create a manager object using the Project Window's Create menu, UIM/X automatically creates a shell as a parent of the manager. Properties of this automatically created shell are not directly available. However, since access to the AllowShellResize property is frequently needed, it is displayed with the manager's core properties as a convenience.

6. Why do buttons and labels change size unexpectedly?

The default value for the `RecomputeSize` property is `true`. This means that whenever a property is changed that affects the appearance of the object, the object will attempt to shrink or grow as needed to exactly fit the new appearance. If the `RecomputeSize` property is set to `false`, the object never attempts to change size on its own.

7. Why doesn't the VisibleWhenOff property work for toggle buttons in menus?

By design, the `VisibleWhenOff` property is always forced to `false` when toggle buttons are used in menus. This is mentioned in the toggle button reference information in the *OSF/Motif Programmer's Reference Manual*.

8. How do I position a menu bar at the top of an interface?

In general, the solution to any layout problem is to use the proper manager object as a parent. In this case, the Form object is probably the best solution. Using the Form's constraint properties, you can attach the menu bar to the top, left, and right edges of the Form.

However, in this specific case, the recommended solution is to use a Main Window object, which automatically handles most of the details for creating a window with a menu bar and a scrolled work area.

9. Why does UIM/X sometimes terminate unexpectedly?

UIM/X itself and the application interface you are building share the same process space. This means that if your interface calls a function that terminates or exits the application, it is likely that the call will also successfully terminate or exit UIM/X. There are several exceptions: UIM/X traps the following functions when they are encountered in interpreted code:

```
abort()
assert()
exit()
XtCloseDisplay()
```

When UIM/X encounters any of these functions in interpreted code it will issue a proper diagnostic message instead of terminating the program. Note that there are still ways to terminate the UIM/X session programmatically:

```
close(ConnectionNumber(XtDisplay(UxWidget)));

for (i=0; i<_NFILE; i++)

   close(i);

kill(getpid(), SIGKILL);

XtDestroyWidget(UxGetWidget(shadowWidget));
```

To avoid this problem, you can use `#ifndef` … `#endif` directives to isolate code that you don't want to execute while working with UIM/X. For example, you might surround a call like this:

```
#ifndef DESIGN_TIME

kill(getpid(), SIGKILL);

#endif
```

Since `DESIGN_TIME` is not defined in your application, the `abort()` function is performed as normal. In UIM/X, however, `DESIGN_TIME` is defined and the `abort()` function is avoided.

Although UIM/X does trap calls to these functions in Test mode, Run mode actually generates the application, compiles it, and runs it. In this case the Interpreter is not involved, and all of the above functions will work as expected.

If UIM/X does terminate unexpectedly, it will automatically attempt to save the project, its interfaces and palettes. If successful, the saved files can be found in the directory:

`/tmp/Ux.`*`userName.processNo`*

where *userName* is your user name (or, if none, `AUTO`), and *processNo* is the process number.

10. Can I use the fork(), exec(), and wait() functions?

In Unix it is customary to call a function of the `fork()` family to create subprocesses. UIM/X does not support calls to `fork()` and intercepts them, giving an appropriate diagnostic message. To the code that called `fork()`, it will look as if `fork()` failed because of a lack of memory. Adding memory would not solve this problem; ENOMEM is just a convenient way to provide a believable reason for failure to the interpreted application.

Since you cannot call `fork()` to obtain new processes, it follows that you also cannot call `wait()` to signify the termination of a child, or call `exec()` to run another executable in a child process. For this reason, UIM/X also traps calls to `exec()` and `wait()`.

11. Are there any characters I should avoid using?

If you type hidden or special characters, such as `escape`, into any text object containing code to be processed by the Interpreter, you may see an error message such as:

```
   error: 2006 unrecognized token
```

If the font you are using does not display special characters, it may be difficult to find and delete special characters. If you encounter this problem, the only way to ensure that there are no illegal characters in UIM/X's windows or your files is to save your interface files, edit them manually to remove the special characters, restart UIM/X, and then reload your interfaces.

12. How do I return the Translations property to its default value?

The default value for the `Translations` property is misleading. Although the default value displays as an empty string (""), when you set this property, the value should be the name of the Translation Table *without quotes*. To return the `Translations` property to its default value, use the Default toggle button. If you enter an empty string and leave the property set to Public or Private, the default translations are replaced with an empty table. The Motif default for Translation tables is `replace`, so if you specify an empty string for the `Translations` property, all behavior for the object is destroyed.

13. Sometimes when I make changes, the appearance of my objects does not change as I expect. What should I do?

Due to the complex interactions between objects and their properties, you may have to occasionally recreate objects. This is frequently true when you are working with properties that are intended to be set only at creation time. These properties are defined in the *OSF/Motif Programmer's Reference Manual*.

To recreate an object, select the object, then choose Selected Objects Recreate. To recreate an entire interface, select the top-level object, then choose Selected Objects Recreate.

14. I am trying to use some Xt functions in my application code but they don't seem to be working.

Several of the Xt functions which require an application context also come in convenience versions which use the default application context, for example, `XtAppAddInput()` and `XtAddInput()`.

The code generated by UIM/X creates an application context at initialization via the function `XtAppInitialize()` and records this application context in a global variable `UxAppContext` (of type `XtAppContext`). If you call the Xt functions which use the default application context, you will not be communicating with the application context that UIM/X uses. The same problem will occur if you make an augmented UIM/X and try to use the Xt functions which use the default application context.

15. I augmented UIM/X with my application library code but the Interpreter can't find the symbols.

When you link with a library, only those object modules that are needed to resolve the external references are actually linked into your executable. If you don't refer to any of the symbols in your library, the object modules will not get linked in. The preferred way to force linking of modules from a library is to use the function `UxRegisterFunction()` to register at least one function from each module. This has the side benefit of speeding up access to those functions the first time they are called from the Interpreter.

16. Do I have to use the Ux Convenience Library?

No, with UIM/X you don't have to use it. You can write your callbacks, resources, actions, and other code directly in X, Xt, and Motif. However consider the following:

- Programming directly in X, Xt, and Motif is likely to take longer to develop an application, result in much more code, and give only slight gains in portability.
- You can write your own convenience library for the difficult cases from above (that is, re-implement the Ux Convenience Library), but then you have to develop, test, and debug it. The Ux Convenience Library has been tested by thousands of users.

- The Ux Convenience Library is small, fast, and easy-to-learn and use. Source code is available for the library and it is also very portable (already used on thirty different platforms).

17. Why doesn't UxLoadResources() work when I don't use the Ux Convenience Library?

If you generate code which does not require the Ux Convenience Library the `UxLoadResources()` function is generated as a stub in `UxXt.c`. You can implement the code yourself using a function like `XrmCombineFileDatabase` to non-destructively merge the resources into the server's resource database.

18. Can I use UIL as an interface archive format?

Yes, provided you are willing to forgo all the C code handling features of UIM/X. If you do, you are limited to external callbacks (C code in external `.c` files that must be loaded into the Interpreter in order for the behavior to be tested). You also lose the global declarations and interface specific variables.

19. What is the relationship between UIL and callbacks?

If you were to write UIL by hand, all your callbacks would exist in an external `.c` file. When you use `uxcgen` to generate UIL code for an interface, it also generates a C code file containing all your callbacks. The output from `uxcgen` is similar to what you would write if you were writing UIL code by hand.

20. How do I structure an interface if I plan to use UIL?

We don't recommend doing anything special in structuring the interface if you plan to use UIL. UIL is a code generation option. The widget creation is handled in UIL but everything else is just C code without using the Ux Convenience Library.

Keep the following in mind if you are planning to use UIL:

- You cannot use the `UxPut` and `UxGet` functions (unless you use the macros from `/usr/uimx2.9/contrib/XtCodePuts`).
- You cannot use non-constant C expressions for resource values in the Property Editor.

21. Are there any Motif widgets that cannot be set through the UIM/X Property Editor?

Although supported, the following Motif 1.2 widgets cannot be created interactively and cannot be set through the Property Editor:

- XmDisplay
- XmDragContext
- XmDragIcon
- XmDropSite
- XmDropTransfer
- XmScreen

Index

Index

Index

Index

Index

Index