UIM/X Advanced Topics

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

Contents

Preface

Overview

UIM/X provides a diverse set of configurable tools and capabilities which enable you to extend and customize UIM/X to suit your own unique application.

Using these features, you can:

- Create compound widgets and compound properties
- Create your own palettes
- Create and integrate new widget classes
- Extend the Ux Convenience Library
- Integrate components you have built yourself or purchased from other vendors
- Build and customize UIM/X executables
- Augment UIM/X executables with object code of other applications

This manual describes the mechanisms which allow you to tailor, add to, or simplify UIM/X to create a custom GUI builder.

Who Should Use this Guide

This manual 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 *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 are a competent software developer and wish to extend and customize the standard features of UIM/X to accommodate your unique application.

Related Books

For more information on UIM/X, see the following documents, available at http://www.ics.com/support/docs/:

- *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.

For more information on designing user interfaces, see any of the following:

- *OSF/Motif Style Guide release 1.2* (Prentice Hall, 1993, ISBN 0-13-643123-2)
  The Windows Interface Guidelines for Software Design: An Application Design Guide (Microsoft Corporation, 1995, 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

This document comprises four chapters, seven appendices, and an index, organized as follows:

- *Chapter 1*, "Compound Widgets," discusses creating and working with compound widgets and explains their properties.
- *Chapter 2*, "Integrating Widgets," describes how to integrate a widget class into UIM/X.
- *Chapter 3*, "Integrating Components," describes how to integrate any class into UIM/X.
- *Chapter 4*, "Building Executables," describes how to customize and build UIM/X executables.
- *Appendix A*, "Compound Properties," provides alphabetical listings of compound properties.
- *Appendix B*, "Interface File Format," describes file format concepts and facets and defining new classes.
- *Appendix C*, "Swidget Class Hierarchy," provides a graphical representation of the hierarchy of swidget classes, as well as a table listing each swidget class, the corresponding widget class and the swidget class' private and public header files.
- *Appendix D*, "Resource Types," describes the mechanism for converting between the different data types expected by swidgets and widgets.
- *Appendix E*, "Class Methods," contains the reference pages for the class methods used by UIM/X to operate on widgets.
- *Appendix F*, "Resource Descriptors," describes data objects called resource descriptors. Every widget property in UIM/X is described by a resource descriptor.
- *Appendix G*, "Ux Builder Functions," contains the reference pages for each of the Ux Builder Functions.

Conventions Used in this Guide

Unless otherwise noted in the text, we use the following symbolic conventions:

Typeface or Symbol	Meaning
literal names	Bold words or characters in command descriptions represent words or values that you must use literally.
user-supplied values	Italic words or characters in command descriptions represent values that you must supply. Italic words in text also indicate the first use of a new term, or emphasis
`sample user input`	In interactive examples, information that you must enter appears in **`this typeface.`**
`output/source code`	Information that the system displays appears in `this typeface.`
...	Horizontal ellipsis points indicate that you can repeat the preceding item one or more times.

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
Mwm*activeForeground: #e5e5e5
Mwm*background: #666666
Mwm*foreground: #e5e5e5
Uimx3_0*calculatedColors: false
Uimx3_0*background: #ededed
Uimx3_0*BottomShadowColor: #000000
Uimx3_0*foreground: #000000
Uimx3_0*TopShadowColor: #ffffff
Uimx3_0*XmText.background: #b3b3b3
Uimx3_0*XmTextField.background: #b3b3b3
```

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.

Compound Widgets

1

Overview

A compound widget is a hierarchy of one or more widgets and Component Instances. For brevity, however, this section refers only to widgets in its discussion of compound widgets. Keep in mind that a compound widget can contain Component Instances.

Unlike Component Instances, the instances of a compound widget do not share a common template. Each time you create an instance of a compound widget, you create a duplicate of the original. Changes to a compound widget are not propagated to the instances of that compound widget.

With compound widgets, you can give users the ability to edit the individual widgets in an instance of the compound widget. The compound properties allow you to control the editing operations applicable to each widget in a compound widget. At the same time, you can have operations such as move, resize, and drag and drop apply to the compound widget as a whole.

Compound widgets can also have their own specialized editors.

1

Specifying the Widgets in a Compound

A compound widget consists of a parent widget and zero, one, or more descendants. The parent is the top (and perhaps only) widget in the compound. Given a hierarchy, you specify the widgets included in the compound widget as follows:

1. Set the compound property `IsCompound` to `true` for the top widget or Component in the compound.
2. Set the compound property `IsInCompound` to `true` for each of the parent's descendants that you want to include in the compound. Leave the `IsCompound` property set to `false` for the descendants.

   You can then specify the editing operations that can be applied to the individual widgets and Components in the compound.

The Adjust Button and Compound Widgets

A compound widget can be manipulated as a single widget with the Adjust button (normally the middle mouse button). To do this, you make all widgets in the compound except the top widget transparent to the Adjust button.

When a widget is transparent, pressing the Adjust button on the widget causes the move or resize action to be applied to another widget in the compound. The widget is said to be transparent because it appears to a user that the Adjust button is acting on a widget underneath the one where the mouse pointer is positioned.

Consider the FileSelectionBox widget provided by Motif. It looks like a collection of many widgets, but no matter where you press the Adjust button, you can only move or resize a file selection box as a whole. You can achieve this same behavior by making the individual widgets in a compound transparent.

For example, consider the form shown in Figure 1-1. Note that although the mouse pointer is positioned on the Next Screen button, the form is being resized. Pressing the Adjust button on a transparent button is the same as pressing the Adjust button on the move or resize region of the form underneath the button.

One advantage of transparent widgets is that the user gets some visual feedback when they press the Adjust button on a widget where the move or resize action is disabled. Another is that transparent widgets visually reinforce the idea that a compound widget is an integral whole.

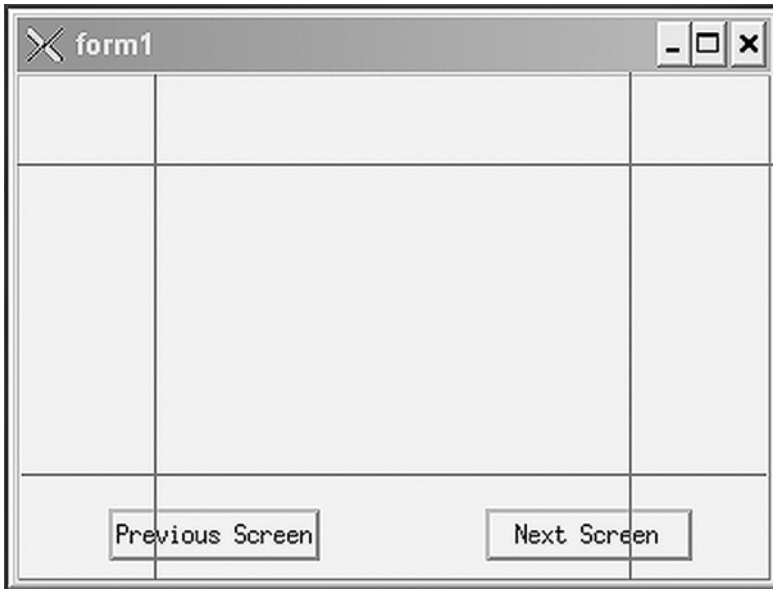

Figure 1-1 Regions and Transparent Widgets

There are three compound properties used to make a widget transparent.

IsRegion

Setting `IsRegion` to `true` makes a widget a region widget. UIM/X uses region widgets to determine whether the Adjust button was pressed on a move or a resize region.

When the Adjust button is pressed, a grid is superimposed on the region widget underneath the mouse pointer (see Figure 1-1). This grid divides the widget into nine different move and resize regions. UIM/X looks at the region where the Adjust button was pressed, and performs the appropriate move or resize operation.

When the Adjust button is pressed on a widget in a compound where `IsRegion` is `false`, UIM/X tries to find a region widget in the widget hierarchy. UIM/X will not perform a move or resize if it cannot find a widget which is a region.

Note that UIM/X cannot know whether a widget is being moved or dragged until the widget is dropped. If a swidget in a compound is reparentable but not moveable, then the user will be able to drag the widget. If the result is a move, then the operation is disallowed.

ResizeRecursion This property determines the direction in which UIM/X traverses the compound widget hierarchy when looking for a resizable widget. UIM/X only checks the value of this property if the region widget is not resizable. Possible values are `up`, `down`, or `none`.

DragRecursion This property determines whether or not UIM/X traverses up through the compound widget hierarchy to look for a draggable widget if the region swidget is not draggable. Possible values are `up` or `none`.

Finding a Region

When the user presses the Adjust button on a widget, UIM/X checks whether the widget is a region widget. If the widget is not a region widget, UIM/X checks the value of the widget's `IsInCompound` property to see if the widget is in a compound.

If the widget is not in a compound, UIM/X does nothing—there is no widget for it to resize or move.

If the widget is in a compound, UIM/X goes up the hierarchy of the compound looking for a region widget. If no region widget is found, UIM/X does nothing. If UIM/X finds a region widget, it checks the mouse press location to determine whether to perform a move or resize.

Once UIM/X knows what action to perform, it has to find a widget to which it can apply the action. This widget may or may not be the region widget. The following sections describe how UIM/X finds resizable and draggable widgets.

Finding a Resizable Widget

If the `IsResizable` property is set to `true` for the region widget, UIM/X allows the user to resize the widget. Otherwise, UIM/X checks the value of the region widget's `ResizeRecursion` property.

The `ResizeRecursion` property tells UIM/X in which direction it must traverse the compound widget's hierarchy. If `ResizeRecursion` is set to `none`, no resize is performed.

UIM/X traverses the compound widget's hierarchy in the specified direction until it finds one of the following:

- A widget that can be resized. UIM/X then allows the user to resize the widget.
- A widget whose `ResizeRecursion` value is not equal to that of the region swidget. If the widget is resizable, UIM/X allows the user to perform a resize.
- A widget that is not in the compound. No resize is performed.

If a resizable widget is not found, nothing happens. If the `ResizeRecursion` of the region widget is `down`, only one descendant, if any, is resized.

Finding a Draggable Widget

If the `IsDraggable` property is set to `true` for the region widget, UIM/X allows the user to drag the widget. Otherwise, UIM/X checks the value of the

region widget's `DragRecursion` property.

The `DragRecursion` property tells UIM/X whether or not it should look for a draggable widget by traversing up the compound widget's hierarchy.

UIM/X traverses up the compound widget's hierarchy until it finds one of the following:

- •A widget that can be dragged. UIM/X then allows the user to drag the widget. When the user drops the widget, UIM/X checks if the widget is
- being moved. If so, UIM/X then checks the value of the widget's
- `IsMovable` property.
- A widget whose `DragRecursion` value is not equal to that of the region swidget. If the widget is draggable, UIM/X allows the user to drag the widget.
- A widget that is not in the compound. In this case, nothing happens since no draggable widget was found.

1

Creating Compound Properties and Swidget Methods

UIM/X enables you to access and manipulate the properties and swidget methods of each individual widget in a compound widget. This is accomplished through the use of the `CompoundResourceSet` and `CompoundSwidgetMethodSet` properties.

Compound resources and compound swidget methods, in effect, allow you to create design-time properties and methods for the individual widgets in the compound.

The Group Box, for example, is a compound widget which comprises a `frame`, a `label`, and a `form`. If you were to change a Group Box's background color, only the color of the `frame` (the parent widget) would change; the background colors of the `label` and `form` would remain unchanged. This is where compound resources and compound swidget methods become useful.

Using compound resources and compound swidget methods, you can ensure that changes applied to the compound widget are applied to the individual widgets in the compound. Accessible through the Compound category of the Property Editor, compound resources and compound swidget methods are specified according to the following formats:

- For compound resources, a quoted string containing one or more resource specifiers, as follows:

  ```
  "<new_resource>:<swidget1>.<resource1>,
     <swidget2>.<resource2>,

     ...

     <swidgetN>.<resourceN>;"
  ```

  You separate adjacent resource specifiers with semi-colons. No semi-colon is required after the last resource specifier.

- For compound swidget methods, a quoted string containing one or more swidget method specifiers, as follows:

  ```
  "<new_sw-method>:<swidget1>.<sw-method1>,
     <swidget2>.<sw-method2>,
     ...
     <swidgetN>.<sw-methodN>;"
  ```

  You separate adjacent swidget method specifiers with semi-colons. No semi-colon is required after the last swidget method specifier. This property cannot be modified if the swidget owning the property is the target of a connection.

Note: In the interest of legibility, the above examples have been placed on separate lines. When you enter design-time properties and swidget methods, specifiers must be placed on one continuous line.

The following is an example of how the `CompoundResourceSet` property may be set for the Group Box compound widget.

```
*groupBox.compoundResourceSet:
"Alignment:labelBox1.childHorizontalAlignment;
Background:groupBox1.background,labelBox1.background,
formBox1.background;FontList:labelBox1.fontList;
Foreground:groupBox1.foreground,labelBox1.foreground,
formBox1.foreground;
LabelPixmap:labelBox1.labelPixmap;
LabelString:labelBox1.labelString;
LabelType:labelBox1.labelType"
```

This code creates the design-time properties `Alignment`, `Background`, `Foreground`, `FontList`, `LabelPixmap`, `LabelString`, and `LabelType` to be applied to the respective widgets in the Group Box.

Note: The design-time properties that you create will appear in the Specific category of the Property Editor.

Similarly, the `CompoundSwidgetMethodSet`property may be set for the Group Box compound widget.

```
*groupBox.compoundSwidgetMethodSet:
"SetBackground:groupBox1.SetBackground,
labelBox1.SetBackground,formBox1.SetBackground;
SetForeground:groupBox1.SetForeground,labelBox1.SetF
   oreground,
formBox1.SetForeground;SetLabelPixmap:labelBox1.SetL
   abelPixmap;
SetLabelString:labelBox1.SetLabelString"
```

This code creates the design-time swidget methods `SetBackground`, `SetForeground`, `SetLabelPixmap`, and `SetLabelString` to perform the appropriate operations on the respective widgets in the Group Box.

Putting a Compound Widget in a Palette

You put a compound widget in a palette as you would any other collection of widgets. As well, you can define the name and the icon used to represent the compound widget in the palette. UIM/X provides compound properties for the name and icon of a compound widget.

Note: Once a compound widget is in a palette, its name and icon can be set from the Edit menu of the palette.

Compound-Name This property holds the name given to the compound widget. This name is displayed on the palette.

CompoundIcon This property identifies the icon used to represent the compound widget. The value of this property must be the name of the file containing the pixmap or bitmap of the icon. Valid file formats are X11 bitmap and XPM.

Installing Compound Editors

Compound widgets can have their own specialized editors. These editors are called compound editors.

There are two compound properties used to install a compound editor. These properties are set for the top widget in the compound—that is, the widget where `IsCompound` is `true`.

Note: UIM/X ignores the settings of the `Editor` and `EditorClientData` properties for the other widgets in a compound—that is, the widgets where `IsInCompound` is `true`.

Editor

This property allows you to enter the callback which pops up the compound editor. This callback function is called whenever you do one of the following:

- Create an instance of the compound widget.
- Double-click the Select mouse button on one of the widgets in the compound widget.
- Select the Compound Editor item from a menu.
- An Editor callback function, like all callback functions, takes three arguments:
- The first argument is `UxWidget`, the widget which triggered the callback. This is always the top widget in the compound widget.
- The second argument is `UxClientData`, the value entered for the EditorClientData property of the top widget in the compound.
- The third argument is `UxCallbackArg`, which is `NULL` for the Compound Editor callback.

Note: In the interface files generated by UIM/X, the `Editor` property is set by a resource specification that looks like this:
bulletinBoard1.compoundEditor:/ Callback code */

A custom example of how to install a compound editor is provided in the RadioPanel contrib located in *uimx_directory*`/contrib/RadioPanel`.

EditorClientData

This property holds any client data to be passed to the callback function that pops up the compound editor.

When you install a compound editor, the value of the property `CompoundName` identifies the compound editor on UIM/X's menus. For example, if you give the name Radio Box to a compound widget, the menu item Compound Editor becomes Radio Box Editor for the compound widget.

1

Integrating Widgets

2

Overview

This chapter describes how to integrate a widget class into UIM/X.

UIM/X fully supports the Motif widget set, which is a library of widget classes derived from the base classes provided by the X Toolkit. The Motif widgets were developed using the general mechanism provided by the X Toolkit for creating new widget classes.

Using this mechanism, developers can create new widget classes by subclassing one of the Motif classes, or by directly subclassing one of the base Xt classes. UIM/X can be extended to support any such custom widget class derived from a Motif or Xt class.

UIM/X treats a new widget class exactly as it does the Motif widget classes. You can interactively create and edit instances of the new class, set property values, and generate code, just as you would for any other widget.

2

Getting Started

To integrate a widget class, you must write a new swidget class and integrate it into UIM/X.

UIM/X uses swidgets to represent widgets. A swidget is a shadow widget—a widget's inseparable companion. A swidget is an object containing the code and data that allows UIM/X to manipulate widgets.

When Ux Convenience Library C++ bindings are being used, an extra level of encapsulation exists. Although UIM/X manipulates Motif widgets internally as swidgets, they are declared in the builder as objects of the Motif wrapper classes provided by the Ux C++ Convenience Library.

UIM/X defines a swidget class hierarchy that parallels the Motif widget class hierarchy. When you integrate a subclass of a Motif (or Xt) widget class, you must subclass the corresponding swidget class. For example, to integrate a subclass of the Motif Primitive class, you must subclass the UIM/X primitive swidget class.

Note: The name of a UIM/X swidget class is the same as the name of the corresponding Motif widget class, except that the swidget class name begins with a lowercase letter. The name of the Ux C++ Convenience Library wrapper class for a swidget is the same as the name of the corresponding Motif widget, except that the name of the class is prefixed with "Ux".

The general procedure for integrating a widget class is as follows:

1. Create a working directory.
2. If you have the source files for the new widget class, copy them to your working directory. If you have only the header files and library for the new widget class, you will have to modify the supplied Makefile to point to these files. See *Building UIM/X*.

3. Copy the following files from *uimx_directory*/`custom/src` to your working directory:

   - `Makefile`

     Template makefile for compiling and linking the widget source code, the swidget source code, and the template `.c` files listed below with UIM/X. Also used to build extended versions of `uxcgen` and `uxreadui` that support new widget classes.

   - cr-menus.c

     Template for adding menu items for new widget classes to the UIM/X Create menus. See *Customizing UIM/X's Create Menus*.

   - cr-mwe.c

     Template for adding menu items to the option menus in the Main Window Editor. See *Customizing the Main Window Editor's Option Menus*.

   - `uxddcppMF.h`

     Template for defining the design-time implementation of the wrapper class member functions.

   - `user-cg-cl.c`

     Template file for extending `uxcgen` and `uxreadui`, the utilities for generating code and reading UIL code.

   - `user-class.c`

     Template for placing calls to the functions that register new swidget classes with UIM/X (these functions are defined in the source files for the swidget classes).

   - `user-rtime.c`

     Template for registering properties for run-time conversion (between the different data types expected by widgets and swidgets). This allows you to use the Ux Convenience Library in the code generated for new widget classes.

   - `user-xtype.c`

     Template for defining new xtypes. An xtype specifies the data type and values of a widget property.

4. Write the source for a new swidget class.
5. Fill in the template `.c` files as required to integrate your new swidget class.

6. Modify `Makefile` to point to the correct files.
7. Compile and link extended versions of `uimx`, `uxcgen`, and `uxreaduil`.

The rest of this chapter is a detailed discussion of how to write a new swidget class and integrate it with UIM/X. As an example, the discussion refers to the code required to integrate the Dog and Square widget classes. The source for these examples can be found in *uimx_directory*`/contrib/DogAndSquare`.

Swidget Class Source Files

The source for a swidget class is contained in two header files and one file of source code. The names of these files are derived from the name of the swidget class. For example, the dog swidget class is implemented in the following files:

- The private header file, `dog.cl.h`, defines the swidget's class and instance structures.
- The source code file, `dog.cl.c`, contains the code defining the swidget class.
- The public header file, `UxDog.h`, contains the definitions of the macros used to manipulate instances of the swidget class, as well as a definition of the `UxDog` wrapper class.

Writing the Private Header File

The private header file of a swidget class (the `.cl.h` file) defines its class and instance structures. UIM/X uses these two structures to implement swidget classes and instances.

The class structure's fields contain the properties—such as pointers to data structures and methods—common to all instances of the swidget class. The instance structure contains the internal details of a swidget instance—for example, pointers to the swidget's Values and Expressions lists.

New swidget classes don't add fields to the instance structure, since these fields contain information used only by UIM/X. For this reason, the following discussion focuses on the class structure.

The Class Structure

The organization of the class (and the instance) structure is determined by the swidget class hierarchy from which the new class is derived. The class structure of the dog swidget class, for example, contains the class fields defined by each of its superclasses, as well as its own class fields.

Note: Given the superclass of the new widget class, you can determine which swidget class to subclass by consulting Appendix C, "Swidget Class Hierarchy." You can use this appendix to find the swidget class corresponding to the superclass of the new widget class.

A swidget class defines its class fields in a separate structure called a `ClassPart`. This `ClassPart` structure is then combined with the `ClassPart` structures of each of the superclasses to form a `Class` structure. By convention, these structures are named `Ux`*WidgetName*`ClassPart` and `Ux`*WidgetName*`Class` in the private header files of the UIM/X swidget classes.

The definitions of the `ClassPart` and `Class` structures for the dog swidget class are shown below:

```
#include "prim.cl.h" /* swidget-superclass header
   file */

/* Definition of the dog class structure */

typedef struct UxDogClassPart
{
   Resource_t *RD_wagTime;
   Resource_t *RD_barkTime;
   Resource_t *RD_barkCallback;
} UxDogClassPart;


typedef struct UxDogClass
{
   UxObjectClassPart object;
   UxVeditableClassPart veditble;
   UxShadowWidgetClassPart ShadowWidget;
   UxRectObjectClassPart RectObject;
   UxCoreClassPart Core;
   UxPrimitiveClassPart primitive;
   UxDogClassPart dog;
} UxDogClass;
```

The UxPrimitiveClassPart structure is defined in prim.cl.h, the private header file of the immediate superclass of the dog swidget class. Note that prim.cl.h is included at the top of the dog class' private header file.

The private header file of a swidget class includes the private header file of its superclass. Thus prim.cl.h includes Core.cl.h, Core.cl.h includes RectO.cl.h, and so on. Including prim.cl.h gives you access to the definitions of the ClassPart structures for each of its superclasses.

Note: When you write the private header file for a new swidget class, you can simply copy and modify the definitions of the ClassPart and Class structures in the private header file of its superclass.

Class Properties

The ClassPart structure of a swidget class contains a field of type Resource_t* for each new property defined by the swidget class. The dog swidget class, for example, defines three new properties. The ClassPart structures of the superclasses of a swidget class define the properties inherited by the swidget class.

The Resource_t* variables are pointers to the resource descriptors associated with the new properties. A resource descriptor is a data structure defined by UIM/X (see *uimx_directory*/custom/include/resource.h). By convention, these variables are named RD_*propertyName*.

These resource descriptors are initialized and installed in the swidget's source (.cl.c) file.

Class Methods

A swidget class inherits the methods of its superclasses. You can replace or augment inherited methods.

A swidget class can also define new methods. The ClassPart structure contains a field of type vhandle for each new method defined by the swidget class. The vhandle variables are internal identifiers used by UIM/X. By convention, these variables are named *_MethodName*. For example, if the dog class was to define a new method named Woof, the ClassPart structure would look like this:

```
typedef struct UxDogClassPart
{
   vhandle _Woof;
   Resource_t *RD_wagTime;
   Resource_t *RD_barkTime;
   Resource_t *RD_barkCallback;
} UxDogClassPart;
```

New class methods are registered in the swidget's source (`.cl.c`) file.

The Instance Structure

As mentioned previously, a swidget class does not need to add fields to the instance structure. However, an instance structure must still be defined for the swidget class. The standard approach is to use the definition of the superclass' instance structure:

```
typedef primitive dog;
```

In this example, the `dog` instance structure is defined to be the instance structure of the primitive swidget class.

Global Variables

The private header file should contain `extern` declarations for the global variables used by the class:

```
/* Declarations of global variables for dog class */
/* Swidget class ID returned by UxRegister_class
 * in dog.cl.c.
 */
extern Class_t UxC_dog;
/* Class property IDs returned by calls to
* UxFixed_class_prop in dog.cl.c.
 */
extern binptr UxP_DogRD_wagTime;
extern binptr UxP_DogRD_barkTime;
extern binptr UxP_DogRD_barkCallback;
/*
   * Class method IDs (if any) returned by calls to
 * UxFixed_class_method in the .cl.c file. There are
   none
* for the dog class.
*
* extern binptr UxM_Woof;
*/
```

Summary

The private header file of a swidget class always has the same basic layout:

- An `#include` of the private header file of the swidget superclass.
- A `typedef` for the `ClassPart` structure.
- A `typedef` for the `Class` structure.
- A `typedef` for the instance structure.
- `extern` declarations for the global variables of the swidget class. These variables are declared and initialized in the swidget's source file.
- The swidget class ID (a `Class_t` variable).
    - IDs of class properties.
    - IDs of class methods.

Writing the Swidget Class Source File

The swidget class' `.cl.c` file contains the function that defines the swidget class. This function registers the swidget class with UIM/X and initializes the class structure.

This section describes the organization of the `.cl.c` file, and tells you how to write the function that registers the swidget class.

Include Files

The `.cl.c` file includes the following files:

- `<Xm/Xm.h>`, the general header file for Motif.
- The public header file of the widget class.
- `"veos.h"`, for the declarations of VEOS (Internal Object System) functions.
- `"valuesOf.h"`, which also includes `"validate.h"` and `"utype.h"`, for the declarations of the `UxValuesOf` functions, the `UxValidate` functions, and the xtype and utype IDs. These names are referred to when you initialize the resource descriptors of the new class properties.
- The private header file of the swidget class.

The #include statements from dog.cl.c are shown below:

```
#include <Xm/Xm.h>     /* Motif header file */
#include "Dog.h"       /* Widget-class public header file */

#include "veos.h"      /* object system */
#include "valuesOf.h"  /* ValuesOf & Validator functions */

#include "dog.cl.h"    /* private swidget-class header file */
```

Global Variable Definitions

Following the #include statements, the .cl.c file should define the global variables used to hold the swidget class ID, the class property IDs, and the class method IDs. These variables are assigned values in the function that registers the dog class.

The dog swidget class defines the following global variables:

```
Class_t UxC_dog = NULL_CLASS;

binptr UxP_DogRD_wagTime;

binptr UxP_DogRD_barkTime;

binptr UxP_DogRD_barkCallback;
```

- UxC_dog, the swidget class ID, is the value returned by UxRegister_class.
- The binptr variables are the class property IDs returned by UxFixed_class_prop.
- The dog class does not define any new class methods. Class method IDs, which are also binptr variables, are returned by UxFixed_class_method.

Defining the Swidget Class

The .cl.c file contains the definition of the function that defines the swidget class. This function, conventionally named UxRegister_*swidgetClass*, is called from the function UxAddUserDefClasses in user-class.c.

The function which defines the swidget class has to accomplish two main tasks:

1. Register the swidget class with UIM/X.
2. Initialize the class structure.

Registering the Swidget Class

A swidget class is registered by calling `UxRegister_class`. This is the first thing done by `UxRegister_dog`:

```
{
UxC_dog = UxRegister_class("dog",
   UxC_primitive,
   sizeof(dog),
   sizeof(UxDogClass) );
```

This call registers the `dog` swidget class as a subclass of the primitive swidget class (`UxC_primitive` is the class ID of the primitive class). The value returned by `UxRegister_class` is the ID of the `dog` class.

Initializing the Class Structure

The class structure is initialized by a series of function calls. The initialization process can be broken down as follows:

- Perform general class initialization by setting various UIM/X class properties. These are internal class properties defined by UIM/X, and there are no resource descriptors associated with these properties.
- Initialize the class resource descriptors:
- Inherit the class properties of the swidget superclasses.
- Initialize the resource descriptors of any new class properties and add them to the class' resource set (the PList of resource descriptors for the class properties—see `UxGetResourceSet` in Appendix G, "Ux Builder Functions").
- Register any new or overriding class methods.

General Class Initialization

A swidget class has a number of properties that can be set during class initialization. In particular, there are two UIM/X class properties that must be set. These are the class properties that specify the name of the swidget class public header file and the name of the corresponding widget class:

```
/* General class initialization */

UxPutUxFilename( UxC_dog, "UxDog.h" );
UxPutToolKitClass( UxC_dog, (char *) dogWidgetClass
   );
```

There are a number of other UIM/X class properties that can be set:

- The name of the bitmap file containing the icon used to represent the swidget class. This property is set by `UxPutIconBitmap`.
- The Class Editor properties. These class properties specify the specialized editor to be used to edit instances of the class. These properties are set by the `UxPutClassEd*` functions.

For example, suppose you wrote a specialized editor for dog swidgets. You could install this editor by adding the following code in `UxRegister_dog`:

```
/* The popup function for the editor */
extern swidget UxPopupDogEditor();

UxPutClassEdName ( UxC_dog, "Dog Editor..." );
UxPutClassEdMnemonic ( UxC_dog, "d" );
UxPutClassEdIsFavorite ( UxC_dog, 1 );
UxPutClassEdForChild ( UxC_dog, 1 );
UxPutClassEdPopup ( UxC_dog, UxPopupDogEditor );
```

Initializing the Resource Descriptors

There are two steps to initializing the resource descriptors of a swidget class. First, the properties of the swidget superclass must be inherited. This is accomplished by calling `UxInheritResources`:

```
UxInheritResources( UxC_dog );
```

Note: UxInheritResources must be called before you add any new properties defined by the swidget class. UxInheritResources gives the derived class (the dog class in the above example) its own copy of the superclass' resource set. If you don't call `UxInheritResources`, any properties you add will be added to the resource set of the superclass.

This is because a derived class shares the resource descriptor of an inherited property with the class that originally defined the property. Note that `UxPutClassResource` and `UxDefineResource` can be used to give a derived class its own resource descriptor for an inherited property.

Second, the resource descriptors of any new properties defined by the swidget class must be initialized and added to the class' resource set. Before you can do this, however, you must obtain a class property ID:

```
UxP_DogRD_wagTime = UxFixed_class_prop(
    "RD_wagTime",
    UxC_dog,
    T_PNTR,
    Offset( UxDogClass, dog.RD_wagTime ) );
```

UxFixed_class_prop registers the wagTime property and returns its ID. The class property ID is passed to UxPutClassResource, the function that adds the property's resource descriptor to the resource set:

```
UxPutClassResource( UxC_dog,
    UxP_DogRD_wagTime,
    UxDefineResource(
        RD_NAME, "wagTime",
        RD_XTNAME, DogNwagTime,
        RD_UTYPE, UxUT_int,
        RD_XTYPE, UxXT_int,
        RD_VALUESOF, UxValuesOfNonnegativeInt,
        RD_VALIDATOR, UxValidateNonnegativeInt,
        RD_DIVISION, UxSPECIFIC,
        /* RD_PUT, UxStdPut_int, (default) */
        /* RD_GET, UxStdGet_int, (default) */
        /* RD_PASS, UxPASS0, (default) */
        RD_END ) );
```

In the above code, `UxDefineResource` initializes a resource descriptor for the wagTime property and returns its `Resource_t*`, which is then passed to `UxPutClassResource`. This must be repeated for each new property defined by the swidget class. See Appendix F, "Resource Descriptors," for a description of the fields in the resource descriptor.

Note: Note that `UxDefineResource` specifies the utype and xtype of the new property. If a widget class declares a property for which there is no corresponding UIM/X xtype, you must define a new xtype before initializing the resource descriptor. See *Defining New Xtypes*.

A derived class can be given its own resource descriptor for an inherited property. For example, the following call to UxPutClassResource gives the dog class its own resource descriptor for the background property defined by the core class:

```
UxPutClassResource( UxC_dog,
    UxP_CoreRD_background,
    UxDefineResource
        (RD_EXAMPLE, UxGetRD_background(UxC_dog),
        RD_END ) );
```

The parameter RD_EXAMPLE tells UxDefineResource to get a copy of the resource descriptor specified by the following parameter and re-initialize its fields.

Registering New Class Methods

Suppose the dog class defined a class method named Woof. This would require a number of changes to dog.cl.c:

- There would be a global declaration at the top of dog.cl.cfor the ID of the class method:

  ```
  binptr UxM_Woof;
  ```

- The UxDogClassPart structure would contain a field of type vhandle for the class method:

  ```
  typedef struct UxDogClassPart

  {
      vhandle _Woof
      Resource_t *RD_wagTime;
      Resource_t *RD_barkTime;
      Resource_t *RD_barkCallback;
  } UxDogClassPart;
  ```

- UxRegister_dog would contain a call to UxFixed_class_method to register the Woof method:

  ```
  UxM_Woof = UxFixed_class_method( "UxWoof", UxC_Dog,
      T_void, Offset( UxDogClass,dog._Woof ) );
  ```

- After the registration of the Woof method, UxInit_method would be called to install the function to be used as the Woof method:

  ```
  UxInit_method( UxC_dog, UxM_Woof, WoofFunction );
  ```

The declaration of `WoofFunction` would also have to be made available in `dog.cl.c`.

Class methods are invoked using the `Ux`*Type*`_get_op` functions (where *Type* is one of `PNTR`, `Void`, or `Int`, and corresponds to the return type of the method). For convenience, you may want to use a macro to invoke a class method. For example, this macro definition could be added to `dog.cl.h`:

```
#ifndef UxWoof
#define UxWoof( obj ) Ux Type _get_op( obj, UxM_Woof )(
   obj )
#endif
```

Note: If you add a new method, you are responsible for making sure that the method is not invoked by a swidget or class that does not know about the method. For example, the Woof method cannot be invoked by the superclasses of the dog class:

```
if ( UxIsSubclass( sw, UxC_dog ) )UxWoof( sw );
```

Summary

The `.cl.c` file for a swidget class has this basic structure:

- •`#include` statements for the required header files:
  - <Xm/Xm.h>
  - The public header file of the widget class.
  - "veos.h"
  - "valuesOf.h"
  - The private header file of the swidget class.
- Global declarations for the variables used by the swidget class:
  - A `Class_t` variable for the swidget class ID.
  - `binptr` variables for the class property and class method IDs.
- The definition of the function `UxRegister_`*class*. This function does the following:
  - Registers the class by calling `UxRegister_class`.
  - Sets UIM/X class properties by calling `UxPutUxFilename` and `UxPutToolKitClass`.
  - Inherits properties by calling `UxInheritResources`.
  - Registers new properties by calling `UxFixed_class_prop`.
  - Initializes the resource descriptors for new properties by calling `UxDefineResource`, and installs the resource descriptors by calling `UxPutClassResource`.
  - Registers new class methods.

Writing the Public Header File

The public header file for a swidget class contains the conditional definitions of C++ class bindings, design-time, and run-time macros:

- `UxPut` and `UxGet` macros for setting and retrieving the values of the new properties defined by the swidget class.
- `UxCreate` macros for creating instances of the new swidget class.
- C++ class declarations.

The design-time macros are used in code compiled with the -DDESIGN_TIME flag, namely UIM/X. The run-time macros are used in applications compiled from generated code.

Note: When you compile generated code, you define the symbol DESIGN_TIME only if you want to link your application with UIM/X.

The public header files use an #ifdef ... #else ... #endif construct to conditionally define the macros. The #ifdef directive tests whether or not the symbol DESIGN_TIME is defined. For example, UxDog.h, the public header file for the dog swidget class, has the following structure:

```
#ifndef UXDog_INCLUDED
#define UXDog_INCLUDED
#include "Dog.h"/* Widget class public header file */
#include "UxPrim.h"
#if defined(__cplusplus) && !defined(XT_CODE)
   /* Class declaration */
   /* Constructors */
   /* Initialisation */
   /* Resource accessor functions */
#endif /* __cplusplus */

#ifdef DESIGN_TIME
   #if defined(__cplusplus) && !defined(XT_CODE)
   /* extern binding */
   #endif /* __cplusplus */
   /* Design-time UxGet and UxPut macros */
   /* Design-time create macro */
#else
#if defined(__cplusplus) && !defined(XT_CODE)
   /* Constructors */
   /* Initialization */
#endif /* __cplusplus */
```

```
    /* Run-time UxGet and UxPut macros */
    /* Run-time create macro */
#endif /* DESIGN_TIME */
#endif /* UXDog_INCLUDED */
```

This example shows the include files required by a public header file:

- The public header file of the corresponding widget class (`UxDog.h` above) must be included.
- The public header file of the swidget superclass (`UxPrim.h` above) must be included if the symbol `DESIGN_TIME` is defined.

Design-Time Macros

For each new property defined by the swidget class, the public header file must define design-time `UxPut` and `UxGet` macros:

```
/* Design-time UxGet and UxPut macros for DogNwagTime
   */
extern binptr UxP_DogRD_wagTime;
#define UxGetWagTime( sw ) \
   UxGET_int( sw, UxP_DogRD_wagTime, "wagTime" )
#define UxPutWagTime( sw, val ) \
   UxPUT_int( sw, UxP_DogRD_wagTime, "wagTime", val )
```

In general, the design-time UxGet and UxPut macros should use the UxGET_*type* and UxPUT_*type* functions corresponding to the utype of the property. For example, if the utype of a property is UxUT_string, then the UxGET_string and UxPUT_string functions should be used.

Note that an `extern` declaration of the class property ID is required.

You must also define a macro for creating instances of the swidget class in UIM/X:

```
/* Design-time create macro */
extern Class_t UxC_dog;
#define UxCreateDog( name, parent ) \
   UxCreateSwidget( UxC_dog, name, parent )
```

Design-Time C++ Member Functions

For each new property defined by the swidget class, the file uxddcppMF.cc must define `Get` and `Set` accessor member functions for design-time use. For example:

```
int UxDog::GetWagTime() const
{ return UxGET_int(UxThis, UxP_DogRD_wagTime,
   "wagTime"); }


void UxDog::SetWagTime(int val)
{ UxPUT_int(UxThis, UxP_DogRD_wagTime, "wagTime",
   val); }
```

In general, the bodies of these member functions should be equivalent to the bodies of the corresponding design-time macros.

You must also define a parameterless constructor, a parametered constructor, and a `CreateSwidget` member function for the class. For example:

```
// Constructors
   UxDog::UxDog() {};
   UxDog::UxDog(const char* name, swidget uXParent)
   {


CreateSwidget(name, uXParent);
   };
```

Run-Time Macros

For each new property defined by the swidget class, the public header file must also define run-time `UxPut` and `UxGet` macros:

```
/* Run-time UxGet and UxPut macros for DogNwagTime */
#define UxGetWagTime( sw ) \
   (int) UxGetProp( sw, DogNwagTime )
#define UxPutWagTime( sw, val ) \
   UxPutProp( sw, DogNwagTime, (XtArgVal)(val) )
```

You can use `UxGetProp` and `UxPutProp` when no run-time conversion of property values is required. Run-time conversion of property values is

required when the xtype and utype of a property differ—that is, when the swidget and the widget don't use the same data type to represent a property value. Note that the value returned by `UxGetProp` must be cast to the appropriate type.

When run-time conversion of property values is required, the macros must use `UxDDGetProp` and `UxDDPutProp`.

You must also define a macro for creating instances of the swidget class in compiled generated code:

```
/* Run-time create macro */

#define UxCreateDog( name, parent )
   \UxCreateSwidget( name, dogWidgetClass, parent )
```

Run-Time C++ Member Functions

For each new property defined by the swidget class, the public header file must also contain `Get` and `Set` accessor member functions for run-time use. For example:

```
inline int UxDog::GetWagTime (void) const
   { return (int)DDGetProp(DogNwagTime);}
inline void UxDog::SetWagTime ( int val)
   { DDSetProp(DogNwagTime, ((XtArgVal)(val)));};
inline int UxDog::GetBarkTime (void) const
   { return (int)DDGetProp(DogNbarkTime);}
inline void UxDog::SetBarkTime ( int val)
   { DDSetProp(DogNbarkTime, ((XtArgVal)(val)));};
```

In general, the bodies of these member functions should be equivalent to the bodies of the corresponding run-time macros.

You must also define a parameterless constructor, a parametered constructor and a CreateSwidget() member function for the class. For example:

```
// Constructors
inline UxDog::UxDog(void) {};
inline UxDog::UxDog(const char* name, swidget
   uXParent)
{
 CreateSwidget(name, uXParent);
   };
```

Note: The design-time `UxPut` and `UxGet` macros as well as the design-time C++ accessor member functions for existing properties are defined in the public header files of the UIM/X swidget classes. The run-time macros are defined in the files `UxPutsMF.h` and `UxGetsMF.h` in *uimx_directory*`/custom/include`.

Summary

The public header file of a swidget class should contain the following elements:

- An `#include` of the public header file of the widget class.
- Definitions of the design-time macros for the swidget class:
  - `UxPut`*Property* and `UxGet`*Property* macros for setting and retrieving the values of the new properties defined by the swidget class.
  - A `UxCreate`*Swidget* macro for creating instances of the swidget class in UIM/X.
- Definitions of the run-time macros for the swidget class:
  - `UxPut`*Property* and `UxGet`*Property* macros for setting and retrieving the values of the new properties defined by the swidget class.
  - A `UxCreate`*Swidget* macro for creating instances of the swidget class in generated code.
- Definitions of the run-time C++ accessor member functions for the swidget class:
  - `Set`*Property* and `Get`*Property* macros for setting and retrieving the values of the new properties defined by the swidget class.
  - A `Create`*Swidget* constructor for creating instances of the swidget class in generated code.

In addition, the following should be added to the file `uxddcppMF.cc`:

- Definitions of the design-time C++ accessor member functions for the swidget class:
  - `Set`*Property* and `Get`*Property* macros for setting and retrieving the values of the new properties defined by the swidget class.
  - A `Create`*Swidget* constructor for creating instances of the swidget class in generated code.

Building UIM/X

You can use the makefile *uimx_directory*`/custom/src/Makefile` to compile the source for the new widget and swidget classes and link them with UIM/X.

If you have the source for the widget class, the `WIDGET_OBJECTS` must list the object files for the new widget class. The macro `SWIDGET_OBJECTS` must list the object files for the swidget class. For example, you would define these macros as follows to build a version of UIM/X that supports the Dog and Square widget classes:

```
WIDGET_OBJECTS = Dog.o Square.o
SWIDGET_OBJECTS = dog.cl.o square.cl.o
```

If you have only the header files and library for a new widget class, you must modify the makefile to point to these files. You can use the `X_CFLAGS` macro to specify the include path, and the `UX_LIBS` macro to specify the library:

```
X_CFLAGS = -I/usr/include/X11R5 -I/where/they/are
UX_LIBS = ExistingFLAGS -L/where/they/are -lXfwf
```

As well, you would have to compile and link any of the `template .c` files you have modified. See *Using the Custom Makefile* for more about using this makefile.

Creating Widgets from UIM/X's Menus

In UIM/X, users create widgets from the Create menus of the Project Window, the Browser, and the Selected Widgets popup menu. You can add items to these Create menus to allow users to create instances of a new widget class.

As well, you can add items to the Message Window and Work Area option menus of the Main Window Editor. These option menus create the message window and work area elements of a main window.

2

Customizing UIM/X's Create Menus

UIM/X's Create menus list the types of widgets the user can create. Figure 2-1 shows the Create menu for custom widgets (custom widgets are new widget classes derived from the base Xt and Motif widget classes).

The Create menus are defined by the functions in *uimx_directory*/`custom/src/cr-menus.c`. By modifying this file, you can customize the Create menus:

- You can add new menu items. For example, when you integrate new widget classes with UIM/X, you can let users create instances of the new classes from the Create menus.
- You can remove menu items. For example, you might want to remove the menu items for hidden classes.
- You can rearrange the Create menus. In a custom GUI builder, you might want to rename and rearrange the items on the Create menus.

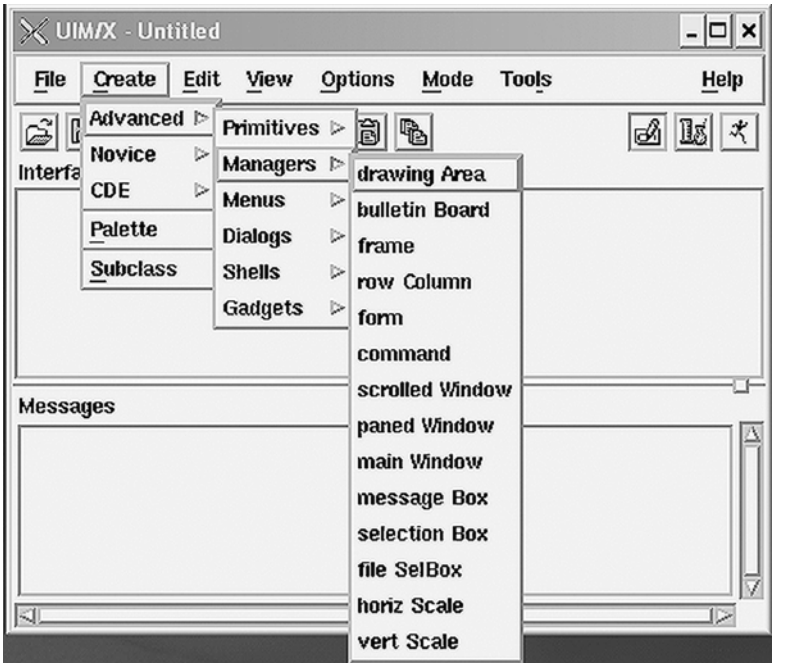

Figure 2-1 Example of a Create Menu

In *uimx_directory*/`custom/src/cr-menus.c`, there is a function for each Create menu:

- The Create menus of the Project Window.
- The Create menus of the Browser.
- The Create menus of the Selected Objects popup menu.

The following table lists the functions that define UIM/X's Create menus. Each of these functions contains a series of calls to `UxAddToCreateMenu`. Each call to `UxAddToCreateMenu` adds an item to a menu. You customize the Create menus by adding, removing, and modifying calls to `UxAddToCreateMenu`.

Interface	**Create menu**	**Function**
Project Window	Shells Managers Dialogs Custom	UxSpecifyTopShellsMenu() UxSpecifyTopManagersMenu() UxSpecifyTopDialogsMenu() UxSpecifyTopCustomMenu()
Browser Selected Objects popup	Managers Primitives Gadgets Custom Menus	UxSpecifyManagersMenu() UxSpecifyPrimitivesMenu() UxSpecifyGadgetsMenu() UxSpecifyCustomMenu() UxSpecifyMenusMenu()

The following code adds the Square menu item (see Figure 2-1) to the Project Window Custom menu. This code is taken from *uimx_directory*/`contrib/DogAndSquare/cr-menus.c`.

```
void UxSpecifyTopCustomMenu( casc_swgt, rowcol_swgt
   )
   swidget casc_swgt;
   swidget rowcol_swgt;
{
   extern Class_t UxC_square;

   (void) UxAddToCreateMenu( rowcol_swgt,
      "Square",
      "q",
      TRUE,
      NULL,
      UxC_square,
      NULL );
}
```

This code shows the general format of the functions that define the Create menus:

- There is an `extern Class_t` declaration for each swidget class for which there is an item on the menu. The `Class_t` variables are the swidget class IDs returned by `UxRegister_class` during class registration.

  In the above example, `UxC_square` is the ID of the square swidget class. This swidget class is registered in *uimx_directory*`/contrib/DogAnd-Square/square.cl.c`.

  A `Class_t` variable is required for each menu item that allows the user to interactively create a widget—that is, by clicking, pressing, and dragging the mouse pointer.

- `UxAddToCreateMenu` is called once for each item on a menu. For full details, see the reference page for this function in Appendix G, "Ux Builder Functions". In the call to `UxAddToCreateMenu` in the above example:

  - The first argument is the rowColumn swidget (the Custom menu pane) passed to `UxSpecifyTopCustomMenu` by UIM/X.
  - The next two arguments are the label and mnemonic of the menu item. These strings can be defined in the UIM/X message system.
  - The fourth argument specifies whether or not the menu item creates a top-level widget. The value `TRUE` indicates that a top-level square widget will be created.
  - The two `NULL` arguments tell UIM/X that there are no user-supplied functions to be called before and after the user creates a square widget.
  - The sixth argument, the swidget class ID `UxC_square`, tells UIM/X what type of widget to create when the user selects the item from the menu.

    If this argument is `NULL`, the widget is not interactively created using the mouse. `NULL` is passed when you want to pop-up a specialized editor—such as the Menu Editor—to create the widget. This can be done by passing the popup function for the editor as the fifth argument to `UxAddToCreateMenu`. See `UxSpecifyMenusMenu` in *uimx_directory*`/custom/src/cr-menus.c`.

Customizing the Browser's New Option

When the Browser is the start-up interface, its File menu contains an item named New. This menu item creates a widget. The class of widgets created by the New option is specified in the function `UxSpecifyTheNewMenu`:

```
swidget UxSpecifyTheNewMenu( rowcol_swgt )
   swidget rowcol_swgt;
{
   extern Class_t UxC_drawingArea;
   extern void UxSetUntitledName();
   swidget new;
new = (swidget) UxAddToCreateMenu( rowcol_swgt,
   CGETS( MS_WB_NEW, DS_MS_WB_NEW ),
   CGETS( MS_WB_NEW_ACC, DS_MS_WB_NEW_ACC ),T
   RUE,
   (void (*)()) NULL,
   UxC_drawingArea,
   UxSetUntitledName);
return new;
}
```

This function is also defined in *uimx_directory*`/custom/src/cr-menus.c`.

By default, the New item creates DrawingArea widgets. To create an instance of another widget class, simply replace both occurrences of `UxC_drawingArea` with another swidget class ID. The function `UxSetUntitledName` generates a unique name for a widget.

Customizing the Main Window Editor's Option Menus

The Main Window Editor has option menus for creating the work window and message window components of a main window. You can customize these option menus.

The file *uimx_directory*`/custom/src/cr-mwe.c` contains the two functions that create the Main Window Editor's option menus.

Each of these functions is a series of calls to the functions UxAddToMweEditor and UxAddMweEditorSeparator. Refer to the reference pages for these two functions in Appendix G, "Ux Builder Functions".

For example, in *uimx_directory*/custom/src/cr-mwe.c, the following code defines the Message Window option menu:

```
void UxCreateMweMsgWindow( ptr )
   void *ptr;

{
   extern Class_t UxC_separator,
      UxC_label,
      UxC_text,
      UxC_textField;
UxAddToMweEditor( ptr, CGETS_MWE(NONE), (Class_t)0);
UxAddMweEditorSeparator( ptr );
UxAddToMweEditor( ptr, CGETS_MWE(LABEL), UxC_label);
UxAddToMweEditor( ptr, CGETS_MWE(TEXT), UxC_text);
UxAddToMweEditor( ptr,CGETS_MWE(TEXTFIELD),
   UxC_textField);
}
```

- The Class_t variables are the swidget class IDs returned by UxRegister_class during class registration. The Class_t variable passed to UxAddToMweEditor tells UIM/X the class of the swidgets created by the menu item.
- The first call to UxAddToMweEditor adds the None item to the Message Window option menu. None destroys any previously selected Message Window swidget. This behavior is achieved by passing (Class_t)0 as the swidget class ID.
- UxAddMweEditorSeparator adds a separator between the first two items on the option menu.
- The subsequent calls to UxAddToMweEditor add menu items for label, text, and textField swidgets.

The CGETS_MWE macro retrieves Main Window Editor messages from the message catalog. This macro is defined in *uimx_directory*/custom/include/cat_macros.h. It calls the macro UXCATGETS which is defined in *uimx_directory*/custom/include/uimx_cat.h.

When you add your own menu items, you can just pass the actual message strings, unless you have set up catalog messages.

You can customize the Message Window option menu by modifying UxCreateMweMsgWindow. For example, adding the following code to UxCreateMweMsgWindow would add the Dog menu item to the Message Window option menu:

```
extern Class_t UxC_dog;

UxAddToMweEditor( ptr, "Dog", UxC_dog );
```

Figure 2-2 shows the Message Window option menu obtained by adding this code to UxCreateMweMsgWindow.

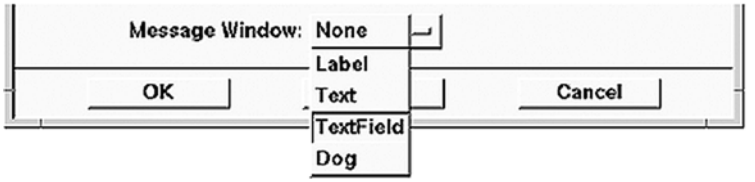

Figure 2-2 Message Window Option Menu

Defining New Xtypes

The data types used to store the property values of a swidget are not necessarily the same as the data types used by the actual widget (in UIM/X, most property values are stored as strings and integers).

UIM/X provides a mechanism for converting between the different data types expected by swidgets and widgets. In UIM/X, each property has a utype and an xtype. The utype specifies the data type of a swidget property. The xtype specifies the data type and possible values of a widget property. For each utype-xtype pair, UIM/X defines a converter function to convert values between the utype and the xtype.

If a widget class declares a property for which there is no corresponding xtype, you must do the following before initializing a resource descriptor for the property:

- Define a new xtype. The ID of the xtype is stored in the resource descriptor of the property.
- Define and register a converter function for the new xtype.

As well, you may want to supply new Validator and ValuesOf functions for the property. A Validator function validates property values. A ValuesOf function provides a textual description of the allowable property values.

UIM/X uses the ValuesOf functions to compose error messages and construct option menus (if a ValuesOf function returns a non-zero value, UIM/X constructs an option menu for the property). The error messages are displayed in the Message Window when an invalid property value is entered, and the option menus are used in the Property Editor to set property values.

You specify the names of the ValuesOf and Validator functions in the call to `UxDefineResource` that initializes the property's resource descriptor.

Enumerated Xtypes

An enumerated xtype describes a widget property whose value range is restricted to a small set of values. For example, the xtype `UxXT_Boolean` describes a widget property whose value is either `0` or `1`.

To define an enumerated xtype, fill in the file `user-xtype.c`:

1. Include the public header file of the widget class.
2. Declare a global `UxXT_` variable of type `int` to hold the ID of the new xtype.
3. Define three `static` arrays. These arrays describe the values expected by the swidget, the values expected by the widget, and the names of constants defined for the values expected by the widget. These arrays are used for type conversion and code generation.
4. Define a converter function (only if the widget expects values of types other than `int` or `unsigned char`). The reference page for `UxAddConv` in Appendix G, "Ux Builder Functions," describes the required format.

   If the widget expects values of type `int` or `unsigned char`, you can use the built-in converters `UxStringToIntEnum` and `UxStringToCharEnum`.
5. Define new ValuesOf and Validator functions for the new xtype.

6. Add a call to UxAddEnumType in the function UxAddUserDefEnum-Types to register your new xtype with UIM/X. The value returned by UxAddEnumType is stored in the UxXT_ variable declared in step 2 above.

Example

The Square widget class declares a new property called MajorDimension. The possible values of this property are SquareWIDTH and SquareHEIGHT (these two constants are defined in Square.h). The definition of a new xtype for this property is contained in the file *uimx_directory*/contrib/DogAndSquare/user-xtype.c.

First, the public header file for the widget class is included:

```
#include "Square.h"
```

A global variable is then declared to hold the ID of the new xtype:

```
int UxXT_MajorDimension;
```

Next, three static arrays are defined for the new xtype:

```
static char *uMajorDimension[] = {
   "width", "height"
   };

static int xMajorDimension[] = {
   SquareWIDTH, SquareHEIGHT};
static char *dMajorDimension[] = {
   "SquareWIDTH", "SquareHEIGHT"};
```

- uMajorDimension contains the string values accepted by a swidget.
- xMajorDimension contains the values accepted by the widget.
- dMajorDimension contains the constants defined by Motif for the possible property values.

These three arrays are passed to UxAddEnumType. The ValuesOf and Validator functions defined for the new xtype refer to the array uMajorDimension.

The ValuesOf function defines the option menu given to the property in the Property Editor.

```
int UxValuesOfMajorDimension( list, n )
   char ***list;
   int *n;
{
   *list = uMajorDimension;
   *n = XtNumber(uMajorDimension);
   return ( *n );
}
```

The Validator function simply checks that a supplied value is one of the strings contained in uMajorDimension:

```
int UxValidateMajorDimension( sw, val )
   swidget sw;
   char *val;

{
   int i, n = XtNumber(uMajorDimension);
   if ( val != NULL )
      {
   for (i = 0; i < n; i++)
   {
      if ( strcmp( val, uMajorDimension[i] ) == 0 )
         return ( NO_ERROR );
   }
}
   return ( ERROR );
}
```

Note: The resource descriptor for the property majorDimension contains pointers to the UxValuesOfMajorDimension and UxValidateMajorDimension functions. These function pointers are stored in the resource descriptor by UxDefineResource. See *uimx_directory*/contrib/DogAndSquare/square.cl.c.

Finally, the new xtype definition is registered with UIM/X by calling UxAddEnumType:

```
void UxAddUserDefEnumTypes()
{
   UxXT_MajorDimension = UxAddEnumType(
      SquareNmajorDimension,
      sizeof(int),
      xMajorDimension,
      uMajorDimension,
      dMajorDimension,
      XtNumber(uMajorDimension),
      UxStringToIntEnum );
}
```

Note that this call to UxAddEnumType installs the built-in converter UxStringToIntEnum for the new xtype.

Non-Enumerated Xtypes

Non-enumerated xtypes can take on any value that can be stored in the data type of the widget property. Like enumerated xtypes, non-enumerated xtypes are defined by filling in the file user-xtype.c:

1. Declare a global UxXT_ variable of type int to hold the ID of the new xtype.
2. Define a converter function. The reference page for UxAddConv in
3. Appendix G, "Ux Builder Functions," describes the required format.
4. Define new ValuesOf and Validator functions, if necessary.
5. Add a call to UxAddXtype in the function UxAddUserDefTypes to register your new xtype with UIM/X. The value returned by UxAddXtype is stored in the UxXT_ variable declared in step 1.
6. Add a call to UxAddConv in the function UxAddUserDefTypes to register the converter function for the new xtype.

Overriding Inherited Class Methods

UIM/X operates on widgets through a suite of methods defined in the swidget classes. For example, when the user attempts to create a new widget as a child of an existing widget, a method is called on the proposed parent to verify that it can accept such a child. Some classes, such as drawingArea, accept most children. Other classes, such as scrolledWindow, can have only a fixed number of children. Each class has its own version of the method that implements the class-specific rules.

A swidget class inherits the methods of its superclasses. You can override these inherited class methods. Adding new class methods was discussed earlier in this chapter, in *Defining the Swidget Class.*

To override an inherited method, you must define a function and install it as the class method. This function can either augment or replace the existing class method. To augment a class method, use `Ux`*`Type`*`_get_op` to invoke a superclass method, and then execute any class-specific code:

```
#include "prim.cl.h"
void UxDogApply( swidget sw )
{
   /* Invoke superclass method */
   UxVoid_get_op( UxC_primitive, UxM_UxApply ) ( sw
   );
   /* Dog-specific stuff
   */printf("Woof ! Woof !\n");
}
```

If the function does not invoke the inherited method, then the derived class is effectively replacing the inherited method.

You install a function as a class method using the function `UxInit_method`:

```
UxInit_method( UxC_dog, UxM_UxApply, UxDogApply )
```

`UxInit_method` must be called from the function which registers the class (`UxRegister_dog`, in this example).

Note: A function installed as an overriding method must have the same return type and number and type of arguments as the inherited method.

Generating Code and Reading UIL

To generate code or load UIL code for a new widget class, you must extend the `uxcgen` and `uxreaduil` utilities. To do this, you must define the new widget class and its properties in the file `user-cg-cl.c`.

This file contains the stub function `CgInitUserDefWidgetClasses`. You enter the definition of a new class and its properties in this stub function as follows:

- Declare a `static` variable of type `WGT_CLASS_INFO`. The `WGT_CLASS_INFO` structure holds the definition of a new widget class. The following table lists the fields in the `WGT_CLASS_INFO` structure.

Field	Description
`char *name`	The name of the swidget class.
`char *filename`	The base name for the name of the public header file of the swidget class. This base name is put in initial caps, prefixed with `Ux`, and given a `.h` extension. For example, `dog` becomes `UxDog.h`.
`char *supername`	The name of the swidget superclass.
`char *xt_name`	The name of the widget class.
`char *uil_name`[1]	The name of the UIL class. The default name is `Xm`*ClassName*.
`char *xt_filename`	The public header file for the widget class.
`int dialog`	1 if this is a dialog, 0 otherwise.
`int num_resources`	The number of new properties (the number of elements in the array of `RES_INFO` structures).
`RES_INFO *resources`	The array of `RES_INFO` structures.

[1.] You set this field when you want to extend the uxreaduil utility.

- Declare a `static` array of type `RES_INFO`. The `RES_INFO` structure holds the definition of a property. This array must contain an element for each new property defined by the new widget class. Note that the `WGT_CLASS_INFO` structure contains a pointer to the array of `RES_INFO` structures. The following table lists the fields in the `RES_INFO` structure.

Field	Description
char *name	The name of the property (the string entered into the resource descriptor with the RD_NAME parameter).
char *xt_name	The Xt name of the property.
char *xt_name_def	The defined constant used in Xt code for this property. The default is XmN*propertyName*.
int callback	1 if this property is a callback, 0 otherwise.
int constraint	1 if this property is a constraint, 0 otherwise.
int rt_conv	1 if this property needs run-time conversion, 0 otherwise.
char *converter	The name of the converter function (for Xt code).
int xtype	ID of this resource's XTYPE.
int wgt_val	1 if this property is of type Widget, 0 otherwise.
int wgt_class_val	1 if this property is of type WidgetClass, 0 otherwise.
int pass	0, 1, or 2. Corresponds to the Pass field in the resource descriptor.
int res-type	The type of resource: NORMAL or SYNTHETIC.
char *list_count_res	The name of the property that counts the number of list items (for example, selectionArrayCount or listItemCount).
char *attach_res	The name of the attachment property associated with a widget-valued constraint resource.

- Add an extern declaration for each new xtype (the UxXT_*propertyName* variables) defined in user-xtype.c.
- Fill in the fields of a RES_INFO structure for each new property. Fill in the fields of the WGT_CLASS_INFO structure for the new widget class.
- Call CgEnterWidgetClassInfo. The address of the WGT_CLASS_INFO structure must be passed to CgEnterWidgetClassInfo.

The code shown below declares and fills in the RES_INFO and WGT_CLASS_INFO structures for the Dog widget class:

```
void CgInitUserDefWidgetClasses()
{
   static WGT_CLASS_INFO dog_class;
   static RES_INFO dog_res[3];
   dog_res[0].name = "barkTime";
   dog_res[0].xt_name_def = "DogNbarkTime";
   dog_res[1].name = "wagTime";
   dog_res[1].xt_name_def = "DogNwagTime";
   dog_res[2].name = "barkCallback";
   dog_res[2].xt_name_def = "DogNbarkCallback";
   dog_res[2].callback = 1;
   dog_class.name = "dog";
   dog_class.filename = "dog";
   dog_class.supername = "primitive";
   dog_class.xt_name = "dogWidgetClass";
   dog_class.xt_filename = "Dog.h";
   dog_class.num_resources = 3;
   dog_class.resources = dog_res;
CgEnterWidgetClassInfo( &dog_class );
}
```

Note that the Dog (and the Square) examples do not set the uil_name field of the WGT_CLASS_INFO structure. This is because the uxreaduil utility is not extended in these examples.

Note: The RES_INFO and WGT_CLASS_INFO structures are defined in *uimx_directory*/custom/include/class_info.h.

Building uxcgen

You can use the makefile *uimx_directory*/custom/src/Makefile to compile user-cg-cl.c and link it into an extended version of uxcgen. Note that if you define new xtypes, you must also compile and link the file user-xtype.c. See *Using the Custom Makefile*.

Building uxreaduil

You can use the makefile *uimx_directory*/custom/src/Makefile to compile user-cg-cl.c and link it into an extended version of uxreaduil. Note that if you define new xtypes, you must also compile and link the file user-xtype.c. See *Using the Custom Makefile*.

Note: There is no point in extending uxreaduil unless you have extended the UIL interpreter to handle the new widget class.

Extending the Ux Convenience Library

To use the Ux Convenience Library in code generated for a new widget class, you must register the properties that need run-time conversion. A property needs run-time conversion if the widget and the swidget expect different types.

You register properties for run-time conversion using UxDDInstall. The file user-rtime.c contains a stub function where you can add calls to UxDDInstall:

```
void UxAddRuntimeResources()
{
   extern int UxXT_MajorDimension;

   UxDDInstall( SquareNmajorDimension,
      UxUT_string,UxXT_MajorDimension );
   UxDDInstall( SquareNmakeSquare,
      UxUT_string,UxXT_Boolean );}
```

This code registers the two properties of the Square widget class for run-time conversion.

Building the Ux Convenience Library

You can use the makefile *uimx_directory*/`custom/src/Makefile` to compile `user-rtime.c` and replace it in the Ux Convenience Library. See *Using the Custom Makefile*.

Summary of Naming Conventions

The following table summarizes the naming conventions for the variables, data structures, and functions associated with a swidget class.

Name	Description
`Ux`*Class*`ClassPart`	The name of the `ClassPart` structure. This structure contains the new fields added to the class structure by a swidget class. For example, `UxDogClassPart`.
`Ux`*Class*`Class`	The name of the class structure. For example, `UxDogClass`.
`Ux`*Class*	The name of the instance structure. For example, `UxDog`.
`UxC_`*class*	The name of the `Class_t` variable that holds the swidget class ID returned by `UxRegister_class`. For example, `UxC_dog`.
`RD_`*propertyName*	A pointer to the resource descriptor for the property. For example, `RD_wagTime`.
`UxP_`*class*`RD_`*propertyName*	The class property ID returned by `UxFixed_class_prop`. For example, `UxP_DogRD_wagTime`.
`_`*MethodName*	The `vhandle` of a class method.
`UxM_`*MethodName*	A class method ID returned by `UxFixed_class_method`.
`UxRegister_`*class*	The function which registers the class and initializes the class structure. For example, `UxRegister_dog`.

Integrating Components 3

Overview

You can use UIM/X to build applications with components you have built yourself or purchased from other vendors. To do this, you must integrate your components with UIM/X. The integration procedure involves the following steps:

1. Preparing the integration code for your components.
2. Compiling the integration code.
3. Augmenting UIM/X with this compiled code.
4. Putting your components in a palette so you can distribute them to users.

To help you understand how to integrate components with UIM/X, this chapter provides a conceptual overview of the tasks performed by the integration code. It also explains how to augment UIM/X with the integration code and put your components in a palette.

"Writing Initialization Code for UIM/X" dissects the integration code for a typical component.

Understanding What to Do

Integrating a component with UIM/X is a lot like integrating a widget. To integrate a widget, you need a *swidget*. UIM/X uses swidgets to represent widgets. A swidget is a shadow widget—a widget's inseparable companion. UIM/X uses swidgets to hold the code and data it needs to manipulate widgets.

To integrate a component, you also need a swidget. In fact, you need a special kind of swidget called an *adapter swidget*. An adapter swidget connects UIM/X to the widgets in a component.

When you integrate a widget, you need to write the code that defines the swidget. You don't have to do this when you integrate a component. UIM/X includes a convenience function for creating adapter swidgets (see the reference page for `UxAdapterSwidget()` in Appendix G, "Ux Builder Functions").

What you do have to do is write some integration code that wraps the component in a UIM/X-compatible interface. UIM/X, via an adapter swidget, operates on the component through this interface.

The integration code presents your component to UIM/X as if it was actually developed in UIM/X. In other words, the integration code gives the illusion of being the generated code for a UIM/X Component.

This procedure is similar to augmenting UIM/X with the generated code for UIM/X Components (components created within UIM/X). The difference is that you must write, rather than generate, some integration code for each of your components.

Wrapping Components

A component defines a public interface consisting of a constructor and a suite of methods. The methods set and retrieve property values, perform operations on the component, and register event procedures (callbacks).

To integrate a component, you write both a C and a C++ version of a wrapper around the component. The C wrapper is a set of methods implemented using the UIM/X Method system. The C++ wrapper is a C++ class.

The wrappers give UIM/X and generated code a way to operate on the component. UIM/X and generated C code use the C wrapper and generated C++ code uses the C++ wrapper. Figure 3-1 shows how the wrapper code provides an interface between a component and both UIM/X and generated code.

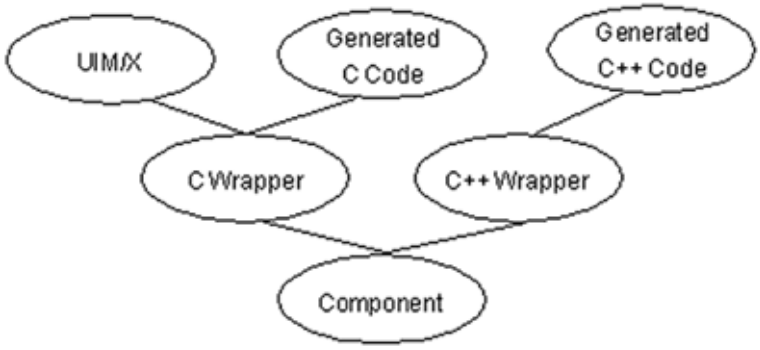

Figure 3-1 Wrapping a Component

Wrapping a component for integration with UIM/X involves these tasks:

- Writing the C wrapper constructor. This function is like the Interface Function of a UIM/X Component. In UIM/X, the name of this function is the value of an instance's Constructor property. UIM/X and generated code calls the wrapper constructor to create instances of the component.
- Writing the C++ wrapper constructor.
- Writing the wrapper methods. These are functions that wrap the real methods of the component, and are used in UIM/X and in generated C code.

  For some components, you will need to write wrapper methods that override methods inherited from the UxVisualInterface base class, such as `_set_x()`, `_get_x()`, `childSite()`, and `Manage()`.
- Getting a class code for the component. You need a class code to be able to register methods for the component in the UIM/X Method system.

  You use `UxNewInterfaceClassId()` to get a class code for your base component class, and `UxNewSubclassId()` to get class codes for its subclasses. This creates a class hierarchy in UIM/X that parallels your component class hierarchy. In particular, this allows methods to be inherited within UIM/X.
- Registering the wrapper methods and their signatures. You do this by calling `UxMethodRegister()` and `UxMethodSignatureRegister()` with the class code of the component.
- Defining the context structure used by UIM/X to create subclasses of the component.
- Defining the C++ wrapper class. This class is used in generated C++ code and when the wrapper implementation itself is compiled. The member functions of the class wrap the real methods of the component.

  For some components, you will need to write member functions that override virtual member functions inherited from the UxVisualInterface base class, such as `_set_x()`, `_get_x()`, `childSite()`, and `Manage()`.
- Defining C and C++ bindings. These bindings are macros whose definitions are conditional on the language being used.

The C bindings use the UIM/X Method system (`UxMethodLookup()`) to invoke the wrapper methods. The C++ bindings call member functions of the component's wrapper class. The binding macros are used by UIM/X and by generated code to set and retrieve properties and to manipulate the component.

Creating Adapter Swidgets

UIM/X handles all of the interface elements in a project using swidgets. Each component must have a constructor function that returns a swidget. When you build components inside UIM/X, this swidget is supplied by the generated code. For your other components, you must create a special adapter swidget to connect UIM/X to the Motif elements in your components.

An *adapter swidget* is a special class of swidget that represents an instance of a component in UIM/X. The adapter swidget holds on to the design-time (or run-time, for generated code) swidget information such as the class code used for method dispatch.

You obtain this class code by calling `UxNewInterfaceClassId()` or `UxNewSubclassId()`. You attach methods to the adapter swidget by registering methods against this class code.

To create an adapter swidget, you use `UxAdapterSwidget()`. This function requires a Motif widget (usually the controlling widget of the component) and a class code.

Managing Instances

The C wrapper constructor must not manage (in the Xt sense of the word) the widgets of the underlying component. UIM/X expects the C wrapper constructor to create the component, call `UxAdapterSwidget()`, and return an adapter swidget. At that point, the component should have created its widgets, but not managed them.

UIM/X manages the component by invoking the method `VisualInterface_Manage()` on the component. Components inherit a version of this method from the UxVisualInterface base class, but can provide their own version if required.

Designating a Child Site

Components that can accept children must define a `childSite()` method. This method designates a child site by returning the swidget whose widget can be used as the parent of the component's children.

A component's child site widget is usually the widget linked to the component's adapter swidget by `UxAdapterSwidget()`. If the child site widget is some other widget, you must create another adapter swidget for that widget.

Creating Instances of your Components

In UIM/X, you create an Instance when you reuse one interface in another interface. The interface being reused is called a Component, and each use of the Component is an Instance.

When you integrate your components with UIM/X, they can be used as Components too. The user can build an interface with your components and then reuse it by creating an Instance of it.

UIM/X calls the method `UxCanBeAnInstance()` to determine whether or not the user can create an Instance of a component. If `UxCanBeAnInstance()` returns False for a component, then the user can not create an Instance of that component. This means that any interface where the component is top-level cannot be reused as an Instance.

If `UxCanBeAnInstance()` returns True, or if the component has no such method, UIM/X lets the user create an Instance of the component. So if you don't want the user to create Instances of one of your components, you must define a method named `UxCanBeAnInstance()` for the component.

```
#ifdef DESIGN_TIME
int UxMessageDialog_UxCanBeAnInstance_Id = -1;
char* UxMessageDialog_UxCanBeAnInstance_Name
   ="UxCanBeAnInstance";

   static int
   _MessageDialog_UxCanBeAnInstance(swidget sw,
   Environment *pEnv)
{
   if (pEnv)
      pEnv->major(CORBA::NO_EXCEPTION);
   return 0;
}
#endif
```

Defining Design-Time Methods

In addition to writing wrapper methods for a component's own methods, you may have to implement some methods for UIM/X to use internally during design-time.

The adapter swidget forwards some design-time actions to the underlying component by translating them into methods. These design-time methods are `UxCheckChildren()`, `UxDrawHandles()`, and `UxObjectToRecreate()`:

- `UxCheckChildren()` determines whether or not the parent can accept the proposed children. By default, the method rejects children if the parent does not have a `childSite()` method.

  ```
  char *UxCheckChildren(swidget parent, Environment
     *pEnv,int nkids, Class_t *classes, swidget
     *kids);
  ```

  If `parent` can accept children, `UxCheckChildren()` returns `NULL`. Otherwise, it returns an error message. The adapter relays the design-time method `UxWidgetCannotAcceptChildren()` to this component method.

- `UxDrawHandles()` draws selection handles on the component. By default, it draws the selection handles on the widget passed to `UxAdapterSwidget()`.

  ```
  void UxDrawHandles(swidget adapter, Environment
     *pEnv);
  ```

  The adapter relays the design-time method `UxDrawHandles()` to the component method of the same name.

- `UxObjectToRecreate()` specifies the object to recreate when the user edits one of the children of a component. By default, it returns the adapter swidget for the component.

  ```
  swidget UxObjectToRecreate(swidget
     adapter,Environment *pEnv, swidget parent);
  ```

  The adapter relays the design-time methods `UxRecreateSwidget()` and `UxRecreateParentOrChild()` to this component method.

You use the convenience function `UxAdapterDesignMethods()` to register one or more of these methods. See the reference page for

`UxAdapterDesignMethods()` in Appendix G, “Ux Builder Functions.”

Overriding the Geometry-Handling Methods

The UxVisualInterface base class is an abstract base class that defines accessor methods for handling instance geometry, both during design time and in generated code. It defines set and get accessors for the `x`, `y`, `height`, and `width` properties of an instance. These properties appear as Core properties in the Property Editor.

When you integrate components with UIM/X, you need to override these accessor methods with versions that let the component handle geometry in its own way.

Adding Event Procedures

Components provide methods for registering event procedures. From the point of view of a component, an event procedure is like a property whose value happens to be a function pointer. This function will be called when the component recognizes that an event has occurred. The function must have the signature expected by the component. For example, the signature for a KeyDown event might be:

```
typedef void (*VwKeyDownEventProcedurePtr_t)
   (VwGuiComponent *comp, void *user_data, short
   *key, short state);
```

You could add event procedures as properties by defining set and get accessors, but this would force the user to define external functions and then enter function pointers directly in the Property Editor. A more elegant approach is to give users access to an editor such as the UIM/X Callback Editor.

Giving users a Callback Editor for event procedures is easy, but you must follow this rule: all event procedures defined in UIM/X must have the standard Xt callback signature:

```
typedef void (*XtCallbackProc)(Widget wid,
   XtPointer client_data, XtPointer call_data);
```

If your component has an event procedure with a different signature, like the KeyDown example, you must write a wrapper event procedure in the integration code to bridge the gap between Xt-style callback procedures and the actual event procedure defined by the component.

You can do this because the `call_data` argument of an `XtCallbackProc` is meant to be a structure holding whatever arguments a particular callback requires. So for an event procedure with special arguments, the wrapper event procedure will transfer these arguments into a call data structure and pass them along to the user's callback. For the KeyDown event, the callback structure would contain the fields `key` and `state`.

You install the wrapper event procedure, not the callback function defined in the Callback Editor, on the component. When the event occurs, the component calls the wrapper event procedure, which composes a call to the user's callback function.

This explanation leaves several important questions unanswered. For example, how do you install a wrapper event procedure? And how does UIM/X know to use the Callback Editor for a given property? Finally, where does the wrapper event procedure store the `XtCallbackProc` pointer? After all, doesn't the wrapper event procedure need this pointer every time an event occurs?

The answers to these questions define what you must do to give a component an editable callback property in UIM/X:

1. To add an event procedure as a Behavior property, define a special type of accessor method called a callback accessor. Callback accessors are named `Add`*EventName*`Proc`(). In this method, you register an event procedure with the component.

Note: You *must* use the `Add`*EventName*`Proc()` naming convention to define a callback accessor. UIM/X examines the name of a method to determine whether or not it is a callback accessor method.

- Define a callback structure to hold the arguments passed to the event procedure by the component.
- Write the wrapper event procedure registered by the callback accessor. This procedure stores the arguments received from the component in the callback structure and then composes a call to the user's callback function (exactly how this is done is explained below). The callback structure is passed as call data to the callback.

Defining the Event Procedure

By defining a callback accessor named `Add`*EventName*`Proc()`, you are telling UIM/X that the component has a property named *EventName*, and that this property belongs in the Behavior category of the Property Editor. UIM/X automatically makes the Callback Editor available for the property.

The job of the callback accessor is to install a wrapper event procedure on the component. The method accepts an `XtCallbackProc` pointer and a client data pointer, both of which must be passed on to the wrapper event procedure.

- The `XtCallbackProc` pointer is the callback defined by the user in the Callback Editor.
- The client data pointer is the context structure for the interface. You *must* pass this on to the user's callback. UIM/X uses it to give the user access to the interface-specific variables and the swidgets in the interface.

Both the callback and the client data must somehow be passed on to the wrapper event procedure. If the component allows you to pass user data into the event procedure, you can store the callback and the client data in a structure and pass it as the user data. For example, consider the following callback accessor:

```
static void _TextBox_AddKeyDownEventProc(swidget
   UxThis,Environment *pEnv, XtCallbackProc proc,
   void *cd)
{
   VwText *pCmpnt =
      (VwText*)UxGetComponentRef(UxThis);
   if (pEnv)
      pEnv->major(CORBA::NO_EXCEPTION);
   if (pCmpnt) {pCmpnt->PutKeyPressEvent(
      (VwKeyPressEventProcedurePtr_t)XkKeyPressEv
         entHandler,
      XkPackageEventHandlerData(proc, cd, pCmpnt)
   );
}
}
```

This method sets an event procedure by calling the component method `PutKeyPressEvent()`. The first argument is the event procedure, and the second the user data.

The user data is obtained from `XkPackageEventHandlerData()`, which allocates a structure and stores the callback and the client data in it:

```
typedef struct evh_data {
   XtCallbackProc proc;
   void* clientData;
} XkEventHandlerData;
```

`XkPackageEventHandlerData()` also installs a destroy callback on the component (which is passed as the third argument) to free this user data structure.

An alternative to passing the callback and client data as user data would be to use the X context manager. The callback accessor would store the callback and client data, and let the wrapper event procedure retrieve it later.

Defining a Callback Structure

You need to define a callback structure for each event type that passes special arguments to the user's event procedure. In your wrapper event procedure, you use the callback structure to pass arguments (as call data) to the user's callback function. For example, the following callback structure is defined for the KeyDown, KeyUp, and KeyPress events of a TextBox:

```
typedef struct xk_key_cb_data {
   unsigned char char_code;
   short key;
   short state;
} XkKeyEventCallbackData;
```

Writing the Event Procedure

You need a wrapper event procedure only if the event procedure's signature does not match `XtCallbackProc`. The job of the wrapper event procedure is to build a callback structure containing its arguments and then call the given callback.

For example, consider `XkKeyDownEventHandler()`, the wrapper event procedure for the TextBox's KeyPress event:

```
void XkKeyDownEventHandler(VwGuiComponent*
   vwcomp,void* cd, short ascii_char, short *key,
   short state)
{
   XkEventHandlerData* ehd =
      (XkEventHandlerData*)cd;
   XkKeyEventCallbackData cbd;
   if (ehd) {
      /*
      * Store the arguments in the callback
         structure */
      cbd.char_code = 0;
      cbd.key = *key;
      cbd.state = state;
      /*
      * Call the callback passed by the
      * AddKeyDownEventProc * method, and pass
         along the
      * original client data. */
      (*(ehd->proc))(vwcomp->GuiTarget(),
         ehd->clientData,(void*)&cbd);
      *key = cbd.key;
   }
}
```

The expression `vwcomp->GuiTarget()` is just this component's way of getting the widget that received the event. The argument `key` is a pointer because the user can modify it in the callback.

Generating Integration Code

To better understand how to integrate a component with UIM/X, you may wish to start by integrating a component built in UIM/X.

UIM/X provides a code generation option called Ux Integration Code, so you can integrate generated C++ classes with UIM/X.

When you have a project that involves a lot of Components, you can use an overnight build process to integrate finished Components into UIM/X. This keeps the size of the project down, making it easier to load, edit, and test.

When you finish a Component, you save its `.i` file and then remove it from the project. The overnight build process converts the `.i` file to C++ code (using `uxcgen`), compiles it, and then links it into UIM/X.

Another reason for integrating a Component into UIM/X is to make it available to other developers. Typically this is done by integrating the Component into UIM/X, creating an Instance, and then putting it in a Palette.

If you intend to generate C++ code for projects and interfaces that use the Components you integrate with UIM/X, it makes sense to generate C++ code for the Component.

But UIM/X is written in C, so to link a generated C++ class with UIM/X, you need extern C wrappers. But that's not all. You also need some special integration code to fit the generated class into the design-time framework of UIM/X. The Ux Integration Code option generates this special integration code.

These options are not normally available on the standard Code Generation Options dialog. (Figure 3-2 shows both Standard and Advanced Code Generation Options dialogs). In order to display the options required, you must merge two Builder Engine resources into the current X-resource database.

The C wrappers make a C++ class callable from a C program. The Ux Integration Code allows UIM/X to manage the component.

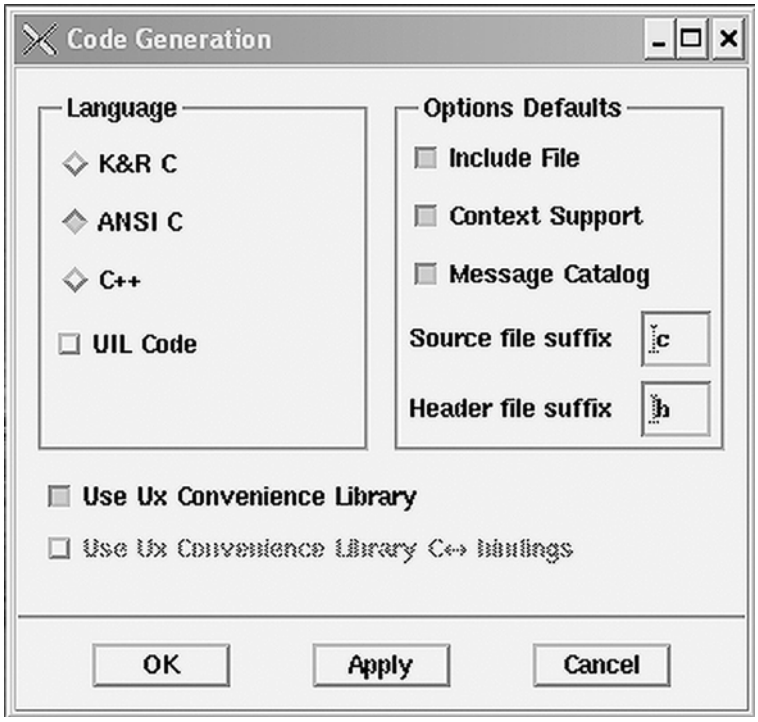

Figure 3-2 Standard UIM/X Code Generation Options Dialog

These advanced C++ code generation options become available in the UIM/X Code Generation Options when `UxPrjOptionsCGenGenCWrappers` and `UxPrjOptionsCGenGenUxIntCode` are set to `true`. It is simply a matter of merging the above resources into the current X-resource database prior to starting UIM/X, as follows:

1. Add the required Builder Engine resources to the resource database:

   ```
   xrdb -m
   Uimx3_0*UxPrjOptionsCGenGenCWrappers.set:true
   Uimx3_0*UxPrjOptionsCGenGenUxIntCode.set:true
   ```

   When you are through typing, press Ctrl-d to end your xrdb session.

2. Start UIM/X from your current directory::

   ```
   uimx &
   ```

Note: When you generate Ux Integration code, you must also use the Ux Convenience Library and select the Context Support option.

The constant `UX_C` controls whether or not the integration code is compiled. Note that to properly compile the integration code you need to define both `EXTERN_C_WRAPPERS` and `UX_C`. In fact, you can generate integration code only if you also generate extern C wrappers.

When you generate integration code for a class, you can either compile and link it with UIM/X or with generated code that uses the class:

- To link with UIM/X, compile the integration code with the `-DEXTERN_C_WRAPPERS`, `-DUX_C`, `-DDESIGN_TIME`, and -I*uimx_directory*`/custom/include` flags.

  The `-DDESIGN_TIME` flag is required for any code that you intend to link into the UIM/X executable. The -I*uimx_directory*`/custom/include` flag is necessary because UIM/X requires header files located in the `/custom/include` directory to compile the integration code.
- To link with generated C++ code that uses the class, compile the integration code without these flags.
- To link with generated C code that uses the class, compile the integration code with the `-DEXTERN_C_WRAPPERS` flag.

Note: A component that contains an instance of a second component cannot be integrated into UIM/X at the same time as the second component—UIM/X requires that both components be compiled with integration code, while the component containing the instance expects the instance component's code to be compiled normally.

The UIM/X distribution includes a makefile (*uimx_directory*`/config/Makefile.uimx`) for augmenting the UIM/X executable with object code. You use this makefile to link the integration code into UIM/X. See *Augmenting UIM/X* later in this chapter.

Writing the Integration Code

The previous section showed how to integrate interfaces designed in UIM/X back into UIM/X as components. Any class where a widget can be provided can be integrated with UIM/X. However, for these you must write the integration code yourself.

This section explains how to write the code that integrates your components with UIM/X. The approach taken is to explain by example, using the integration code for an actual CheckBox component. For this reason, the discussion is very specific about the details of files and source code.

You can choose to do some things differently, as long as you satisfy the essential requirement for integration. You must wrap your components in a software layer that matches the kind of API UIM/X generates for its own components. To do this, your integration code must contain the following elements:

- A C-callable constructor function that takes a parent swidget as its first argument and returns a swidget bound to the widget that represents the component.
- A set of C-callable methods registered through the UIM/X Method system, so that UIM/X can operate on the component. These include the `Manage()`, `childSite()`, and the set and get accessor methods.
- A context structure or class declaration (for C or C++ code, respectively) that can be subclassed by the kind of code UIM/X generates.
- A destroy callback on the widget bound to the component's swidget. UIM/X uses `XtDestroyWidget()` on this widget to destroy the component. You must attach a destroy callback to that widget (in the constructor) to free any extra data structures allocated by your component.

Writing the Header File

When you create a Component in UIM/X and generate code for it, a header file is generated. This header file is specified by an instance's `HeaderFile` property in UIM/X. To integrate an external component, you have to write its header file yourself. In this header file, you do the following:

- Include the required files.
- Define the C and C++ bindings for the wrapper methods.
- Define the context structure.
- Define a C++ wrapper class for the component.
- Declare the C wrapper constructor, if necessary. Abstract base classes don't have a wrapper constructor, since they are never instantiated directly.

Note: When you write the header file for a component, you use conditional compilation to create a common C and C++ header.

Including the Required Files

The header file for the base component class of your class hierarchy should include any general header files required by your components. You must also include the standard UIM/X headers:

```
#ifdef XT_CODE
#include "UxXt.h"
#else
#include "UxLib.h"
#include "uxproto.h"
#endif
```

The header `UxLib.h` contains the declarations and definitions for the Ux Convenience Library. `UxXt.h` is the equivalent header for Xt code, which is code that does not use the Ux Convenience Library.

The header file of a derived component class should also include the header file of its superclass.

Defining the C and C++ Bindings

The C and C++ bindings for a component's wrapper methods are two sets of macros. The C bindings are macros that expand to calls to `UxMethodLookup()`. The C++ bindings are macros that expand to calls to member functions of the component's C++ wrapper class. In this example, the `UX_C` macro controls whether the C or the C++ bindings are in force:

```
#ifdef UX_C
// #define C bindings
#else// #define C++ bindings
#endif
```

The C binding for a wrapper method is a macro that uses `UxMethodLookup()` to find and invoke the method. For example, the C binding for the Check Box's `_set_Alignment()` method is defined as follows:

```
#define CheckBox__set_Alignment(UxThis,pEnv,
   Value) \
   ((int(*)UXPROTO((swidget, Environment *,
      int)))\
   UxMethodLookup(UxThis,
      UxCheckBox__set_Alignment_Id,\
   UxCheckBox__set_Alignment_Name))( UxThis,
      pEnv, Value)
EXTERNC int UxCheckBox__set_Alignment_Id;
EXTERNC char* UxCheckBox__set_Alignment_Name;
```

All C binding macro definitions follow the same format:

```
#define Component_Method(UxThis, pEnv, Value) \
   ((int(*)UXPROTO((swidget, Environment *,
   long))) \
   UxMethodLookup(UxThis, UxComponent_Method_Id,\
   UxComponent_Method_Name)) \( UxThis, pEnv,
      Value)
EXTERNC int UxComponent_Method_Id;
EXTERNC char* UxComponent_Method_Name;
```

- *Component* is the name of the component.
- *Method* is the name of the wrapper method. This is the value assigned to `Ux`*Component*`_`*Method*`_Name` in the component's source file. For example, `_set_BackColor` is the name of the wrapper method that sets the component's `BackColor` property.
- `Ux`*Component*`_`*Method*`_Id` holds the method ID returned by `UxMethodRegister()`.
- `Ux`*Component*`_`*Method*`_Name` is the method name passed to `UxMethodRegister()`. This name is assigned in the component's source file.
- The `EXTERNC` macro controls the linkage of an identifier. For C++ code, it is defined as `extern "C"`. For C code, it is defined as `extern`.

The C++ binding for a wrapper method calls a member function of the C++ wrapper class:

```
#define Component_Method(c,e,v) \
(int)(((_UxCComponent*)UxGetContext(c))->Method(e
   ,v))
```

Defining the Context Structure

You need to define a stub context structure for each component:

```
#if UX_C

typedef struct {
   int classId;
} _UxCComponent;
#else
   // C++ wrapper class definition.
#endif
```

The context structure is used in generated C code. When you subclass the component, its context structure becomes the base part of the subclass's context structure. In generated C++ code, the context structure is replaced with a true C++ class.

Defining the C++ Wrapper Class

When you integrate components, you must create a hierarchy of wrapper classes that parallels the hierarchy of your component classes. For example, if component A is derived from component B, then the wrapper class for component A is publicly derived from the wrapper class for component B.

The wrapper class for your base component class must be derived from the UxVisualInterface base class. For example, the CheckBox wrapper class `_UxCCheckBox` is derived from `_UxCComponent`, which is derived from `UxVisualInterface`:

```
#include <vwcheck.hh>

class _UxCCheckBox : public _UxCComponent
{
protected:
    _UxCCheckBox () {}
public:
    _UxCCheckBox (swidget parent, String
   ObjectName);

   swidget _create_CheckBox()
      { return UxThis; }
   VwCheck* CheckBox()
      {return (VwCheck*)XkThisComponent; }
// …inline accessors…
}
```

The UxVisualInterface class is the base class for every component or interface class in UIM/X. It defines set and get accessor methods for the `x`, `y`, `height`, and `width` properties. It also defines the `Manage()` and `childSite()` methods. Subclasses of the UxVisualInterface base class inherit these methods and can override them as required.

Each wrapper class defines member functions that wrap the real methods of the corresponding component:

- A public member function that returns `XkThisComponent`, cast to the appropriate type. The data member `XkThisComponent` is the class pointer for the underlying component. It is inherited from the base wrapper class (see "Defining the Base Wrapper Class" on page 70).

  For example, the wrapper class for the Check Box defines the following member function:

```
VwCheck* CheckBox()
    { return (VwCheck*)XkThisComponent; }
```

This member function is used by the member functions that invoke the component's methods.

- Public member functions that wrap the component's real methods (the inline accessors). These member functions are invoked by the C++ bindings defined earlier in the header file.

  For example, the following member function sets the `Alignment` property of a Check Box:

```
int _set_Alignment(Environment*, int value) {
   return ((int) CheckBox()->PutAlignment(
      (VwToggleAlignment)value));
}
```

  The name of the member function is the same as the name of the C language wrapper method registered with the UIM/X Method system. Note that the `Environment` pointer is the first argument because this method is CORBA 1.1.

Defining the Base Wrapper Class

In addition to the member functions that wrap the component methods, the base wrapper class also defines the following data members and member functions:

- A public data member `XkThisComponent`. At run-time, this data member stores the class pointer of the underlying component. The C++ wrapper constructor sets `XkThisComponent` after it creates the underlying component.

  Each derived wrapper class provides a member function for accessing this data member and casting it to the appropriate type.
- A constructor that initializes `XkThisComponent`.
- A destructor that destroys `XkThisComponent`.
- A version of the `childSite()` method inherited from the UxVisualInterface base class.
- Versions of the `x`, `y`, `width`, and `height` property accessor methods inherited from the UxVisualInterface base class.

Defining a Derived Wrapper Class

In addition to the member functions that wrap the component's methods, a wrapper class derived from the base wrapper class also defines the following member functions:

- A public, default constructor that does nothing.
- A public constructor. The constructor definition is placed in the integration source file.
- A public member function that returns `UxThis` (the adapter swidget). The data member `UxThis` is inherited from the UxBase base class. The wrapper class constructor sets `UxThis` after it creates the underlying component.

Note: Abstract classes do not require any constructors.

Declaring the C Wrapper Constructor

The header file must contain a declaration for the C wrapper constructor. Note that you must use the `extern "C"` linkage when compiling under C++.

```
EXTERNC int create_CheckBox_ClassId
     UXPROTO((void));


EXTERNC swidget create_CheckBox UXPROTO((swidget
   parent, string Objectname));
```

The function `create_CheckBox_ClassId()` is used by the C wrapper constructor, and by subclasses of this class, to get a class code and to register the wrapper methods.

Writing the Source File

In the source file for a component, you do the following:

- Include the required files.
- Write the wrapper methods.
- Write the C++ wrapper constructor.
- Write the C wrapper constructor.
- Register the wrapper methods.

Including the Required Files

The source file for a component includes the following files:

- The header files required by the underlying component.
- The wrapper header file for the component.
- The UIM/X header files `veos.h` and `uxmethod.h`. These header files are required only for design-time code.

Writing the Wrapper Methods

For each component method that you want to expose to UIM/X (including property accessor methods), you must write a wrapper method. A wrapper method is a static function that you register using `UxMethodRegister()`. This function calls the corresponding method of the component itself:

```
static int _CheckBox__set_Alignment(swidget
   UxThis,
   Environment *pEnv, int val)
{
   VwCheck *pCmpnt =
      (VwCheck*)UxGetComponentRef(UxThis);
   if (pEnv)
      pEnv->major(CORBA::NO_EXCEPTION);
  if (pCmpnt) {
      return((int) pCmpnt->PutAlignment(
         (VwToggleAlignment)val));
}
return ERROR;
}
```

This example is the set accessor for the Check Box's `Alignment` property. It illustrates the essential elements of a wrapper method:

- The return type of the wrapper method corresponds to the return type of the component method.
- Like all UIM/X methods, the first argument is a swidget (the adapter swidget). The `Environment` pointer is the second argument because this method is CORBA 1.1. Subsequent arguments are values passed to the component method. For example, a set accessor method takes a third argument, which is the property value.
- You use `UxGetComponentRef()` to get the component reference (a pointer to the component) from the adapter swidget, and then cast it to the appropriate type.
- You set the `_major` field of the `Environment` structure to `NO_EXCEPTION`.
- You invoke the component method using the component reference.

For each wrapper method, you must also define the method Id and Name variables that you declared in the component's header file:

```
int UxCheckBox__set_Alignment_Id = -1;

char* UxCheckBox__set_Alignment_Name =
   "_set_Alignment";
```

The Id variable holds the value returned by `UxMethodRegister()` when you register the method. Initialize this variable to `-1`. The Name variable holds the name of the method. You pass this variable to `UxMethodRegister()` when you register the method.

By convention, the name of the function that implements the wrapper method is derived from the name of the wrapper method. However, the name of the function is not important—it is the value of the Name variable that identifies the method in the UIM/X Method system.

The names of property accessors must be `_set_`*Property* and `_get_`*Property*, where *Property* is the name that appears in the Property Editor. Other wrapper methods use the same name as the component method.

Understanding the Wrapper Constructors

The *constructor* is the function that creates instances of a component. You need two constructors: a C wrapper constructor (an Interface Function) for UIM/X and generated code and a C++ wrapper constructor.

At design time, the C wrapper constructor does the work. At run time, the C wrapper constructor invokes the C++ wrapper class constructor. The design-time C wrapper constructor must return the adapter swidget that connects UIM/X to the widgets in the component.

Both constructors accept the same arguments. The first argument is the swidget parent of the component. This argument is always required. Subsequent arguments are property values passed to the constructor of the underlying component. In UIM/X, these properties appear as Core properties in the Property Editor.

Writing the C++ Wrapper Constructor

The C++ wrapper constructor creates the underlying component as a member of the wrapper class. You use conditional compilation to define the wrapper class constructor only for run-time code:

```
#ifndef DESIGN_TIME
_UxCCheckBox::_UxCCheckBox (swidget parent,
   String ObjectName)
{
   Widget widgetParent;

   widgetParent = (parent == NULL) ?
      XkCreateImplicitShell(ObjectName) :
         UxGetWidget(parent);
   VwContainer *vwparent =
      VwGetContainerAdaptor(widgetParent);
   XkThisComponent = new VwCheck (ObjectName,
      vwparent,VW_DEF_X, VW_DEF_Y, VW_DEF_WIDTH,
      VW_DEF_HEIGHT,VwFalse);
   VwCheck *real_VwCheck = CheckBox();
   if (parent == NULL && widgetParent != NULL) {
```

```
        //
        // Install destroy callback to free implicit
           shell when
        // component is destroyed.//
    XtAddCallback(real_VwCheck->GUI(),
        XmNdestroyCallback,
        (XtCallbackProc) XkDestroyImplicitShell,
           widgetParent);
    }
    UxThis = XkAdapter (parent,
        create_CheckBox_ClassId(),this,
        XkThisComponent);
}
#endif /* DESIGN_TIME */
```

The wrapper class constructor for the Check Box has to do more than just create the component and get an adapter swidget for it. The process involves some additional steps that may or may not apply to your own components:

1. Determine the parent widget. A CheckBox component must have a parent widget, so it is parented to a shell widget if no parent is given. This handles the case when the user creates a primitive component on the desktop.

   The function `XkCreateImplicitShell()` creates a topLevelShell widget and returns it.
2. Get the corresponding component for the parent widget. This parent component is passed to the component's class constructor.
3. Create an instance of the component. Store the pointer in `XkThisComponent`.
4. Install a destroy callback to free the implicit shell when the component is destroyed.
5. Get a class code for the component and register its methods. The CheckBox constructor does this by calling the function `create_CheckBox_ClassId()`. To understand what this function does, see "Registering the Methods" on page 79.
6. Get an adapter swidget by calling `XkAdapter()`. Use `this` to pass a pointer to the wrapper class object. Store the adapter swidget returned by `XkAdapter()` in `UxThis`. See "Wrapping UxAdapterSwidget()" on page 77.

Writing the C Wrapper Constructor

By convention, the C wrapper constructor is named `create_`*Component*, where *Component* is the name of the component. When you put an instance of a component in a palette, you use this name as the value of the instance's `Constructor` property.

You use conditional compilation to define the C wrapper constructor so that it uses the wrapper class constructor at run time:

```
swidget create_CheckBox (swidget parent, String
   ObjectName)
{
#ifdef DESIGN_TIME
   Widget widgetParent;
   widgetParent = (parent == NULL) ?
      XkCreateImplicitShell(ObjectName) :
         UxGetWidget(parent);
   VwContainer *vwparent =
      VwGetContainerAdaptor(widgetParent);
   VwCheck *real_VwCheck = new VwCheck
      (ObjectName, vwparent,VW_DEF_X, VW_DEF_Y,
      VW_DEF_WIDTH, VW_DEF_HEIGHT,VwFalse);
if (parent == NULL && widgetParent != NULL) {
   //
   // Install destroy callback to free implicit
      shell when
   // component is destroyed.
   //
   XtAddCallback(real_VwCheck->GUI(),
      XmNdestroyCallback,(XtCallbackProc)
      XkDestroyImplicitShell, widgetParent);
}
return XkAdapter(parent,
   create_CheckBox_ClassId(), 0,real_VwCheck);
#else
```

```
    _UxCCheckBox *uxc_CheckBox = new _UxCCheckBox
        (parent,ObjectName);

    return uxc_CheckBox->_create_CheckBox();
#endif
}
```

The C wrapper constructor is the C-callable version of the C++ wrapper class constructor. It does what the wrapper class constructor does, with one exception: it passes 0 to `XkAdapter()` as the wrapper class pointer.

At run time, the C wrapper constructor creates an `_UxCCheckBox` object. The function `_create_CheckBox()` retrieves the adapter swidget, which the wrapper class constructor stores in the data member `UxThis`.

Wrapping UxAdapterSwidget()

The CheckBox example uses the function `XkAdapter()` as a wrapper for `UxAdapterSwidget()`. This wrapper function takes care of setting things up properly so both UIM/X and generated code can work with the CheckBox component.

The four arguments are the parent swidget, the class code, a pointer to the wrapper class object, and a pointer to the component. (In the generated code, the wrapper class object is the context.)

While the code in `XkAdapter()` is specific to the CheckBox, it is a good example of what you have to do:

- Get the principal widget of a component, and get an adapter swidget for this widget by calling `UxAdapterSwidget()`:

  ```
  Widget it = vwcomp->GUI();

  sw = UxAdapterSwidget(it, parent, XtName(it),
     clsCode,vwcomp, UxNO_CONTEXT);
  ```

- Check whether or not the component can be an instance by calling its `UxCanBeAnInstance()` method.

```
#ifdef DESIGN_TIME
   int (*canBeAnInstance)( swidget, void *) =
      NULL;
   canBeAnInstance = (int (*)(swidget, void *))
      UxMethodLookup( sw, -1,
      "UxCanBeAnInstance");
   if (canBeAnInstance)
   {
      /* If it cannot be an instance, then we must
         mark
       * it as a shell.
       */
      if (!(*canBeAnInstance)( sw, &UxEnv))
      {
         XkSetShell( sw);
   }
}
```

Note: A *top-level* component is either a shell widget or a widget with an implicit shell. An example of a widget with an implicit shell is the FileSelectionBoxDialog widget, which consists of a DialogShell with a FileSelectionBox widget as its child.

- Add a callback to the widget to destroy the component when the widget is destroyed:

```
#ifdef DESIGN_TIMEXt
   AddCallback(it, XmNdestroyCallback,
      (XtCallbackProc)
      XkComponentWidgetDestroyed, vwcomp);
   if (XtIsShell(it)) {XtVaSetValues(it,
      XmNdeleteResponse, XmUNMAP, NULL);
   }
#else
   XtAddCallback(it, XmNdestroyCallback,
```

```
(XtCallbackProc) XkComponentWidgetDestroyed,
uxc_if);
```

Note: During design time, some shells cause UIM/X to exit when they are destroyed, so `XmNdeleteResponse` must be set to `XmUNMAP` if the widget is a shell.

- At run time, store a pointer to the wrapper class object with `UxPutContext()`:

  ```
  UxPutContext (sw, (void *) uxc_if);
  ```

  This makes the wrapper class object the context in generated code. (Hint: look for calls to `UxGetContext()` in the generated code.)

- Install a callback on the component to clear `XkThisComponent` (the component pointer stored in the wrapper class) when the component is destroyed:

  ```
  vwcomp->AddDestroyCallback(
      (VwCallbackProcedurePtr_t)XkComponentDestroyed
         ,(void *)uxc_if);
  ```

- Return the adapter swidget obtained from `UxAdapterSwidget()`.

Note: One last thing to note about `XkAdapter()` is the following line:

```
vwcomp->PutUserData(sw);
```

The CheckBox component has a user data field, which `XkAdapter()` uses to store the adapter swidget. This provides a way to get the adapter swidget for a given component.

Registering the Methods

You must register the methods of your component so that UIM/X can call them from C code. To do this, you use a function separate from the wrapper constructor. This allows subclasses to check that their base class methods are registered.

You use a separate function to register the methods. This function contains a one-time block of code that gets the class code and then registers the methods:

```
int create_Derived_ClassId( void )
   {
   static int IfClassCode = -1;
   if (IfClassCode == -1)
   {
      IfClassCode = UxNewSubclassId
         (create_Base_ClassId())
   // Register methods.
}
   return IfClassCode;
}
```

You use `UxNewSubclassId()` to get a class code for the component class. `UxNewSubclassId()` accepts one argument, which is the class code for the component's base class. To get this class code, you simply call the base class's `ClassId()` function.

For the root class of your hierarchy, you call `UxNewInterfaceClassId()` instead of `UxNewSubclassId()`. `UxNewInterfaceClassId()` registers a class as a subclass of the `UxVisualInterface` base class.

Once you have a class code, you can register methods against that code with UxMethodRegister():

```
UxCheckBox__set_Alignment_Id =
   UxMethodRegister(IfClassCode,UxCheckBox__set_A
   lignment_Name,(void (*)
   ())_CheckBox__set_Alignment);
```

As well, if you want your methods to be available from the Connection Editor, you must register the method's signature using `UxMethodSignatureRegister()`:

```
UxMethodSignatureRegister(ifClassCode,
   UxCheckBox__set_Alignment_Name,
   UxCreateMethodSignature(UxCheckBox__set_Alignm
      ent_Name,Corba1, "int", UxEnvArgResource(),
      UxGetArgResource("value", UxUT_int,
      "VwAlignLeft",
         XkValidateVwToggleAlignment,
         XkValuesOfVwToggleAlignment),
      NULL));
```

Writing Initialization Code for UIM/X

Some of the integration code for your components has to take care of initializing UIM/X. This code makes the Interpreter aware of your components and customizes the Property Editor.

The standard way of doing this is to put all the initialization code in one function, and then call it from the main program file used to augment UIM/X.

Loading Header Files

You *must* load the integration header files for your components into the Interpreter during UIM/X initialization. Otherwise, the definitions and declarations they contain will not be known to UIM/X.

To load a component's integration header file, you call `UxLoadGlobalInclude()` with the name of the header file:

```
UxLoadGlobalInclude("xkcheck.h");
```

You call `UxLoadGlobalInclude()` from `main()` in the UIM/X main program file *uimx_directory*`/config/uimx_main.cc`. This ensures that the header files are loaded before any palette files, so palette files can use symbols defined in the header files.

Because `UxAppInitialize()` initializes the Interpreter, you can not call `UxLoadGlobalInclude()` before `UxAppInitialize()`.

Registering Functions

You *must* register the wrapper constructors with the Interpreter during UIM/X initialization. Otherwise, these functions will not be linked into the augmented UIM/X executable.

To register a wrapper constructor, you call `UxRegisterFunction()` with the name of the function and the function pointer:

```
UxRegisterFunction("create_CheckBox",
   create_CheckBox);
```

You call UxRegisterFunction() from UxRegisterFunctions() in the UIM/X main program file *uimx_directory*/config/uimx_main.cc.

Installing Option Menus and Resource Editors

You use the functions `UxInstanceResource()` and `UxGlobalInstanceResource()` to install option menus and resource editors for instances of your components. In UIM/X, these functions support properties added by defining set and get accessor methods on the component.

`UxInstanceResource()` installs an option menu or resource editor for a given property of a given component.

`UxGlobalInstanceResource()` installs an option menu or resource editor for a property with a given name. Every property with that name gets the same option menu or resource editor. If two components have a property with the same name, both properties get the same option menu or resource editor.

For example, suppose two components have an `Alignment` property, but each component defines a different set of possible values for the property. If you use `UxGlobalInstanceResource()` to install an option menu, both Alignment properties get the same option menu.

See Chapter 2, "Integrating Widgets," for more information on installing option menus and registering resource editors.

Augmenting UIM/X

Once you have prepared the integration code for your components, you are ready to build an augmented UIM/X. By augmenting UIM/X, you give users the ability to create instances of your components.

To build an augmented UIM/X, you use the template makefile *uimx_directory*`/ config/Makefile.uimx`

1. Create a working directory.
2. Copy *uimx_directory*`/config/Makefile.uimx` to your working directory. Rename the file to Makefile, and use `chmod` to make the file writable.

3. Copy any required source files to your working directory. For example, you may have customized the UIM/X main program file (by modifying the template file *uimx_directory*/`config/uimx_main.cc`).
4. Edit the makefile macros:

   a. Use `AUGEXEC` to change the name of the augmented UIM/X executable. The default name is `uimx_aug`.

   b. Use `AUGMAIN` to change the name for the UIM/X main program file. The default name is `uimx_main.cc`.

   c. Use `APPL_OBJS` to list any C object files, such as the main program file, that you want to link into the augmented executable.

   d. Use `APPL_CPLUSOBJS` to list any C++ object files you want to link into the augmented executable. For example, if you did not compile your integration code into a library, you would use `APPL_CPLUSOBJS` to list the object files for your integration code.

   e. Use `EXTRA_CFLAGS` to add C compiler flags.

   f. Use `EXTRA_CPLUSFLAGS` to add C++ compiler flags.

   g. Use `EXTRA_LDFLAGS` to add linker flags.

   h. Use `EXTRA_UXLIBS` to list any libraries you want to link into the augmented executable. For example, you use `EXTRA_UXLIBS` to list the design-time library of integration code and the implementation library for your components.
5. Use `touch` to ensure that all dependent files are more recent than their targets.
6. Invoke `make` using the name of the augmented executable as the target.

Building a Palette

Once you have integrated your components with UIM/X, you need to give users a way to create instances of these components. You do this by building a palette that contains an instance of each component.

Before you start creating instances, you need to create a new palette (or open an existing one), and decide what categories you need. For example, you may want to divide your components into categories such as Primitives, Dialogs, and Managers.

Note: Refer to the *UIM/X User's Guide* for more information on creating and editing palettes.

Creating Instances

After you decide where you are going to put your components and create any new categories or palettes, you can start creating instances:

1. Make sure there are no selected interfaces.
2. Create an empty instance by selecting CreateSubclass from the Project Window menu bar. (When you create a Subclass without first selecting an interface, you actually end up creating an empty instance—an instance that has no component.)

   In the augmented UIM/X, your components exist only as compiled code. To create an instance of one of these components, you fill in the Declaration properties of an empty instance. These properties define the component for an instance.
3. Double-click on the instance to load it into the Property Editor.
4. Select Declaration from the Category option menu.
5. Enter the name of the component in the `Component` property.
6. Enter the name of the component's header file in the `HeaderFile` property.
7. Enter the arguments to the component's constructor in the `ArgDefinition` property. The `ArgDefinition` property is a string of declarations:

   ```
   "swidget parent; char *name;"
   ```
8. Enter the properties and callbacks in the `PropDefinition` property. The `PropDefinition` property is a string of declarations, one for each accessor property or callback:

   ```
   "int x; int y; int wid; int h; void
      (*ClickEvent)();"
   ```

Note: Sometimes you may not want to give the user access to a property or event of a component. To hide properties and events, omit them from the string of declarations you enter in the `PropDefinition` property. The user sees only the properties and events specified by the `PropDefinition` property.

---

9. If the component can accept children, enter the class of the component's child site in the `ChildSiteClass` property.
10. Click Apply.

Putting Instances in the Palette

You now have an instance to put in the palette. But before you do, you might want to give it an icon and a name. Otherwise, you will get the same icon and name shown in the Interfaces Area for a subclass.

1. Select Compound from the Category option menu.
2. Set `CompoundIcon` to the name of the file containing the icon you want to use.
3. Set `CompoundName` to the name you want to appear in the Palette. Use the underscore character (_) to break a long name across two or more lines. This only works if the UIM/X resource `splitPalIconNames` is set to `true` (its default value). Note also that you can truncate long names by changing `shortPalIconNames` from `false` (its default value) to `true`.
4. Click Apply. You are now ready to put the instance in the palette.
5. Drag the instance to the palette and drop it in the appropriate category. Continue for each component until you have a full palette.

Building Executables

4

Overview

This chapter describes how to customize and build UIM/X executables using the makefiles supplied with UIM/X. These makefiles are generic templates that you can use to update executables and libraries.

By changing the macro definitions in a copy of a makefile template, you can quickly adapt a makefile to new requirements. In most cases, you have only to change the macros that specify the names of files and paths.

The following makefiles are discussed in this chapter:

- *uimx_directory*/`custom/src/Makefile`

  This makefile template updates executables and libraries that depend on the source files in *uimx_directory*/`custom/src`.

- *uimx_directory*/`build/src/Makefile`

  This makefile template updates executables and libraries that depend on the source files in *uimx_directory*/`build/src`.

- *uimx_directory*/`config/Makefile.uimx`

  This makefile augments the UIM/X executable by linking it with object code from other applications.

- *uimx_directory*/`mkinclude/central.mk`

  This makefile contains the rules and additional macro definitions required by *uimx_directory*/`config/Makefile.uimx` and several `contrib` makefiles.

Using the Custom Makefile

Use the makefile *uimx_directory*/`custom/src/Makefile` when you want to do one of the following:

- Make the library `libuxcustom.a`. The library must be recompiled when:
    - You modify copies of the source files found in the directory *uimx_directory*/`custom/src`. There are many reasons for modifying the code in one or more of these files. For example, you can change UIM/X's Create menus by modifying the code in `cr-menus.c`, or you can hard-code UIM/X resources by inserting code in `uimx-conf.c`.
    - You want to integrate a new widget class with UIM/X.
- Make a new UIM/X executable. Do this when:
    - You modify *uimx_directory*/`custom/src/uimx_main.cc`.
    - You want to link in a new version of the library `libuxcustom.a`.
- Make an extended version of the `uxcgen` utility. When you integrate new widget classes with UIM/X, you must extend `uxcgen` so that it can generate code for the new widget classes.

  This involves modifying the files `user-cg-cl.c` and `user-xtype.c` in the directory *uimx_directory*/`custom/src`.
- Make an extended version of the `uxreaduil` utility. As with the `uxcgen` utility, you would do this to support new widget classes.
- Make a new version of the `libux.a`, the Ux Convenience Library. You must re-build this library when you integrate a new widget class with properties that require run-time conversion.

Note: UIM/X must be compiled under ANSI mode.To customize UIM/X with K&R code, modify the custom makefile uimx_directory/`custom/src/Makefile` so that UIM/X is compiled under ANSI mode and the K&R code is compiled under K&R mode.

Note: If the SoftBench Encapsulator is not installed on your system, remove the flag `-DUSING_SOFTBENCH` from the `SB_CFLAGS` macro before using the custom makefile.

Custom Makefile Macros

The makefile template *uimx_directory*/custom/src/Makefile contains a number of macros that you can redefine when you use the makefile. The following are some common reasons for changing a macro definition:

- To change the file name of an executable.
- To specify the object files used to update a library or executable.
- To adapt the makefile to a different directory structure.

The following table defines the macros that you are most likely to want to modify. You should also examine the makefile itself.

Macro Name	Definition
UIMXDIR	UIM/X's home directory.
UX_CFLAGS	Used to specify the path or directory containing the header files included by the source files in *uimx_directory*/custom/src.
LIBUXCUSTOM	The copy of libuxcustom.a being updated. Note that the makefile template assumes this file is in the current directory.
LIBUXBUILD	The copy of libuxbuild.a linked with the UIM/X executable. Change this macro definition to link an updated version of the library. For example, you may have used *uimx_directory*/build/src/Makefile to make a new libuxbuild.a.
LIBUXRUNTIME	The copy of libux.a being modified. Note that the makefile template assumes that this file is in the current directory.
EXECUTABLE	The name of the UIM/X executable. Change this macro definition if you want another name for your executable.
CGEN	The name of the code generation utility.
READUIL	The name of the utility that reads UIL files.
MAIN	The name of the source file containing the main() function that initializes UIM/X.
WIDGET_OBJECTS	A list of the object files for new widget classes being integrated with UIM/X.

Macro Name	Definition
SWIDGET_OBJECTS	A list of the object files for new swidget classes.
BE_OBJECTS	A partial list of the object files in libuxcustom.a. You can remove references to files that are unchanged.
SB_CFLAGS	Defines the constant USING_SOFTBENCH. If your system does not have the SoftBench include files, you should replace this macro's definition with "SB_CFLAGS = ". Otherwise you may get errors if you recompile any of the files UxIo.c, UxSbinit.c, UxSbutils.c, and UxSbappl.c.
CUSTOM_OBJECTS	A list of the object files used to update libuxcustom.a. The macros WIDGET_OBJECTS, SWIDGET_OBJECTS, and BE_OBJECTS are expanded and added to this list. You can remove references to files that are unchanged.
RUNTIME_OBJECTS	A list of the object files used to update libux.a. You can remove references to files that are unchanged. To archive the run-time object files for components in the run-time library, add the names of the object files to this list.
CPLUS_OBJECTS	A list of the object files compiled in C++.

Invoking Make on the Custom Makefile

The following table lists the targets in *uimx_directory*/custom/src/Makefile whose names are not macro values. These targets are constant and are always valid arguments for make, which will always update the same file(s) each time it is passed one of these target names.

For example, consider the target and rule lines shown below (these lines are taken from *uimx_directory*/custom/src/Makefile). The command make executable updates the UIM/X executable no matter what value is assigned to the EXECUTABLE macro.

```
executable: $(EXECUTABLE)


$(EXECUTABLE): libuxcustom $(MAINOBJ) $(UXOBJ)
    @echo "***** Linking $(EXECUTABLE)"$(CPLUS)
    $(LDFLAGS) $(MAINOBJ) $(UXOBJ) $(LIBS) -o \
    $(EXECUTABLE)
```

Contrast this with the command `make uimx`, which only updates the executable if the macro `EXECUTABLE` is set to `uimx`. If you change the value of `EXECUTABLE` to something other than `uimx`, there is no longer any target named `uimx` in the makefile.

In the following table, entries such as `$(EXECUTABLE)` refer to the value of the macro whose name is enclosed in parentheses.

Target Name	File(s) Updated
libuxcustom	libuxcustom.a[1]
libux	libux.a
executable	$(EXECUTABLE)
cgen	$(CGEN)
readuil	$(READUIL)
all	libuxcustom.a $(EXECUTABLE) libux.a $(CGEN) $(READUIL)

1. Invoking make with no arguments will make the library `libuxcustom.a`.

General Procedure for Using the Custom Makefile

The general procedure for using the makefile template *uimx_directory*`/custom/src/Makefile` is as follows:

1. Create a working directory.
2. Copy *uimx_directory*`/custom/src/Makefile` to your working directory.
3. Copy *uimx_directory*`/custom/src/uimx_main.cc` to your working directory.
4. Copy any source files and header files in *uimx_directory*`/custom/src` or *uimx_directory*`/custom/include` that you intend to modify to your working directory. Edit the files as required.
5. If you are integrating a new widget class, copy the `.cc` and `.h` files for the new widget and swidget classes to your working directory. Modify the makefile macros `WIDGET_OBJECTS` and `SWIDGET_OBJECTS` to list the object files for the new widget and swidget classes.

To ensure that `make` properly handles the file dependencies, you should list the files in the following format:

```
WIDGET_OBJECTS = \
    $(LIBUXCUSTOM)(Dog.o)
SWIDGET_OBJECTS = \
    $(LIBUXCUSTOM)(dog.cl.o)
```

6. Copy the libraries `libuxcustom.a` and `libux.a` to your working directory from *uimx_directory*`/lib`. Note that the makefile template already refers to the local copies of these libraries.
7. Execute the command `touch *.cc`. This ensures that the source files are more recent than the libraries.
8. Execute the command `make all`.

Using the Build Makefile

Use the makefile *uimx_directory*`/build/src/Makefile` when you want to do one of the following:

- Make the library `libuxbuild.a`. The library must be recompiled when you modify copies of the files in *uimx_directory*`/build/src` or *uimx_directory*`/build/include`.

  These files control the interfaces and widget classes linked into the UIM/X executable.
- Make a new UIM/X executable. You would do this when:
  - You modify *uimx_directory*`/build/src/uimx_main.cc`.
  - You want to link in a new version of the library `libuxbuild.a`.

Note: UIM/X must be compiled under ANSI mode.

Build Makefile Macros

The makefile template *uimx_directory*/build/src/Makefile contains a number of macros that you can redefine when you use the makefile. The following are some common reasons for changing a macro definition:

- To change the name of the UIM/X executable.
- To specify the object files used to update a library or executable.
- To adapt the makefile to a different directory structure.

  The following table defines the macros that you are most likely to want to modify. We suggest you also examine the makefile itself.

Macro Name	Definition
UIMXDIR	UIM/X's home directory.
LIBUXBUILD	The copy of libuxbuild.a being updated. Note that the makefile template assumes this file is in the current directory.
LIBUXCUSTOM	The copy of libuxcustom.a linked with the UIM/X executable. Change this macro definition to link an updated version of the library. For example, you may have used *uimx_directory*/custom/src/Makefile to make a new libuxcustom.a.
MAIN	The name of the source file containing the main() function that initializes UIM/X.
EXECUTABLE	The name of the UIM/X executable. Change this macro definition if you want another name for your executable.
BUILD_OBJECTS	A list of the object files in libuxbuild.a. You can remove references to files that are unchanged.

Invoking Make on the Build Makefile

The following table lists the target in *uimx_directory*/build/src/Makefile whose names are not macro values. These targets are constant and are always valid arguments for make, which will always update the same file(s) each time it is passed one of these target names.

Entries such as $(EXECUTABLE) refer to the value of the macro whose name is enclosed in parentheses.

Target Name	File(s) Updated
libuxbuild	libuxbuild.a[1]
executable	$(EXECUTABLE)
all	libuxbuild.a $(EXECUTABLE)

[1] Invoking `make` with no arguments will make the library `libuxbuild.a`.

General Procedure for Using the Build Makefile

The general procedure for using the makefile template *uimx_directory*`/build/src/Makefile` is as follows:

1. Create a working directory.
2. Copy *uimx_directory*`/build/src/Makefile` to your working directory.
3. Copy *uimx_directory*`/build/src/uimx_main.cc` to your working directory.
4. Copy any source files in *uimx_directory*`/build/src` that you intend to modify to your working directory. Copy the corresponding header files for these source files to your working directory from *uimx_directory*`/build/include`. Edit the files as required.
5. Copy the library `libuxbuild.a` to your working directory from *uimx_directory*`/lib`. Note that the makefile template already refers to the local copy of this library.
6. Execute the command `touch *.cc`. This ensures that the source files are more recent than the library.
7. Execute the command `make all`.

Augmenting UIM/X

The UIM/X executable can be augmented with the object code of other applications. In particular, you can compile code generated by UIM/X and link it into the UIM/X executable.

Linking object code with UIM/X gives the Interpreter access to the functions in the object code. The Interpreter can execute any compiled function (or method) contained within the UIM/X executable.

Augmenting UIM/X allows you to:

- Simplify the development of an interface for an application program. You can design the application's interface, insert calls to the compiled application functions, and test the interface, all without having to exit UIM/X.
- Link in Components distributed in object form.
- Link interfaces into the development environment.

  Large projects have many interfaces. As individual interfaces are finished, you can remove them from the project and make them part of the UIM/X development environment. As the project progresses, there will be fewer interfaces to load, edit, and test.

  To do this, you generate the code for the interface, compile it, link it with UIM/X, and remove the interface from the project (but keep a backup copy of the interface's `.i` file).

  UIM/X allows you to mix compiled and interpreted code, so you can still test the entire project—the interfaces you load and create interactively can call the create functions of the interfaces that exist only as object code.
- Use a compiled interface as an editor within UIM/X. Suppose you use UIM/X to create a specialized widget editor. You can make this editor a part of UIM/X by compiling its generated code and linking it with UIM/X.

The makefile template *uimx_directory*`/config/Makefile.uimx` allows you to augment UIM/X with C and C++ object files and libraries.

Note: The object code should not contain a `main()` function. Any initialization required by the application can be done from within the `main()` function in *uimx_directory*`/config/uimx_main.cc`. If you need to access the internal data structures of the swidget classes in augmented UIM/X, you must make sure that the symbol `PRIVATE_SWIDGET` is defined when you compile UIM/X. You can do this by adding the flag `-DPRIVATE_SWIDGET` to the `cflags` resource or to one of the makefile macros in `Makefile.uimx`.

Registering Functions

The file *uimx_directory*`/config/uimx_main.cc` contains the function `UxRegisterFunctions`. You register a function with the Interpreter by inserting a call to `UxRegisterFunction` in `UxRegisterFunctions`.

Registering a function has two advantages:

- It makes the address of the function known to the Interpreter, eliminating the delay associated with looking up the function the first time it is encountered.
- It ensures that functions from X, C, or other libraries are included in the executable, and are thus accessible from the Interpreter.

`UxRegisterFunction` is declared as follows:

```
void UxRegisterFunction(char *name, void *fptr);
```

The parameter `name` is the name of the function, and `fptr` is a pointer to the function.

When you register a function, you must also declare it. (If a function is not referenced, it will not be linked into the UIM/X executable.) You can do this by including the appropriate header file in `uimx_main.cc`, or adding an `extern` declaration. The following example illustrates both approaches:

```
#include <math.h>

extern char *yourFunction(void);
void UxRegisterFunction()
{
   UxRegisterFunction("sin", sin);
   UxRegisterFunction("yourFunction",
   yourFunction);
}
```

Note: Ensure that the function you are preregistering has been declared with its proper linkage. A C function must be declared as `extern "C"`.

Registering Globals

The file *uimx_directory*/config/uimx_main.cc contains the function UxRegisterGlobals. You register a global with the Interpreter by inserting a call to UxRegisterGlobal in UxRegisterGlobals.

Registering globals has the same advantages as registering functions.

Note: UIM/X preregisters the globals in the C library that are part of the ANSI standard. To use any other globals in the C library, you must register them with the Interpreter.

UxRegisterGlobal is declared as follows:

```
void UxRegisterGlobal(void *name, void *gptr);
```

The parameter name is the name of the variable, and gptr is a pointer to the variable.

When you register a global, you must also declare it. You can do this by including the appropriate header file in uimx_main.c, or adding an extern declaration. The following example illustrates the second approach:

```
{   extern int yourGlobal;
    void UxRegisterGlobal();
}   UxRegisterGlobal("yourGlobal",
    &yourGlobal);
```

Note: Ensure that the global you are preregistering has been declared with its proper linkage. A C global must be declared as extern "C".

Conditional Compilation in Generated Code

When you compile generated code and link it with UIM/X, you may want to avoid certain function calls. A good example is XtCloseDisplay. Calling this function during testing will terminate the UIM/X session. You can use the DESIGN_TIME symbol to control compilation:

```
#ifndef DESIGN_TIME
XtCloseDisplay(UxDisplay);
#endif
```

When you use `Makefile.uimx` to augment UIM/X (see below), this symbol is defined.

Using Makefile.uimx

If you examine *uimx_directory*`/config/Makefile.uimx`, you will see that the makefile contains a limited number of macro definitions. The rules and additional macro definitions required to build an augmented UIM/X are contained in the makefile *uimx_directory*`/mkinclude/central.mk`, which is included at the end of `Makefile.uimx`.

The macros in *uimx_directory*`/config/Makefile.uimx` define the target and dependent files for augmenting the UIM/X executable. The following table describes these macros.

Macro Name	Definition
`AUGEXEC`	The name of the augmented UIM/X executable. In *uimx_directory*`/mkinclude/central.mk`, `$(AUGEXEC)` is the target that builds an augmented UIM/X.
`AUGMAIN`	The object file for the main program file of the augmented executable.
`APPL_OBJS`	The list of C object files to be linked with UIM/X.
`APPL_CPLUSOBJS`	A list of C++ object files to be linked with UIM/X.
`EXTRA_CFLAGS`	Use this macro to define extra C compiler options required for compiling the files `$(APPL_OBJS)`. By default, this macro sets the `-DDESIGN_TIME` flag. Generated code must be compiled with the `-DUIMX_INTERNAL` flag to make an interface into an editor in UIM/X. You can also use this macro to add the `-DPRIVATE_SWIDGET` flag.
`EXTRA_CPLUSFLAGS`	Use this macro to add C++ compiler flags.
`EXTRA_LDFLAGS`	Use this macro to define any extra link editor options required for linking object code with UIM/X.
`EXTRA_UXLIBS`	Use this macro to list the libraries you want linked into the UIM/X executable.

General Procedure for Using Makefile.uimx

The general procedure for using the makefile *uimx_directory*`/config/Makefile.uimx` is as follows:

1. Create a working directory.

2. Copy *uimx_directory*/config/Makefile.uimx to the file Makefile in your working directory. Renaming the makefile allows you to invoke make without specifying the name of the makefile.
3. Copy the file *uimx_directory*/config/uimx_main.cc to your working directory. Insert any required initialization code in uimx_main.cc. The comments in uimx_main.cc indicate where such code should be inserted.
4. Copy the source (or object) files you want to compile and link with UIM/X to your working directory.
5. Modify the makefile macros described in the above table. Use the macros to name the executable and to list the object file for each source file in your working directory.
6. If you want to make an interface into an editor in UIM/X, compile the interface's generated code with the flag -DUIMX_INTERNAL.

   To do this, add -DUIMX_INTERNAL to the EXTRA_CFLAGS macro as follows:

```
EXTRA_CFLAGS = -DDESIGN_TIME -DUIMX_INTERNAL
```

7. Use touch to ensure that all the files the target depends on are more recent than the target.
8. Invoke make. Use the value of the macro AUGEXEC to specify the target.

Using central.mk

The makefile *uimx_directory*/mkinclude/central.mk contains the rules and additional macro definitions required to augment UIM/X. This makefile is included by Makefile.uimx.

The target and rule lines in central.mk that build an augmented executable are shown below:

```
$(AUGEXEC): $(APPL_OBJS) $(APPL_CPLUSOBJS)
    $(UIMXOBJ) $(CPLUS)\ $(LDFLAGS)
    $(EXTRA_LDFLAGS) -o $@ $(APPL_OBJS)\
    $(APPL_CPLUSOBJS) $(UIMXOBJ) $(LIBS1)
```

The macros AUGEXEC, APPL_OBJS, APPL_CPLUSOBJS, and EXTRA_LDFLAGS are defined in *uimx_directory*/config/Makefile.uimx. See *Augmenting UIM/X*.

The other macros are defined in `central.mk`. The following table describes some of the macros which you can edit to tailor the compilation and linkage of an augmented executable.

Macro Name	Definition
`UIMXOBJ`	UIM/X's object files and libraries.
`LIBS1`	X, Motif, and other libraries linked into an augmented executable.
`LIBS2`	X, Motif, and other libraries linked into an application executable.

Compound Properties

A

CanBeTopLevel

Determines whether the widget can be a top level widget.

CanHaveChildren

Determines whether the widget can have additional children. The widget will not accept any more children after the property `CanHaveChildren` is set to `false` and applied.

ClipboardOps

Determines whether the widget can be cut, copied, and pasted. Note that a widget can only be cut if it is deletable.

CompoundEditorName

Determines the name of a compound widget's specialized editor.

CompoundIcon

Determines the icon used to represent the compound widget. The value of this property must be the name of the file containing the pixmap or bitmap of the icon. Valid file formats are X11 bitmap and XPM.

CompoundName

Determines the name given to a compound widget. This name is displayed on the palette.

CompoundResourceSet

Allows you to create design-time properties for the individual widgets in a compound widget.

CompoundSwidgetMethodSet

Allows you to create design-time swidget methods for the individual widgets in a compound widget.

DragRecursion

Determines the direction in which UIM/X traverses the compound widget hierarchy when looking for a dragable widget. UIM/X only checks the value of this property if the region widget is not dragable.

Editor

Allows you to enter the callback which pops up the compound editor. This callback function is called whenever you do one of the following:

- Create an instance of the compound widget.
- Double-click the Select mouse button on one of the widgets in the compound widget.
- Select the Compound Editor item from a menu.

EditorClientData

Determines the client data to be passed to the callback function that pops up the compound editor.

Note that when you install a compound editor, the value of the property `CompoundName` identifies the compound editor on UIM/X's menus. For example, if you give the name Radio Box to a compound widget, the menu item Compound Editor becomes Radio Box Editor for the compound widget.

IsAlignable

Determines whether the widget can be aligned with other widgets. If `IsAlignable` is set to `false` for at least one of the selected widgets, the Align menu is insensitive.

IsAreaSelectable

Determines whether the widget can be selected using range selection. If the widget is included in a range of selected widgets, UIM/X will disallow the selection. A widget is only area-selectable if it is selectable (see `IsSelectable`).

IsArrangeable

Determines whether the widget can be arranged with other widgets. If `IsArrangeable` is set to `false` for at least one of the selected widgets, the Arrange menu is insensitive.

IsCompound

Determines whether the widget is a compound widget; used to set the top widget in a compound.

IsDeletable

Determines whether the widget can be deleted. Note that a widget can only be cut if it is deletable.

IsDraggable

Determines whether the widget can be dragged. Note that you can only move a widget if it is draggable.

IsDuplicatable

Determines whether the widget can be duplicated.

IsInCompound

Determines whether the widget is part of a compound widget.

IsMovable

Determines whether the widget can be moved. A widget can be dragged even if it is not movable. For example, you can drag and drop a widget in the Property Editor even if its `IsMovable` property is set to `false`. See also `IsReparentable`.

IsNovice

Determines whether the widget is built for UIM/X Novice Mode.

IsRecreatable

Determines whether the widget is recreatable.

IsRegion

Determines whether a widget is a region widget. UIM/X uses region widgets to determine whether the Adjust button was pressed on a move or a resize region.

IsReorderable

Determines whether you can change the order of a generation of children. When you view a widget tree in the Browser, a generation of children is ordered from top to bottom. You reorder a generation of children as follows:

1. Drag and drop widgets in the Browser. When you drop a child on its parent, the child goes to the bottom of the order.
2. Paste widgets in the Browser.
3. Use the exchange operations in the Menu Editor.

Note: Reordering changes the numeric order of a list of children. For example, the numeric order of items on a menu corresponds to their relative position—the first (or top) menu item, the second item, and so on. Adding a new menu item reorders all menu items below it.

IsReparentable

Determines whether the widget can be given a new parent.

IsResizable

Determines whether the widget can be resized.

IsSelectable

Determines whether the widget can be selected. When this property is set to `false`, you cannot select the widget. If a widget is not selectable, then it is not area-selectable.

ResizeRecursion

Determines the direction in which UIM/X traverses the compound widget hierarchy when looking for a resizable widget. UIM/X only checks the value of this property if the region widget is not resizable.

ShowInBrowser

Determines whether the widget is shown in the Browser. This property is typically set to `false` when you want to make a widget an invisible part of a widget hierarchy.

UsePropEditor

Determines whether the widget can be loaded into the Property Editor. Note that once `UsePropEditor` is set to `false` and you remove the widget from the Property Editor, you cannot load the widget back into the Property Editor.

You can override the UsePropEditor property with the toggle `UxPEEditAny`. Set `UxPEEditAny.set` to true and either restart or reset UIM/X.

A

Interface File Format

B

File Format Concepts

Object Instantiation

The first task that must be accomplished by the UIM/X Interface File Format (IFF) is the task of instantiating an object, for example a `pushbutton`, and setting its properties. The mechanism for this is to include a line of the following format:

```
*ObjectName.class: ObjectClass
```

and then set properties using the X Toolkit syntax for resource specifications:

```
*ObjectName.resource: value
```

Note: When the value of a resource requires more than one line, each line should be terminated with the backslash "\" character to indicate that the value continues on the next line.

The acceptable values for `value` are defined using the standard resource converters.

For example, to instantiate a `pushButton` and set some of its properties, the following would be used:

```
*pushButton.class: pushButton
*pushButton.parent: myrowColumn
*pushButton.x: 100
*pushButton.y: 200
*pushButton.width: 500
*pushButton.height: 600
*pushButton.labelString: "Push Me"
```

Instance-Specific and Proprietary Resources

Builders often have resources which are proprietary, or which only exist on specific instances of a widget. Examples of the former are the `createManaged` resource added to each widget to determine whether to create the widget managed or unmanaged, and the `allowShellResize` resource that UIM/X adds to certain types of managers to control resize behavior. An example of the latter is a constraint resource, which only exists as a resource of the widget when the widget is a child of, for example, a form widget.

Using the example above, to add a manage resource and a constraint resource to the `pushButton` would result in the following:

```
*pushButton.class:pushButton
*pushButton.parent: myrowColumn
*pushButton.x:100
*pushButton.y: 200
*pushButton.width: 500
*pushButton.height: 600
*pushButton.labelString: "Push Me"
*pushButton.createManaged: "true"
*pushButton.leftAttachment: "attach_form"
```

Note: Most instance specific resources are found in the Constraint category of the Property Editor.Most proprietary resources are found in the Compound and Declaration categories of the Property Editor.

Facets

It is frequently required to specify attributes of resources. These "properties of properties" are called facets.

For genuine widget resources, the possible facet values for a resource are the following:

- source facet

  Indicates where the generated code is to be placed. Specifying the value `public` indicates that the value is to be placed in a resource file. The value of the callback represents an expression yielding a pointer to a callback function (which may be external to the interface).

  Omitting the value `public` specifies that the value of the resource represents a body of code to be placed in a callback function to be generated.

- lock facet

  Specifies whether the resource should be considered locked by UIM/X.

  When a padlock symbol is displayed beside the text field where the user would type in the property value, the text field cannot be edited.

Interface-Specific Resources

The following resources, found once in each `.i` file, represent information that is global to the entire interface:

class	The class type of the object being described. It can be a widget name, an instance, a connection_action, or a connection_event, or a palette.
classconstructor *	The constructor code executed prior to creation of the GUI portions of the interface.
classdestructor *	The destructor code added after the interface GUI has been destroyed but before the class is destroyed.
classinc *	The declaration statements provided in the class includes fields of the class view of the Declarations editor.
classmembers *	Class member variables and functions.

*. Used only when generating C++ code. Fields where this information is entered by the user are in the class view page of the Declaration Editor.

classspec *	User- supplied class names to also use as parent of the interface class being defined.
defaultShell	The type of shell to provide for this component if it is to be created as a toplevel interface and the root of the widget hierarchy is not itself a shell.
gbldecl	The values as entered by the user in the global properties of the Declaration Editor.
ispecdecl	The declaration of the instance-specific variables as declared by the user in the Declaration Editor.
ispeclist	A comma-separated list of instance-specific variable names.
ispeclist.ispecname	For each variable name in `ispeclist`, a facet is generated that contains the decomposition of the variable's name and type. The builder derives this information from the source entered in `ispeclist`. All comments and blanks are removed.
funcdecl	The `create/popup` function declaration as entered by the user in the Declaration Editor. Note that it is legal for the user to include comments and conditional compilation switches in this value. Note also that the declaration of the `interface` function can have either prototypes (as in C++ function declarations) or a comma-separated list of arguments (as in K&R C).
funcname	The actual name of the `interface` function. The builder extracts this value from the `funcdecl` resource. This fields contains only an identifier. Any comments found in `funcdecl`are removed.
funcdef	A decomposition of the signature of the `interface` function. The builder extracts this information from `funcdecl`.

argdecl	A semicolon-separated list of `interface` function arguments. This list is derived from `funcdecl`. Comments surrounding the arguments are removed.
arglist	A comma-separated list of the names of the arguments of the `interface` function.
arglist.argname	For each argument of the `interface` function, a facet of the resource of `arglist` is created, defining the argument's name and type.
icode	The initial code executed prior to the creation of the GUI portion of the interface.
fcode	The final code executed after the creation of the interface.
auxdecl	Auxiliary functions supplied by the user.
ifacefunctype	The type of the interface creation function. Can be either `create`or `popup`.

Methods

Components are user-defined segments of GUIs that can be re-used, just as any widget can be re-used. Their behavior, defined by the user, is provided to UIM/X in the form of virtual methods.

These methods represent, not only behavior of the component, but also the declaration of an interface to the component. These interfaces are declared by defining a `get` and a `set` method.

methodType	The method return type as entered by the user.
methodArgs	The arguments of the method as entered by the user in the Method Editor.
methodBody	The body of the function as entered by the user.
methodSpec	The method specifier. Values can be `virtual` or `static`. Used in C++ to determine the kind of member function to generate.
accessSpec	The access specifier. Used in C++-generated code, determines the access to a method. Values are `public`, `protected`, or `private`.

corba	Determines the CORBA support in force for a particular method. Values are: `corba2`(`env` as last argument), `corba1`(`env` as second argument), and `none` (no `env` argument).
argument	A comma-separated list of the names of the arguments of the method.
argname.def	A facet, named after a formal argument of the method, defining the argument's name and type.

Note that some method names have special meaning for both the builder and the code generator (uxcgen):

_get_*Property*	A `Get` method (as seen in the Method Editor).
_set_*Property*	A `Set` method (as seen in the Method Editor).
Add*Callback*Proc	A callback accessor method. UIM/X creates a callback resource in instances of the component. The method is expected to have a callback signature.

Connections

The Connection Editor simplifies the user's task of specifying interface behavior by providing pick-and-choose mechanisms. To accomplish this, UIM/X saves the resulting information as objects, with the properties specifying individual aspects of the connections.

These objects are of the `connection_event` class and are parented to the source object of the connection. The source object indicates the presence of connections on the specific callback of a particular widget.

For each connection belonging to a particular `connection_event`, an object of `connection_action` is created. The resources of this object specify the particulars of the method and the actual arguments to be used for the connection.

The resources of a `connection_event` are: `class`, `name`, `parent`, and `callback`.

The following resources of the `connection_action` type are possible:

target	The widget that is the target of the connection.
method	The method to use for the particular connection

argument.argname	One facet, named after the formal argument it represents, giving the actual argument to use for this formal parameter.

Swidget Methods

The Connection Editor relies on the methods defined for a class (usually defined in the Method Editor) to present a choice of methods for the connection. Although this is suitable for classes built from UIM/X components, as the user will have defined some methods, it is less appropriate for Motif widgets.

Swidget methods are provided as an alternative. They define the methods as objects that supply small segments of code to be pasted into the generated callback code. In this sense, they are similar to in-line member functions in C++ or their equivalent macro-defined functions. Since the language options for code generation are not known when the interface file is generated, code segments are generated for all possible choices.

In the code segments, certain reserved words are replaced by the appropriate code to reference either the source or target widget (or swidget). The following

table indicates how the search strings are replaced. “Target” is the action’s target resource value and “return” is the action’s optional return resource value:

Search String	Ux Mode	Xt Mode
UxTargetSwidget	target	target
UxTargetWidget	UxGetWidget(target)	target
UxReturn	return	return

Note that there is no special keyword for the source of the connection. However in a callback, the variable Ux can always be used to refer to the source. Since connections are always expanded within a callback, UxThis is always available for this purpose.

Note also that the value of the UxReturn resource does not have to appear on the right-hand side of an assignment expression. UxReturn is expected to be an lvalue, consequently its address can be passed to a function. This could be used, for example, with XtGetValue in Xt code generation mode.

A swidget method can be used as many times as there are `connection_actions` on an interface, but only one swidget method object is written in the interface file. The swidget method object is shared among all `connection_actions`.

The following are resources specific to the `swidgetmethod` class object:

methodType	The method return type.
methodArgs	A semicolon-separated list of method arguments.
methodBody.Xt	The code body supplied for Xt code (C++, ANSI C, K&R C).
methodBody.Ux	The code body supplied for Ux code (C++, ANSI C, K&R C).
methodBody.C++	The code body supplied for C++ code with C++ bindings.
arguments	A comma-separated list of argument names of a method.
argname.def	The decomposition of the argument's name and it's type.

Loading Interface Files of an Earlier Version

In `.i` files of UIM/X version 2.9, private callbacks began with an open brace "{" and ended with a corresponding closing brace "}". These braces are no longer required, and are removed by UIM/X when loading a version 2.9 interface file.

Swidget Class Hierarchy

C

Overview

The hierarchy of swidget classes generally parallels the hierarchy of Motif widget classes. Figure C-1 indicates the depth of subclassing. For each swidget class, the corresponding widget class and the swidget class' private and public header files are shown.

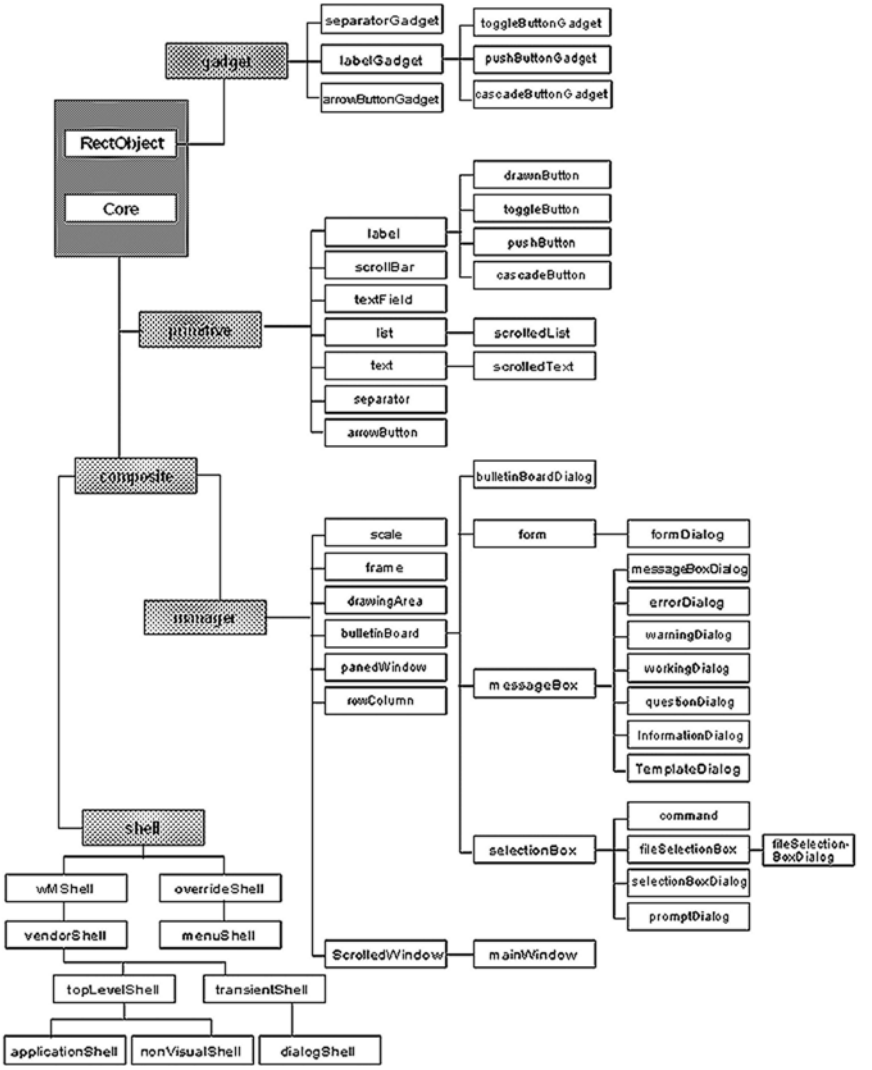

Figure C-1 Swidget Class Hierarchy

Swidget Class	Widget Class	Private Header	Public Header
applicationShell	applicationShellWidgetClass	applSh.cl.h	UxApplSh.h
arrowButton	xmArrowButtonWidgetClass	arrB.cl.h	UxArrB.h
arrowButtonGadget	xmArrowButtonGadgetClass	arrBG.cl.h	UxArrBG.h
bulletinBoard	xmBulletinBoardWidgetClass	bboard.cl.h	UxBboard.h
bulletinBoardDialog	xmBulletinBoardWidgetClass	bbD.cl.h	UxBbD.h
cascadeButton	xmCascadeButtonWidgetClass	cascB.cl.h	UxCascB.h
cascadeButtonGadget	xmCascadeButtonGadgetClass	cascBG.cl.h	UxCascBG.h
command	xmCommandWidgetClass	comm.cl.h	UxComm.h
composite	compositeWidgetClass	comp.cl.h	UxComp.h
Core	widgetClass	Core.cl.h	UxCore.h
dialogShell	xmDialogShellWidgetClass	dialSh.cl.h	UxDialSh.h
drawingArea	xmDrawingAreaWidgetClass	drArea.cl.h	UxDrArea.h
drawnButton	xmDrawnButtonWidgetClass	drawnB.cl.h	UxDrawnB.h
errorDialog	xmMessageBoxWidgetClass	errorD.cl.h	UxErrorD.h
fileSelectionBox	xmFileSelectionBoxWidgetClass	fsBox.cl.h	UxFsBox.h
fileSelectionBoxDialog	xmFileSelectionBoxWidgetClass	fsBD.cl.h	UxFsBD.h
form	xmFormWidgetClass	form.cl.h	UxForm.h
formDialog	xmFormWidgetClass	formD.cl.h	UxFormD.h
frame	xmFrameWidgetClass	frame.cl.h	UxFrame.h
gadget	xmGadgetClass	gadget.cl.h	UxGadget.h
informationDialog	xmMessageBoxWidgetClass	infoD.cl.h	UxInfoD.h
label	xmLabelWidgetClass	label.cl.h	UxLabel.h
labelGadget	xmLabelGadgetClass	labelG.cl.h	UxLabelG.h
list	xmListWidgetClass	list.cl.h	UxList.h
mainWindow	xmMainWindowWidgetClass	mainW.cl.h	UxMainW.h

Swidget Class	Widget Class	Private Header	Public Header
manager	xmManagerWidgetClass	mgr.cl.h	UxMgr.h
menuShell	xmMenuShellWidgetClass	menuSh.cl.h	UxMenuSh.h
messageBox	xmMessageBoxWidgetClass	msgBox.cl.h	UxMsgBox.h
messageBoxDialog	xmMessageBoxWidgetClass	msgBD.cl.h	UxMsgBD.h
nonVisualShell	xmTopLevelShellWidgetClass	nvSh.cl.h	UxNvSh.h
overrideShell	overrideShellWidgetClass	overSh.cl.h	UxOverSh.h
panedWindow	xmPanedWindowWidgetClass	paneW.cl.h	UxPaneW.h
primitive	xmPrimitiveWidgetClass	prim.cl.h	UxPrim.h
promptDialog	xmSelectionBoxWidgetClass	prompD.cl.h	UxPrompD.h
pushButton	xmPushButtonWidgetClass	pushB.cl.h	UxPushB.h
pushButtonGadget	xmPushButtonGadgetClass	pushBG.cl.h	UxPushBG.h
questionDialog	xmMessageBoxWidgetClass	questD.cl.h	UxQuestD.h
RectObject	rectObjClass	rectO.cl.h	UxRectO.h
rowColumn	xmRowColumnWidgetClass	rowCol.cl.h	UxRowCol.h
scale	xmScaleWidgetClass	scale.cl.h	UxScale.h
scrollBar	xmScrollBarWidgetClass	scrBar.cl.h	UxScrBar.h
scrolledList	xmListWidgetClass	scList.cl.h	UxScList.h
scrolledText	xmTextWidgetClass	scText.cl.h	UxScText.h
scrolledWindow	xmScrolledWindowWidgetClass	scrW.cl.h	UxScrW.h
selectionBox	xmSelectionBoxWidgetClass	selBox.cl.h	UxSelBox.h
selectionBoxDialog	xmSelectionBoxWidgetClass	selBD.cl.h	UxSelBD.h
separator	xmSeparatorWidgetClass	sep.cl.h	UxSep.h
separatorGadget	xmSeparatorGadgetClass	sepG.cl.h	UxSepG.h
shell	shellWidgetClass	shell.cl.h	UxShell.h
templateDialog	xmMessageBoxWidgetClass	templD.cl.h	UxTemplD.h
text	xmTextWidgetClass	text.cl.h	UxText.h
textField	xmTextFieldWidgetClass	textF.cl.h	UxTextF.h

Swidget Class	Widget Class	Private Header	Public Header
toggleButton	xmToggleButtonWidgetClass	togB.cl.h	UxTogB.h
toggleButtonGadget	xmToggleButtonGadgetClass	togBG.cl.h	UxTogBG.h
topLevelShell	topLevelShellWidgetClass	topSh.cl.h	UxTopSh.h
transientShell	transientShellWidgetClass	tranSh.cl.h	UxTranSh.h
vendorShell	vendorShellWidgetClass	vendSh.cl.h	UxVendSh.h
Swidget Class	Widget Class	Private Header	Public Header
warningDialog	xmMessageBoxWidgetClass	warnD.cl.h	UxWarnD.h
wMShell	wmShellWidgetClass	wmSh.cl.h	UxWmSh.h
workingDialog	xmMessageBoxWidgetClass	workD.cl.h	UxWorkD.h

C

Resource Types

D

Overview

In UIM/X, the values of most swidget properties are stored as integers and strings. The actual data types required by the widgets, however, are not necessarily the same as those used internally by UIM/X.

UIM/X provides a mechanism for converting between the different data types expected by swidgets and widgets. The resource descriptor of a property identifies the data type expected by the swidget, the data type expected by the widget, and the function used to convert between the two types. As well, the resource descriptor contains pointers to functions for validating and listing property values.

Utypes and xtypes are defined in

uimx_directory`/custom/include/utype.h`.

Utypes

A utype is a data type used to hold the value of a swidget property. A utype ID is stored in the resource descriptor of a swidget property. The following table lists the utype IDs defined by UIM/X and the corresponding data types.

Type ID	Data Type
UxUT_cardFunction	Cardinal (*) ()
UxUT_char	char
UxUT_float	float
UxUT_int	int
UxUT_long	long
UxUT_short	short
UxUT_string	char*
UxUT_stringTable	char**
UxUT_vhandle	char*
UxUT_visualPointer	Visual*
UxUT_voidFunction	void (*)()
UxUT_XmTextSource	XmTextSource

Xtypes

An xtype specifies the data type and possible values of a widget property. UIM/X uses xtypes to describe the property values accepted by a widget. An xtype ID is stored in the resource descriptor of a swidget property.

Enumerated Xtypes

An enumerated xtype is a data type with a list of possible values. For example, the enumerated xtype UxXT_ArrowDirection describes a widget property whose possible values are XmARROW_UP, XmARROW_DOWN, XmARROW_LEFT, or XmARROW_RIGHT. UIM/X converts enumerated xtypes to either UxUT_string or UxUT_int.

The following table lists the IDs of the enumerated xtype defined by UIM/X.

Enumerated Xtype IDs	
UxXT_Alignment	UxXT_MessageDialogType
UxXT_ArrowDirection	UxXT_MultiClick
UxXT_AudibleWarning	UxXT_MwmInputMode
UxXT_AttachmentType	UxXT_NavigationType
UxXT_Bool	UxXT_Orientation
UxXT_Boolean	UxXT_Packing
UxXT_ChildType	UxXT_ProcessingDirection
UxXT_ChildPlacement	UxXT_ResizePolicy
UxXT_ChildVerticalAlignment	UxXT_RowColumnType
UxXT_CommandWindowLocation	UxXT_ScrollBarDisplayPolicy
UxXT_EntryVerticalAlignment	UxXT_ScrollBarPlacement
UxXT_DefaultButtonType	UxXT_ScrollingPolicy
UxXT_DeleteResponse	UxXT_SelectionPolicy
UxXT_DialogStyle	UxXT_SelectionArray
UxXT_DialogType	UxXT_SeparatorType
UxXT_EditMode	UxXT_ShadowType
UxXT_FileTypeMask	UxXT_StringDirection
UxXT_IndicatorType	UxXT_TearOffModel
UxXT_InitialWindowState	UxXT_UnitType
UxXT_KeyboardFocusPolicy	UxXT_VisualPolicy
UxXT_LabelType	UxXT_WinGravity
UxXT_ListSizePolicy	

Non-Enumerated Xtypes

Non-enumerated xtypes can take on any value that can be stored in the data type of the widget property. The following table lists the non-enumerated xtypes defined by UIM/X. As well, for each non-enumerated xtype, the table lists the utype used to represent the property value. UIM/X installs converters for each xtype, utype pair listed. If there is no utype listed, it is because no conversion is required—the xtype and utype are the same data type.

Xtype ID	Utype Converted To
UxXT_Accelerators	UxUT_string
UxXT_Atom	UxUT_string
UxXT_bitmap	UxUT_string
UxXT_BorderPixmap	UxUT_string
UxXT_BottomShadowPixmap	UxUT_string
UxXT_char	–
UxXT_Colormap	UxUT_long
UxXT_CreatePopupChildProc	UxUT_voidFunction
UxXT_Dimension	UxUT_int
UxXT_DirListItems	UxUT_string
UxXT_DirSearchProc	UxUT_voidFunction

Xtype ID	Utype Converted To
UxXT_FileListItems	UxUT_string
UxXT_FileSearchProc	UxUT_voidFunction
UxXT_FontStruct	UxUT_string
UxXT_HighlightPixmap	UxUT_string
UxXT_HistoryItems	UxUT_string
UxXT_InsertPosition	UxUT_cardFunction
UxXT_int	UxUT_int
UxXT_Items	UxUT_string
UxXT_KeySym	UxUT_string
UxXT_ListItems	UxUT_string
UxXT_Pixel	UxUT_string
UxXT_Pixmap	UxUT_string
UxXT_Position	UxUT_int
UxXT_QualifySearchDataProc	UxUT_voidFunction
UxXT_SelectedItems	UxUT_string
UxXT_short	UxUT_short
UxXT_String	UxUT_string
UxXT_StringOrNull	UxUT_string
UxXT_TopShadowPixmap	UxUT_string
UxXT_Translations	UxUT_string
UxXT_ValueWcs	UxUT_string
UxXT_VisualPointer	-
UxXT_Widget	UxUT_string
UxXT_WidgetClass	UxUT_string
UxXT_WidgetList	UxUT_stringTable
UxXT_Window	UxUT_string
UxXT_XID	UxUT_string
UxXT_XmFontList	UxUT_string
UxXT_XmString	UxUT_string
UxXT_XmTextSource	-

Validator And ValuesOf Functions

For each xtype defined by UIM/X, there is a ValuesOf function and a Validator function:

- A ValuesOf function provides a textual description of the allowable property values. UIM/X uses this text:
  - To compose the error messages displayed in the Message Window when an invalid property value is entered.
  - To construct an option menu for the property in the Property Editor.
- A Validator function validates a property value.

The names of the ValuesOf and Validator functions are derived from the name of the associated xtype. For example, the xtype `UxXT_Alignment` has the associated functions `UxValuesOfAlignment` and `UxValidateAlignment`.

The following table lists the Validator and ValuesOf functions defined by UIM/X. The declarations for these functions are contained in the files *uimx_directory*/custom/include/valuesOf.h and *uimx_directory*/custom/include/validate.h.

Validator Function	ValuesOf Function
UxValidateAccelerators	UxValuesOfAccelerators
UxValidateAlignment	UxValuesOfAlignment
UxValidateAny	UxValuesOfAny
UxValidateArgc	UxValuesOfArgc
UxValidateArgv	UxValuesOfArgv
UxValidateArrowDirection	UxValuesOfArrowDirection
UxValidateAtom	UxValuesOfAtom
UxValidateAudibleWarning	UxValuesOfAudibleWarning
UxValidateBitmap	-
UxValidateBool	UxValuesOfBool
UxValidateBoolean	UxValuesOfBoolean
UxValidateBottomAttachment	UxValuesOfAttachmentType
UxValidateBottomWidget	UxValuesOfConstraintWidget
-	UxValuesOfCallback
UxValidateCardFunction	UxValuesOfCardFunction
UxValidateChar	UxValuesOfChar
UxValidateChildPlacement	UxValuesOfChildPlacement
UxValidateChildType	UxValuesOfChildType
UxValidateChildVerticalAlignment	UxValuesOfChildVerticalAlignment
UxValidateChildrenList	UxValuesOfChildrenList
UxValidateCmpndResSet	UxValuesOfCmpndResSet
UxValidateCmpndSwidgetMethodSet	UxValuesOfCmpndSwidgetMethodSet
UxValidateColormap	UxValuesOfColormap
UxValidateCommandWindowLocation	UxValuesOfCommandWindowLocation
UxValidateConstraintWidget	UxValuesOfConstraintWidget
UxValidateDecimalPoints	UxValuesOfDecimalPoints
UxValidateDefaultButtonType	UxValuesOfDefaultButtonType
UxValidateDeleteResponse	UxValuesOfDeleteResponse
UxValidateDescendantWidget	UxValuesOfDescendantWidget
UxValidateDialogStyle	UxValuesOfDialogStyle
UxValidateDialogType	UxValuesOfDialogType
UxValidateDimension	UxValuesOfDimension
UxValidateDirListItemCount	UxValidateDirListItemCount
UxValidateDirListItems	UxValuesOfXmStringTable
UxValidateDragRecursion	UxValuesOfDragRecursion
UxValidateEditMode	UxValuesOfEditMode
UxValidateEntryClass	UxValuesOfEntryClass
UxValidateEntryVerticalAlignment	UxValuesOfEntryVerticalAlignment
UxValidateFileListItemCount	UxValuesOfItemCount
UxValidateFileListItems	UxValuesOfXmStringTable
UxValidateFileTypeMask	UxValuesOfFileTypeMask
UxValidateFontStruct	UxValuesOfFontStruct

Validator Function	ValuesOf Function
UxValidateGenericItemsOrItemCount	-
UxValidateGeometry	UxValidateGeometry
UxValidateHistoryItemCount	UxValuesOfItemCount
UxValidateHistoryItems	UxValuesOfXmStringTable
UxValidateImage	UxValuesOfImage
UxValidateIndicatorType	UxValuesOfIndicatorType
UxValidateInitialWindowState	UxValuesOfInitialWindowState
-	UxValuesOfInput
UxValidateInt	UxValuesOfInt
UxValidateIsHomogeneous	UxValuesOfIsHomogeneous
UxValidateItemCount	UxValuesOfItemCount
UxValidateItems	UxValuesOfXmStringTable
UxValidateKeyboardFocusPolicy	UxValuesOfKeyboardFocusPolicy
UxValidateKeysym	UxValuesOfKeysym
UxValidateLabelType	UxValuesOfLabelType
UxValidateLeftAttachment	UxValuesOfAttachmentType
UxValidateLeftWidget	UxValuesOfConstraintWidget
UxValidateListItemCount	UxValuesOfItemCount
UxValidateListItems	UxValuesOfXmStringTable
UxValidateListSizePolicy	UxValuesOfListSizePolicy
UxValidateMaximum	UxValuesOfScaleMinMaxValue
UxValidateMenuHistoryWidget	UxValuesOfMenuHistoryWidget
UxValidateMenuPost	UxValuesOfMenuPost
UxValidateMinimum	UxValuesOfScaleMinMaxValue
UxValidateMsgDialogType	UxValuesOfDialogType
UxValidateMultiClick	UxValuesOfMultiClick
UxValidateMwmInputMode	UxValuesOfMwmInputMode
UxValidateNavigationType	UxValuesOfNavigationType
UxValidateNonnegativeInt	UxValuesOfNonnegativeInt
UxValidateNonnegativeShort	UxValuesOfNonnegativeShort
UxValidateOrientation	UxValuesOfOrientation
UxValidatePacking	UxValuesOfPacking
UxValidatePaneMaximum	UxValuesOfPaneMaximum
UxValidatePaneMinimum	UxValuesOfPaneMinimum
UxValidatePixel	UxValuesOfPixel
UxValidatePixmap	UxValuesOfPixmap
UxValidatePointer	UxValuesOfPointer
UxValidatePosition	UxValuesOfPosition
UxValidatePositionIndex	UxValuesOfPositionIndex
UxValidatePositiveDimension	UxValuesOfPositiveDimension
UxValidatePositiveInt	UxValuesOfPositiveInt
UxValidatePositiveShort	UxValuesOfPositiveShort
UxValidateProcessingDirection	UxValuesOfProcessingDirection
UxValidateRadioBehavior	UxValuesOfRadioBehavior
UxValidateResizePolicy	UxValuesOfResizePolicy
UxValidateResizeRecursion	UxValuesOfResizeRecursion
UxValidateRightAttachment	UxValuesOfAttachmentType
UxValidateRightWidget	UxValuesOfConstraintWidget
UxValidateRowColumnType	UxValuesOfRowColumnType
-	UxValuesOfScaleMinMaxValue
UxValidateScaleMultiple	UxValuesOfScaleMultiple

Validator Function	ValuesOf Function
UxValidateScrollBarDisplayPolicy	UxValuesOfScrollBarDisplayPolicy
UxValidateScrollBarPlacement	UxValuesOfScrollBarPlacement
UxValidateScrollingPolicy	UxValuesOfScrollingPolicy
UxValidateSelectedItemCount	UxValuesOfItemCount
UxValidateSelectedItems	UxValuesOfSelectedItems
UxValidateSelectionArray	UxValuesOfSelectionArray
UxValidateSelectionArrayCount	UxValuesOfSelectionArrayCount
UxValidateSelectionPolicy	UxValuesOfSelectionPolicy
UxValidateSeparatorType	UxValuesOfSeparatorType
UxValidateShadowThickness	UxValuesOfShadowThickness
UxValidateShadowType	UxValuesOfShadowType
UxValidateShort	UxValuesOfShort
UxValidateShortDimension	UxValuesOfShortDimension
UxValidateString	UxValuesOfString
UxValidateStringDirection	UxValuesOfStringDirection
UxValidateStringTable	UxValuesOfStringTable
UxValidateTearOffModel	UxValuesOfTearOffModel
UxValidateTopAttachment	UxValuesOfAttachmentType
UxValidateTopWidget	UxValuesOfConstraintWidget
UxValidateTopItemPosition	UxValuesOfTopItemPosition
UxValidateTranslationString	-
UxValidateTranslations	UxValuesOfTranslations
UxValidateUnitType	UxValuesOfUnitType
UxValidateUserData	UxValuesOfUserData
UxValidateValue	UxValuesOfSBMinMaxValue
UxValidateValueWcs	UxValuesOfValueWcs
UxValidateVisualPointer	UxValuesOfVisualPointer
UxValidateVisualPolicy	UxValuesOfVisualPolicy
UxValidateVoidFunction	UxValuesOfVoidFunction
UxValidateWhichButton	-
UxValidateWidget	UxValuesOfWidget
UxValidateWidgetClass	UxValuesOfWidgetClass
UxValidateWinGravity	UxValuesOfWinGravity
UxValidateWindow	UxValuesOfWindow
UxValidateXID	UxValuesOfXID
UxValidateXmFontList	UxValuesOfXmFontList
UxValidateXmString	UxValuesOfXmString
UxValidateXmStringTable	UxValuesOfXmStringTable
UxValidateXmTextSource	UxValuesOfXmTextSource

D

Class Methods

E

Overview

This appendix contains the reference pages for the class methods used by UIM/X to operate on widgets. These methods are defined in the swidget classes. For example, when the user attempts to create a new widget as a child of an existing widget, a method is called on the proposed parent to verify that it can accept such a child. Some classes, such as drawingArea, accept most children. Other classes, such as scrolledWindow, can have only a fixed number of children. Each class has its own version of the method that implements the class-specific rules.

A swidget class inherits the methods of its superclasses. You can override these inherited class methods. See Chapter 2, “Integrating Widgets.”

init

Initializes an instance of a swidget class.

Method ID UxM_init

Synopsis

```
void init(swidget sw);
```

Arguments sw A swidget.

Return Value None.

Description The `init` method initializes the fields of a swidget's instance structure.

Example

```
/* macro to invoke method */
#define init(sw) UxVoid_get_op( sw, UxM_init )(sw);
```

See Also Ux*Type*_get_op in Appendix G, "Ux Builder Functions"

UxApply

Recreates a swidget hierarchy and exposes the selected swidgets.

Method ID

UxM_UxApply

Synopsis

```
void UxApply (swidget sw);
```

Arguments

sw A swidget.

Return Value

None.

Description

`UxApply` calls `UxBuild` and then redisplays (exposes) all selected swidgets.

The method is named `UxApply` because it is often the last thing done to a swidget when an editor (such as the Property Editor) is applied.

Example

```
swidget RadioBox;/* rowColumn swidget */
PList_c *buttons;/* list toggleButton children of
   RadioBox. */
/* Build list of edited buttons */

UxReorder( buttons, RadioBox, 0 );
UxApply( RadioBox );
```

See Also

UxBuild, *uimx_directory*`/contrib/BuilderEngine/radio.i` for the complete source for the above example.

UxBuild

Recreates a swidget hierarchy.

Method ID

UxM_UxBuild

Synopsis

```
void UxBuild( swidget sw, Boolean manage );
```

Arguments

sw	A swidget.
manage	A flag indicating whether or not to pop-up any top-level swidgets when they are recreated.

Return Value

None.

Description

`UxBuild` uses the method `UxRecreateParentOrChild` to determine which swidget to recreate. Once the proper swidget has been found, `UxBuild` recreates the swidget and its descendants:

- It unrealizes the swidget and its descendants.
- It evaluates the Expressions list and re-initializes the Values list of the swidget and each of its descendants.
- It validates the properties of the swidget and its descendants (invalid property values are not applied).
- It realizes the swidget and its descendants, and pops-up any top-level swidgets.

Note: Overridden subclass methods should always call this version of the method.

UxCanLoseChild

Invoked when a child of a swidget is about to be reparented or destroyed.

Method ID

UxM_UxCanLoseChild

Synopsis

```
int UxCanLoseChild(swidget parent, swidget child,
   swidget newParent);
```

Arguments

parent The parent swidget.

child A child swidget.

newParent The new parent swidget, if `child` is being reparented. `NULL` otherwise.

Return Value

`UxCanLoseChild` returns `NO_ERROR` to indicate that `child` can be reparented or destroyed, `ERROR` otherwise.

Description

Removing a child from a constraining parent may violate some cross-dependency. For example, removing a child from a form may break attachment dependencies.

This method can be used to check with the user before removing a child from a parent. As well, this method could be used to issue observer updates when a child is removed.

UxCheckChildren

Determines whether an adapter swidget is a valid parent for a set of children.

Synopsis

```
char *UxCheckChildren(swidget parent, Environment
   *env, int numChildren, class_t *childClasses,
   swidget *children);
```

Arguments

parent The parent adapter swidget.

env A pointer to a CORBA-compliant `Environment` structure declared in *uimx_directory*`/include/UxCorba.h`. You can pass `&UxEnv`to all methods.

numChildren The number of children.

childClasses The class of each child.

children The children

Return Value

`UxCheckChildren` returns an error message if the parent cannot accept one or more of the proposed children, and NULL otherwise.

Description

The adapter swidget always relays the UIM/X method `UxWidgetCannotAcceptChildren` to the component by calling the method `UxCheckChildren` on itself. The component can register the method `UxCheckChildren` with its own class code if it wants to validate the interactive reparenting and creation of children swidgets.

A convenient way to do this is to call `UxAdapterDesignMethods`. If no method is registered, the adapter will accept any child as long as there is a designated child site.

UxChildAdded

Invoked after a child is added to a swidget.

Method ID UxM_UxChildAdded

Synopsis

```
void UxChildAdded(swidget parent, swidget child);
```

Arguments

parent The parent swidget.

child A child swidget.

Return Value None.

Description `UxChildAdded` informs a parent swidget that a child is being added. This method can be used to clean up (for example, to issue observer updates) when a child is added.

UxChildRemoved

Invoked when a child swidget is removed from its parent.

Method ID

UxM_UxChildRemoved

Synopsis

```
void UxChildRemoved(swidget parent, swidget child,
    swidget newParent);
```

Arguments

parent	The parent swidget.
child	A child swidget.
newParent	The new parent swidget, if `child` is being reparented. `NULL` otherwise.

Return Value

None.

Description

`UxChildRemoved` informs a parent swidget that a child has been removed. This method can be used to clean up (for example, to issue observer updates) when a child is removed.

UxClassValidate

Validates the property values of an instance of a swidget class.

Method ID

UxM_UxClassValidate

Synopsis

```
int UxClassValidate( swidget sw, Resource_t **res );
```

Arguments

sw Swidget whose Values list is validated.

res Output parameter.

Return Value

`UxClassValidate` exits and returns `ERROR` when it finds the first invalid property value. A pointer to the invalid property's resource descriptor is passed back in `res`. The return value `NO_ERROR` indicates that all property values are valid.

Description

`UxClassValidate` sequences through `sw`'s Value list, applying to each property value the Validator function found in the property's resource descriptor. The Declaration properties are skipped.

If `UxClassValidate` finds an invalid property value, it outputs a message (using the property's ValuesOf function) to the Message Area of the start-up interface and returns.

UxClearExpressions

Removes all initial value expressions from a swidget's Expressions list.

Method ID UxM_UxClearExpressions

Synopsis

```
void UxClearExpressions(swidget sw);
```

Arguments sw A swidget.

Return Value None.

Description `UxClearExpressions` clears a swidget's Expressions list by destroying all initial value expressions and removing their entries from the Expressions list.

UxClearValues

Removes all entries from a swidget's Values list.

Method ID UxM_UxClearValues

Synopsis

```
void UxClearValues(swidget sw);
```

Arguments sw A swidget.

Return Value None.

Description `UxClearValues` clears the Values list of a swidget by destroying the associated data structures and removing their entries from the Values list.

UxDrawHandles

Draws selection handles around a widget.

Method ID UxM_UxDrawHandles

Synopsis

```
void UxDrawHandles(swidget sw);
```

Arguments sw A swidget.

Return Value None.

Description `UxDrawHandles` draws selection handles around the widget and any gadget children that have swidgets in the selected list.

UxDrawHandles

Draws selection handles around an adapter widget.

Synopsis

```
void UxDrawHandles(swidget sw, Environment *env);
```

Arguments

sw	An adapter swidget.
env	A pointer to a CORBA-compliant `Environment` structure declared in *uimx_directory*`/include/UxCorba.h`. You can pass `&UxEnv` to all methods.

Return Value

None.

Description

The adapter swidget relays the UIM/X method `UxDrawHandles` to the component by calling the method of the same name on itself. The component can register the method `UxDrawHandles` with its own class code if it has special needs for its selection handles.

UxHandlePostCreation

Invoked after a swidget is created.

Method ID UxM_UxHandlePostCreation

Synopsis

```
void UxHandlePostCreation( swidget sw );
```

Arguments sw A swidget.

Return Value None.

Description UIM/X uses this method to ensure that the children of a MainWindow or ScrolledWindow swidget are managed correctly. This method gets the children of the swidget `sw` and ultimately calls either `XmMainWindowSetAreas` or `XmScrolledWindowSetArea`.

UxInteractiveChildCreate

Supplies an error message if the interactive creation of a child swidget is not permitted.

Method ID UxM_UxInteractiveChildCreate

Synopsis

```
char *InteractiveChildCreate( swidget sw );
```

Arguments sw The proposed parent swidget.

Return Value Returns `NULL` if the user is allowed to interactively create a child of `sw`. Otherwise, the method returns an error message.

Description This method specifies whether or not a child can be interactively created for a given swidget.

UxInteractiveCreateAndApply

Creates and applies a new swidget after the user has dragged and released the Select mouse button.

Method ID

UxM_UxInteractiveCreateAndApply

Synopsis

```
swidget InteractiveCreateAndApply( Class_t cl,
   swidget parent, Position cx, Position cy,
   Dimension dx, Dimension dy, int toplevel, Position
   mx, Position my );
```

Arguments

cl	The class of swidget to create.
parent	The parent of the swidget being created. This parameter is `NULL` if a top-level swidget is being created.
cx, cy	The coordinates of the mouse pointer when the Select button was released.
dx, dy	The differences between the mouse pointer coordinates `mx` and `my` and the coordinates of the upper-left inner corner of the parent swidget.
toplevel	`TRUE` if a top-level swidget is to be created, `FALSE` otherwise.
mx, my	The coordinates of the mouse pointer when the Select button was pressed.

Return Value

Returns the new swidget.

Description

Creates an instance of the swidget class `cl` and sizes it according to the passed coordinates. If `parent` is `NULL`, it is set to `UxParent`.

UxMakeArglist

Creates an XtArglist from the swidget's Values list.

Method ID

UxM_UxMakeArglist

Synopsis

```
int UxSwidgetMakeArglist(swidget sw, Arg a[], int
   pass);
```

Arguments

sw	A swidget.
a	A list of Arg structures.
pass	`UxPASS0`, `UxPASS1`, or `UxPASS2`

Return Value

`UxMakeArglist` returns the number of properties added to arglist.

Description

`UxMakeArglist` creates an XtArglist from a swidget's Values list by converting the value in each entry.

UxMenusMenuSensitivities

Sets the sensitivities of the items on the Menus submenu.

Method ID

UxM_UxMenusMenuSensitivities

Synopsis

```
void UxMenusMenuSensitivities(swidget sw, int
   *popup, int *pulldown, int *option);
```

Arguments

sw	A swidget.
popup	Returns `True` or `False` to indicate whether or not `sw` can have a popup menu as a child.
pulldown	Returns `True` or `False` to indicate whether or not `sw` can have a pulldown menu as a child.
option	Returns `True` or `False` to indicate whether or not `sw` can have an option menu as a child.

Return Value

None.

Description

`UxMenusMenuSensitivities` controls the sensitivities of the Popup, Pulldown, and Option items on the Menus submenu of the Selected Objects popup menu.

UxObjectToRecreate

Determines whether to recreate an adapter swidget or one of its ancestors.

Synopsis

```
UxObjectToRecreate(swidget child, Environment *env,
    swidget parent);
```

Arguments

child	A child adapter swidget.
env	A pointer to a CORBA-compliant `Environment` structure declared in *uimx_directory*`/include/UxCorba.h`. You can pass `&UxEnv`to all methods.
parent	A parent swidget.

Return Value

Returns the swidget to be recreated.

Description

The adapter swidget relays the UIM/X methods `UxRecreateParentOrChild` and `UxRecreateSwidget` to the component by calling the method `UxObjectToRecreate` on itself. The component can register the method `UxObjectToRecreate` with its own class code to determine which swidget to recreate when the adapter swidget when the adapter swidget must be recreated. The return value could be the adapter or one of its ancestors.

A convenient way is to call `UxAdapterDesignMethods`. If no method is registered, the adapter will return the component.

UxRealize

Realizes the swidget tree into an X widget tree.

Method ID UxM_UxRealize

Synopsis

```
widget UxRealize(swidget sw)
```

Arguments sw A swidget

Return Value Returns the widget for swidget `sw`.

Description If the swidget has no X widget, `UxRealize` realizes the swidget tree into an X widget tree, using all the values in the swidget's Resources lists.

UxRecreateParentOrChild

Determines whether to recreate a swidget or one of its ancestors.

Method ID UxM_UxRecreateParentOrChild

Synopsis

```
swidget UxRecreateParentOrChild( swidget sw );
```

Arguments sw A swidget.

Return Value Returns the swidget to be recreated.

Description `UxRecreateParentOrChild` determines whether a swidget or one of its ancestors must be recreated. This method tells you which swidget you must recreate to ensure that the swidget `sw` is properly recreated.

Note: UIM/X recreates a swidget by first destroying (unrealizing) it and then creating (realizing) it.

UxRecreateSwidget

Determines whether a child or a parent swidget should be recreated.

Method ID UxM_UxRecreateSwidget

Synopsis

```
swidget UxRecreateSwidget( swidget child, swidget
   parent );
```

Arguments

child A child swidget.

parent The parent swidget.

Return Value Returns `child` or `parent`, whichever should be recreated in response to a geometry change to `child`.

Description `UxRecreateSwidget` determines whether or not you must recreate the parent to properly recreate a child.

UxSetNonarglist

Sets a property value that cannot be set by `XtSetValues`.

Method ID UxM_UxSetNonarglist

Synopsis

```
void UxSetNonarglist( swidget sw );
```

Arguments sw A swidget.

Return Value None.

Description `UxSetNonarglist` is called to set property values such as accelerators and translations.

UxUnrealize

Destroys the underlying widget.

Method ID UxM_UxUnrealize

Synopsis

```
void UxUnrealize(swidget sw);
```

Arguments sw A swidget.

Return Value None.

Description `UxUnrealize` destroys the underlying widget.

UxValidMoveOrResize

Determines whether or not a valid move or resize operation is being performed.

Method ID

UxM_UxValidMoveOrResize

Synopsis

```
char *UxValidMoveOrResize(swidget sw, int *action,
   swidget *actual_sw);
```

Arguments

sw	The swidget being moved or resized.
action	Either DO_MOVEor DO_RESIZE. The method may reset this value.
actual_sw	Returns the actual swidget to move or resize.

Return Value

UxValidMoveOrResize returns NULL if action can be performed on actual_sw. Otherwise, an error message is returned.

Description

UxValidMoveOrResize determines whether or not sw or some other swidget (a parent or child) can be moved or resized. actual_sw always contains the swidget to move or resize.

If the requested action is illegal, the method may permit the alternate action to be performed. For example, if *action is DO_RESIZE and sw is a menu bar, UxValidMoveOrResize sets *action to DO_MOVE and returns.

UxWidgetCannotAcceptChildren

Determines whether or not a swidget is a valid parent for a set of children.

Method ID

UxM_UxWidgetCannotAcceptChildren

Synopsis

```
char *UxWidgetCannotAcceptChildren(swidget parent,
   int numChildren, Class_t *childClasses, swidget
   *children);
```

Arguments

parent The parent swidget.

numChildren The number of children.

childClasses The class of each child.

children The children.

Return Value

`UxWidgetCannotAcceptChildren` returns an error message if the parent cannot accept one or more of the proposed children, and `NULL` otherwise.

Description

`UxWidgetCannotAcceptChildren` is used to validate the interactive reparenting and creation of swidgets.

Resource Descriptors

F

Overview

Every widget property in UIM/X is described by a data object called a resource descriptor. This data object is defined in the header file *uimx_directory*/custom/include/resource.h. It is a Resource_t structure, with access macros defined for its fields.

Every swidget class holds two PLists of resource descriptors: the resource set and the constraint set. The resource set is the list of resource descriptors for the properties declared by the class. The constraint set is the list of resource descriptors for the constraint properties of the class. You can use UxGetResourceSet and UxGetConstraints to get these lists.

When you define new properties for widget classes, you use UxDefineResource to create and initialize new resource descriptors. The arguments to UxDefineResource are simply a list of values for the fields of the resource descriptor, keyed with keys from the enumeration RD_Key_t, which is also defined in *uimx_directory*/custom/include/resource.h.

For components, you use UxInstanceResource() and UxGlobalInstanceResource().

Resource Descriptor Fields

Field	Description	Default
Format	Format string for making C code from the property value. For example, `"\"%s\""`.	NULL1
UimxName	The name of the property for interface files and `UxGet`and `UxPut` macros. For example, `"allowShellResize"`.	None
UxXtName	The name used in `XtSetValues`, `XtGetValues`, and `XtAddCallback`. For example, `"allowShellResize"`.	UimxName
PrintName	The name used in the Property Editor and in messages. For example, `"AllowShellResize"`.	`UimxName` with the first letter capitalized.
UType	The utype of the property. One of the `UxUT_` constants from *uimx_directory*`/custom/include/utype.h`.	UxUT_int
XType	The xtype of the property (the type used in `XtGetValues`and `XtSetValues`). One of the `UxXT_` variables from *uimx_directory*`/custom/include/utype.h`.	UxXT_int
Division	The category in the Property Editor. Possible values are `UxCORE`, `UxSPECIFIC`, `UxCONSTRAINT`, `UxBEHAVIOR`, `UxDECL`, and `UxCOMPOUND`.	UxCORE
Toolkit	`True` or `False` indicating whether or not the property value can be set with `XtSetValues`.	True
Treatment	Indicates how the Interpreter treats the value. Possible values are `UxSTATEMENT` (for callbacks), `UxEXPRESSION` (for properties), and `UxLITERAL` (for names).	`UxEXPRESSION`2
Pass	Indicates when during widget creation the property value is set: `UxPASS0` - initial creation `UxPASS1` -just after creation `UxPASS2` -after all widgets created	UxPASS0

Field	Description	Default
ReadOnly	Controls the sensitivity of the property in the Property Editor. If set to `True`, the property is insensitive, and the lock pixmap is displayed beside the property name.	False
CanBePublic	`True` or `False` indicating whether or not the property can be set in a resource file.	True
Editable	`True` or `False` indicating whether or not the property appears in the Property Editor.	True
NoEditBtn	`True` or `False` indicating whether or not the Editor pushButton is available for the property in the Property Editor.	False
ValuesOf	Pointer to a function that returns a description of possible values of a property.	NULL
Validator	Pointer to a function that validates an input property value.	NULL
GetFunction	Pointer to the function used to get the xtype value from the widget and convert the result into a utype value. Not used by callback properties.	UIM/X installs the appropriate function based on the value of the `UType` field.
PutFunction	Pointer to the function that converts a utype value to an xtype value and passes the result to the widget. Not used by callback properties.	UIM/X installs the appropriate function based on the value of the `UType` field.
CopyFunction	Pointer to a function that makes a copy of a utype value for storage in the swidget.	`UxCopyString` if `UType` is `UxUT_string`. `NULL` otherwise.
FreeFunction	Pointer to a function that frees a copy of a value made with `CopyFunction`.	`UxFree` if `UType` is `UxUT_string`. `NULL` otherwise.
EditFunction	Pointer to a popup editor function for the property.	NULL
SaveFunction	Pointer to a function used to write a resource specification to a file.	NULL3

Field	Description	Default
UserDataRD	User data field for extending and customizing UIM/X. Users can add their own property data in this field.	NULL

[1.] For callbacks, "{(void) %s (UxWidget, UxClientData, UxCallbackArg);}".

For userData properties, "(XtPointer) 0x%1x".

For properties whose utype is UxUT_string, "\"%s\"".

Otherwise the default is NULL.

[2.] UIM/X sets Treatment to UxSTATEMENT if Division is set to UxBEHAVIOR.

[3.] If this field is NULL, UIM/X uses its own default functions for writing properties. Most of these fields apply only to resource properties. A callback can be defined simply by supplying a UimxName and specifying that the Division is UxBEHAVIOR.

Ux Builder Functions

G

Overview

This appendix contains detailed descriptions of each of the Ux Builder Functions. In a few cases, a group of functions are described on a single reference page (for example, the `Ux`*Type*`_get_op` functions for looking up class methods) because they perform similar or related operations.

Note: The functions described in this appendix are used to extend and customize UIM/X. They are not for use in generated applications.

UxAdapterDesignMethods()

Registers design-time component methods.

Synopsis

```
#include <UxLib.h>
void UxAdapterDesignMethods( int clsCode, void
   *checkChildren, void *drawHandles, void
   *objectToRecreate);
```

Arguments

clsCode The class code for the component.

checkChildren Function pointer for the `UxCheckChildren()` method.

drawHandles Function pointer for the `UxDrawHandles()` method.

objectToRecreate Function pointer for the `UxObjectToRecreate()` method.

Return Value

None.

Description

`UxAdapterDesignMethods()` is a convenience function for registering design-time methods for a given component. You register a method by passing a function pointer. You pass `NULL` if you don't want to register a method.

Example

```
#ifdef DESIGN_TIME

int UxComponent_UxCheckChildren_Id = -1;
char* UxComponent_UxCheckChildren_Name =
   "UxCheckChildren";
static char* _Component_UxCheckChildren( swidget sw,
   Environment *pEnv)
{
if (pEnv)
   pEnv->major(CORBA::NO_EXCEPTION);

   return "This component cannot have children.";
}
#endif
int create_Component_ClassId( void )
{
```

```
    static int IfClassCode = -1;
    if ( IfClassCode == -1)
    {
       IfClassCode = UxNewInterfaceClassId();
       …
       #ifdef DESIGN_TIME
          UxAdapterDesignMethods(IfClassCode,_Compone
          nt_UxCheckChildren, NULL, NULL);
       #endif
    }
    return(IfClassCode);
}
```

UxAdapterSwidget()

Creates an adapter swidget for a component.

Synopsis

```
#ifdef XT_CODE
   #include <UxXt.h>
#else
   #include <UxLib.h>
#endif

swidget UxAdapterSwidget( Widget wid, swidget
   parent, char *name, int clsCode, void* cmptRef,
   void* context );
```

Arguments

wid	A widget. This widget must be the controlling widget from the component.
parent	The parent swidget of the widget. This value is passed to the component's constructor.
name	The name given to the adapter swidget. You can simply pass `XtName(wid)`.
clsCode	The class code for the component. This value is obtained from `UxNewInterfaceClassId()` or `UxNewSubclassId()`.
cmptRef	The component reference (a pointer to the component). This value is stored in the instance structure of the adapter swidget.
context	A pointer to the context structure for the adapter swidget. This should be `NULL` (or `UxNO_CONTEXT`) for the adapter swidget created in the body of a GUI component's constructor. If you create a separate adapter swidget for a child site widget, pass the component's context (obtained by calling `UxGetContext()`).

Return Value

Returns an adapter swidget.

Description

`UxAdapterSwidget()` creates an adapter swidget used to represent a widget in UIM/X. You use adapter swidgets to integrate components into UIM/X.

You use `UxAdapterSwidget()` to get a swidget that can be returned by a component constructor (its Interface Function) or by a `childSite()` method.

To integrate a component into UIM/X, you write a constructor that returns an adapter swidget. This adapter swidget is connected to the underlying component through the UIM/X Method system.

You don't have to destroy the adapter swidget. `UxAdapterSwidget()` adds a destroy callback to the widget `wid` that destroys the swidget for you. You are responsible, however, for destroying the component. You can do this by adding a destroy callback to `wid` that destroys the component.

See Also UxAdapterDesignMethods(), UxGetComponentRef(), UxPutComponentRef()

UxAddConv()

Installs a type converter.

Synopsis

```
#include <utype.h>

void UxAddConv( int utype, int xtype, int
   (*conv_fcn)() );
```

Arguments

utype	ID of the utype.
xtype	ID of the xtype.
conv_fcn	The converter function to be used.

Return Value

None.

Description

`UxAddConv()` installs `conv_fcn` as the converter to be used when converting property values between the given `utype` and `xtype`. UIM/X stores the

converter functions in an internal table—the xtype and utype IDs are the indices to the table.

The converter function should follow the following format:

```
int convert_A_B( swidget sw, utype *udata, xtype *xdata,int
flag, int xtype );
```

- `sw` is the swidget requiring the conversion.
- `*udata` is the UIM/X value, and `*xdata` is the Xt value.
- `flag` indicates the direction of conversion:
  - `TO_UIMX` to convert `*xdata` to `*udata`.
  - `TO_X` to convert `*udata` to `*xdata`.
- `xtype` is the ID of the property xtype.

If a converter has already been installed for this pair of types, an error message is given.You should install a converter for a new xtype if the property values expected by the widgets do not match those expected by the swidgets. This is done by calling `UxAddConv()` from the function `UxAddUserDefXtypes()` in *uimx_directory*`/custom/src/user-xtype.c`.

Example

Widgets expect the values of some constraint properties to be widget pointers but swidgets expect them to be swidget names. Thus, a converter is needed to switch between widget pointers and swidget names. The following code outlines such a converter:

```
int convert_name_Widget( swidget swgt, char
   **ptr_name, Widget *ptr_wgt, int flag, int xtype )
{
   int status = NO_ERROR;
   if ( flag == TO_UIMX )
   {
      if ( *ptr_wgt != NULL )
         *ptr_name = XtName( *ptr_wgt );
      else
         *ptr_name = "";
   }
   else
   {
      swidget named_swgt = UxNameToSwidget( swgt,
         *ptr_name );
      if ( named_swgt == NULL )
         *ptr_wgt = NULL;
      else
         *ptr_wgt = UxGetWidget( named_swgt );
      if ( *ptr_wgt == NULL )
         status = ERROR;
      }
      return ( status );
}
```

This converter would be installed by the following function call:

```
UxAddConv( UxUT_string, UxXT_Widget,
   convert_name_Widget );
```

See Also Appendix D, “Resource Types,” for listings of the utype and xtype IDs defined by UIM/X, UxAddEnumType(), UxAddXtype(), UxCallConverter()

UxAddEnumType()

Adds the definition of an enumerated xtype.

Synopsis

```
#include <utype.h>

int UxAddEnumType( char* name, int xt_size,
    unsigned char *xt_vals, char** uimx_vals, char**
    xdef_vals, int num_vals, int (*converter)() );
```

Arguments

name	The name of the xtype.
xt_size	The size of the xtype.
xt_vals	Array of Xt values.
uimx_vals	Array of UIM/X values.
xdef_vals	Array of Xt-defined constants.
num_vals	The number of array elements.
converter	The converter function. UIM/X supplies two pre-defined conversion functions: `UxStringToCharEnum` and `UxStringToIntEnum`.

Return Value

`UxAddEnumType()` returns the ID of the new xtype. By convention, the xtype IDs are stored in global variables named `UxXT_`xtype.

Description

`UxAddEnumType()` installs the arrays of possible values and the converter for an enumerated xtype. Xtypes describe the data type and possible values of a widget property (as opposed to a swidget property, which is described by a utype). An enumerated xtype is a data type with a fixed list of possible values.

`UxAddEnumType()` does its work by calling `UxAddXtype()` and `UxAddConv()`.

Note that most enumerated properties are of type `unsigned char` and use the converter `UxStringToCharEnum()`. The exceptions are of type `int` and use the converter `UxStringToIntEnum()`.

UxAddMweEditorSeparator()

Adds a separator to one of the option menus in the Main Window Editor.

Synopsis

```
void UxAddMweEditorSeparator( void *ptr );
```

Arguments

ptr — An opaque pointer to an internal data structure used to manage the Main Window Editor option menu. UIM/X passes the appropriate pointer to `UxCreateMweWorkArea` and `UxCreateMweMsgWindow`.

Return Value

None.

Description

The Work Area and Message Window option menus in the Main Window Editor are defined by the functions `UxCreateMweWorkArea` and `UxCreateMweMsgWindow` in *uimx_directory*`/custom/src/cr-mwe.c`.

To add a separator to the Work Area option menu, add a call to `UxAddMweEditorSeparator()` in `UxCreateMweWorkArea`. To add a separator to the Message Window option menu, add a call to `UxAddMweEditorSeparator()` in `UxCreateMweMsgWindow`.

Example

```
void UxCreateMweMsgWindow( ptr )
void *ptr;
{
   extern Class_t UxC_separator,
      UxC_label,
      UxC_text,
      UxC_textField;
   UxAddToMweEditor( ptr, CGETS_MWE(NONE),
   (Class_t)0);
   UxAddMweEditorSeparator( ptr );
   UxAddToMweEditor( ptr, CGETS_MWE(LABEL),
   UxC_label);
   UxAddToMweEditor( ptr, CGETS_MWE(TEXT),
   UxC_text);
   UxAddToMweEditor( ptr, CGETS_MWE(TEXTFIELD),
   UxC_textField);
}
```

See Also UxAddToMweEditor(), “Customizing the Main Window Editor’s Option Menus” in Chapter 2, “Integrating Widgets”

UxAddToCreateMenu()

Adds an item to a Create menu.

Synopsis

```
swidget UxAddToCreateMenu(swidget rowcol_sw, char*
   label,char* mnemonic, int toplevel_flag, void
   (*before_func)(), Class_t cl, void
   (*after_func)() );
```

Arguments

rowcol_sw Identifies the rowColumn swidget (the menu pane) of a Create menu.

label Specifies the label for the new menu item.

mnemonic Specifies the mnemonic for the menu item.

toplevel_flag `TRUE` if the menu item creates a top-level widget; `FALSE` if the menu item creates a parented widget.

before_func Specifies the function called before the user interactively creates an instance of the swidget class. `before_func` is called by the callback function registered with the XmNactivateCallback property of the pushButton.

The callback calls `before_func` with a single argument which is the swidget under the mouse pointer when the user interactively creates a new swidget.

cl Specifies the class of the swidget instances created by the menu item. If `cl` is `NULL`, there is no interactive creation and `after_func` is not called. When `cl` is `NULL`, `before_func` can be used to pop-up a specialized editor for creating swidgets—for example, UIM/X's Menu Editor.

after_func Specifies the function called after the user interactively creates an instance of the swidget class. `after_func` is called by the callback function registered with the XmNactivateCallback property of the pushbutton.

The callback calls `after_func` with a single argument which is the new swidget.

Return Value

`UxAddToCreateMenu()` returns the `swidget` variable of the new pushButton swidget.

Description

`UxAddToCreateMenu()` adds a pushButton menu item to the menu pane `rowcol_sw`. The pushButton is given the specified label and mnemonic. When the user clicks on the pushButton:

- The callback registered with XmNactivateCallback calls `before_func` (if it is not `NULL`). The callback passes a `swidget` variable to `before_func`:
  - When the user creates a swidget from the Selected Widgets popup, the swidget for which the menu was popped-up is passed to `before_func`.
  - When the user creates a swidget from a Browser Create menu, the first swidget in the list of selected swidgets is passed to `before_func`. If there are no selected swidgets, the value `NULL` is passed to `before_func`.
  - When the user creates a swidget from a Project Window Create menu, the value `NULL` is always passed to `before_func`. This is because the Project Window Create menus create top-level swidgets that have no parents.
- If `cl` is not `NULL`, the user then interactively creates an instance of the swidget class. `after_func` (if it is not `NULL`) is called immediately after the swidget is created. `after_func`can be used to pop-up a specialized editor—UIM/X's Main Window Editor is popped-up by an `after_func`.

UIM/X's Create menus are defined by functions in *uimx_directory*`/custom/src/cr-menus.c`. For each Create menu, there is a function containing a series of calls to `UxAddToCreateMenu()`. You can add new items to these menus by adding calls to `UxAddToCreateMenu()` in the appropriate function.

Example

In *uimx_directory*`/custom/src/cr-menus.c`, the function `UxSpecifyTopCustomMenu()` defines the Project Window's Custom Create menu. The call to `UxAddToCreateMenu()` shown below adds a menu item for the Square widget class to that Create menu:

```
void UxSpecifyTopCustomMenu( casc_swgt, rowcol_swgt
   )
   swidget casc_swgt;
   swidget rowcol_swgt;
{
   extern Class_t UxC_square;

   (void) UxAddToCreateMenu( rowcol_swgt,"Square",
```

```
        "q",
        TRUE,
        (void (*)()) NULL,
        UxC_square,
        (void (*)()) NULL );
}
```

See Also “Customizing UIM/X’s Create Menus” in Chapter 2, “Integrating Widgets”.

UxAddToMweEditor()

Adds a menu item to one of the Main Window Editor's option menus.

Synopsis

```
void UxAddToMweEditor(void *ptr, char* label,
   Class_t cl);
```

Arguments

ptr	An opaque pointer to an internal data structure used to manage the Main Window Editor option menu. UIM/X passes the appropriate pointer to `UxCreateMweWorkArea` and `UxCreateMweMsgWindow`.
label	Specifies the label of the menu item.
cl	Identifies the swidget class associated with the menu item.

Return Value

None.

Description

The Work Area and Message Window option menus in the Main Window Editor are defined by the functions `UxCreateMweWorkArea` and `UxCreateMweMsgWindow` in *uimx_directory*`/custom/src/cr-mwe.c`.

These functions contain a series of calls to `UxAddToMweEditor()`. Each call to `UxAddToMweEditor()` adds a pushButton menu item to one of the option menus in the Main Window Editor.

To add an item to the Work Area option menu, add a call to `UxAddToMweEditor()` in `UxCreateMweWorkArea`. To add an item to the Message Window option menu, add a call to `UxAddToMweEditor()` in `UxCreateMweMsgWindow`.

Note: Passing `UxC_separator` (or `UxC_separatorGadget`) as the swidget class ID (the `cl` parameter) adds a separator to the menu.

Items on the Work Area option menu must be subclasses of the manager swidget class (`UxC_manager`). Items on the Message Window option menu must be subclasses of the primitive swidget class (`UxC_primitive`).

Example

```
void UxCreateMweWorkArea( ptr )
   void *ptr;
{
   externClass_t UxC_separator,
      UxC_bulletinBoard,
```

```
        UxC_drawingArea,
        UxC_fileSelectionBox,
        UxC_form,
        UxC_frame,
        UxC_mainWindow,
        UxC_messageBox,
        UxC_panedWindow,
        UxC_rowColumn,
        UxC_scale,
        UxC_scrolledWindow,
        UxC_selectionBox,
        UxC_square;
    UxAddToMweEditor( ptr, CGETS_MWE(NONE),
    (Class_t)0);
    UxAddMweEditorSeparator( ptr );
    UxAddToMweEditor( ptr, CGETS_MWE(BULLTNBRD),
        UxC_bulletinBoard);
    UxAddToMweEditor( ptr, CGETS_MWE(DRWGAREA),
        UxC_drawingArea);
    UxAddToMweEditor( ptr, CGETS_MWE(FILESELBOX),
        UxC_fileSelectionBox);
    UxAddToMweEditor( ptr, CGETS_MWE(FORM),
        UxC_form);
    UxAddToMweEditor( ptr, CGETS_MWE(FRAME),
        UxC_frame);
    UxAddToMweEditor( ptr, CGETS_MWE(MAINWND),
        UxC_mainWindow);
    UxAddToMweEditor( ptr, CGETS_MWE(MSGBOX),
        UxC_messageBox);
    UxAddToMweEditor( ptr, CGETS_MWE(PANEDWND),
        UxC_panedWindow);
    UxAddToMweEditor( ptr, CGETS_MWE(ROWCOLUMN),
```

```
        UxC_rowColumn);
    UxAddToMweEditor( ptr, CGETS_MWE(SCALE),
        UxC_scale);
    UxAddToMweEditor( ptr, CGETS_MWE(SCRLWND),
        UxC_scrolledWindow);
    UxAddToMweEditor( ptr, CGETS_MWE(SELBOX),
        UxC_selectionBox);
    UxAddToMweEditor( ptr, "Square", UxC_square);
}
```

See Also

UxAddMweEditorSeparator(), "Customizing the Main Window Editor's Option Menus" in Chapter 2, "Integrating Widgets"

UxAddXtype()

Adds a new xtype definition.

Synopsis

```
#include <utype.h>
int UxAddXtype( char *name, int size );
```

Arguments

name	The name of the xtype.
size	The size of the new type.

Return Value

`UxAddXtype()` returns the ID the new xtype. By convention, the xtype IDs are stored in global variables named `UxXT_`xtype.

Description

`UxAddXtype()` adds an xtype for a new data type of an Xt property. If you add a new swidget class that has a property for which the values expected by the widget are of a different data type or have a different set of permissible values than any of the existing properties, you must add a new xtype. You do this by adding a call to `UxAddXtype()` in the function `UxAddUserDefXtypes` in *uimx_directory*`/custom/src/user-xtype.c`.

`UxAddConv()` registers a function to be used to convert between the new xtype and a given utype.

Example

```
int UxXT_Widget = UxAddXtype( "Widget",
   sizeof(Widget) );
```

See Also

“Defining New Xtypes” in Chapter 2, “Integrating Widgets” , UxAddConv(), UxAddEnumType()

UxCallConverter()

Calls the function that converts between a given utype and a given xtype.

Synopsis

```
#include <utype.h>

int UxCallConverter( swidget sw, int utype, char
   **udata, int xtype, char* xdata, int flag );
```

Arguments

sw	Swidget whose property value is being converted.
utype	A utype ID (a `UxUT_`variable).
udata	The UIM/X data.
xtype	An xtype ID (a `UxXT_`variable).
xdata	The X data.
flag	Flag indicating the direction of conversion. `TO_X` converts `udata`to `xdata`. `TO_UIMX` converts `xdata`to `udata`.

Return Value

If a converter function exists for `xtype` and `utype`, `UxCallConverter()` returns the value returned by the converter function. Otherwise `UxCallConverter()` outputs an error message to `stderr` and returns `ERROR`.

Description

`UxCallConverter()` looks-up and calls the converter function registered (by `UxAddConv()`) for a specific `utype`, `xtype` pair.

Examples

```
#include <utype.h>
swidget sw;
Resource_t *res;
…
UxCallConverter ( sw, (int)(res->XType), uvalue,
   (int)(res->UType), xvalue, TO_UIMX );
```

See Also

UxAddConv(), UxAddEnumType()

UxCreateMethodSignature()

Creates a method signature from a description of the method arguments.

Synopsis

```
#include <uxmethod.h>

struct Method_t *UxCreateMethodSignature(char
   *mname, VTCorbaSupport corba, char *rettype,
   ...);
```

Arguments

mname	The method name.
corba	The kind of CORBA support. This determines the presence and position of the CORBA environment in the method arguments.
rettype	The return type of the method.
...	A possibly empty, NULL-terminated list of resource descriptors describing the method arguments.

Return Value

Returns the method signature.

Description

A method signature is the description of the arguments of a method and its return type. The builder needs this information at design time to determine how to call compiled-in methods and how to display them in the Connection Editor.

`UxCreateMethodSignature` () creates the signature that you register with `UxMethodSignatureRegister()`. For backward compatibility, if you do not register a signature, the builder will assume it is a CORBA 1 method, and it will not be available in the Connection Editor.

You use enumerated values to specify the presence and position of the CORBA environment in the method arguments. Use `Corba1` if the environment is the second argument. Use `Corba2` if the environment is the last argument. Use `CorbaNONE` if there is no environment.

The value of `rettype` should be `NULL` if the method returns void, otherwise `rettype` is the type as it appears in the source code, surrounded by double quotes. For example, use `"int"` for a method returning int.

The following arguments are a list of `Resource_t*` describing the method arguments in the same order as they appear in the method. The target swidget that appears as the first argument in C is omitted. The function `UxEnvArgResource()` returns the `Resource_t*` needed to specify the CORBA environment. The `Resource_t*` for other arguments is obtained by calling `UxGetArgResource()`. See that reference page for details.

The list must be NULL-terminated.

Example

```
UxCreateMethodSignature("_set_height", Corba1, NULL,
   UxEnvArgResource(),
UxGetArgResource("height", UxUT_int,
   "0",UxValidateInt, UxValuesOfInt),
NULL);
```

See Also

UxEnvArgResource(), UxGetArgResource(), UxMethodSignatureLookup(), UxMethodSignatureRegister()

UxDDGetProp()

Gets the value of a swidget property.

Synopsis

```
#include <UxLib.h>

XtArgVal UxDDGetProp(swidget sw, char* name);
```

Arguments

sw	A swidget.
name	The name of the property (an XmNor XtN constant).

Return Value

UxDDGetProp() returns the specified property value if successful, and 0 otherwise.

Description

UxDDGetProp() is part of the Ux Convenience Library. This function is used to define the run-time versions of the UxGetProperty macros (in the swidget class public header files).

UxDDGetProp() is used when run-time conversion of property values is required. Run-time conversion is required when the data type of the values expected by the swidget and a widget are not the same. Note that any such property should be installed in the Ux Convenience Library using UxDDInstall().

UxGetProp() is used to define run-time UxGetProperty macros for properties that do not require run-time conversion.

Example

```
#define UxGetSelectColor(o) \
    (char *)UxDDGetProp(o, XmNselectColor)
```

See Also

UxDDInstall(), UxDDPutProp(), UxGetProp(), UxPutProp()

UxDDInstall()

Registers a property for run-time conversion.

Synopsis

```
#include <UxLib.h>
#include <utype.h>
void UxDDInstall( char *name, int utype, int xtype );
```

Arguments

name	The name of the property.
utype	The ID of the utype (one of the `UxUT` variables declared in *uimx_directory*`/custom/include/utype.h`).
xtype	The ID of the xtype (one of the `UxXT` variables declared in *uimx_directory*`/custom/include/utype.h`).

Return Value

None.

Description

`UxDDInstall()` registers a property needing conversion with the Ux Convenience Library. A property needs conversion when the values expected by the swidget do not match those expected by the widget.

Example

```
UxDDInstall( XmNalignment, UxUT_string,
   UxXT_Alignment );
```

See Also

UxDDGetProp()

UxDDPutProp()

Sets the current value of a swidget property.

Synopsis

```
#include <UxLib.h>
int UxDDPutProp(swidget sw, char *name, XtArgVal
   value);
```

Arguments

sw A swidget.

name The name of the property (an XmNor XtN constant).

value The property value.

Return Value

UxDDPutProp() returns NO_ERROR if successful, ERROR otherwise.

Description

UxDDPutProp() is part of the Ux Convenience Library. This function is used to define the run-time versions of the UxPutProperty macros (in the swidget class public header files).

UxDDPutProp() is used when run-time conversion of property values is required. Run-time conversion is required when the data type of the values expected by the swidget and a widget are not the same. Note that any such property should be installed in the Ux Convenience Library using

UxDDInstall().
UxPutProp() is used to define run-time UxPutProperty macros for properties that do not require run-time conversion.

Example

```
#define UxPutWinGravity(o, v) \
UxDDPutProp(o, XmNwinGravity, ((XtArgVal)(v))
```

See Also

UxDDGetProp(), UxDDInstall(), UxGetProp(), UxPutProp()

UxDefineResource()

Creates and initializes a new resource descriptor.

Synopsis

```
#include <resource.h>
Resource_t *UxDefineResource( [name, value,]… RD_END
   );
```

Arguments

name — One of values of the enumerated type `RD_key_t`. These values identify the fields in the resource descriptor structure. The value `RD_END` terminates the argument list.

value — The value to be stored in a field of the resource descriptor

Return Value

Returns a pointer to a new resource descriptor. The return value is usually passed as the third parameter in a call to `UxPutClassResource`.

Description

`UxDefineResource()` allocates a resource descriptor (a `Resource_t` structure) and initializes its fields according the name/value pairs in the argument list.

Example

```
#include <resource.h>
#include <label.cl.h>
   /* for UxC_label, UxP_LabelRD_alignment */
#include <swidget.h>
   /* for UxPutClassResource */
#include <valuesOf.h>
   /* for UxValuesOf, UxValidate functions */
   /* and UxUT, UxXT variables */
UxPutClassResource( UxC_label,
   UxP_LabelRD_alignment,
   UxDefineResource(
   RD_NAME, "alignment", /* You must supply a name.
   */
   RD_UTYPE, UxUT_string,
   RD_XTYPE, UxXT_Alignment,
   RD_DIVISION, UxSPECIFIC,
```

```
RD_VALIDATOR, UxValidateAlignment,
RD_VALUESOF, UxValuesOfAlignment,
RD_END ) );
```

Types

The resource descriptor is a `Resource_t` structure. This structure is defined in *uimx_directory*`/custom/include/resource.h`. The `RD_key_t` enumeration is

also defined in `resource.h`.

Key	Field / Purpose	Type
`RD_END`	Terminates the argument list.	None
`RD_EXAMPLE`	Gets a copy of the specified resource descriptor.	Resource_t*
`RD_NAME`	UimxName	char*
`RD_UTYPE`	UType	short
`RD_XTYPE`	XType	short
`RD_XTNAME`	UxXtName	char*
`RD_PRINTNAME`	PrintName	char*
`RD_UIMXNAME`	UimxName	char*
`RD_VALIDATOR`	Validator	int (*)()
`RD_VALUESOF`	ValuesOf	int (*)()
`RD_PUTFUNCTION`	PutFunction	int(*)()
`RD_GETFUNCTION`	GetFunction	int(*)()
`RD_TOOLKIT`	Toolkit	Boolean
`RD_PASS`	Pass	`Pass_t` (see below)
`RD_DIVISION`	Division	`Division_t` (see below)
`RD_READONLY`	ReadOnly	Boolean
`RD_CANBEPUBLIC`	CanBePublic	Boolean
`RD_EDITABLE`	Editable	short
`RD_NOEDITBTN`	NoEditBtn	Boolean
`RD_COPYFUNCTION`	CopyFunction	XtArgVal (*)()
`RD_FREEFUNCTION`	FreeFunction	void (*)()

Key	Field / Purpose	Type
RD_EDITFUNCTION	EditFunction	void (*)()
RD_FORMAT	Format	char*
RD_TREATMENT	Treatment	ExprTreatment_t (see below)
RD_SAVEFUNCTION	SaveFunction	void (*)()
RD_USERDATARD	UserDataRD	XtPointer

The following enumerated types are also used to initialize a resource descriptor:

```
typedef enum {
   UxPASS0 = 0, /* Widget creation argument list. */
   UxPASS1 = 1, /* Design time: creation arg list. */
   UxPASS2 = 2 /* XtSetValues after interface tree
      exists. */
} Pass_t;

typedef enum {
   UxCORE = 0,
   UxSPECIFIC = 1,
   UxBEHAVIOR = 2,
   UxCONSTRAINT = 3,
   UxDECL = 4, /* Widget declaration properties */
   UxCOMPOUND = 5 /* UIM/X internal compound props */
} Division_t;

typedef enum { /* How does the interpreter see it?
   */UxLITERAL, /* For name, other variable names
   */UxEXPRESSION, /* For resources */UxSTATEMENT /*
   For callbacks */
} ExprTreatment_t;
```

See Also

UxPutClassResource(), Appendix F, “Resource Descriptors,” for a description of the fields in the resource descriptor.

UxEnvArgResource()

Returns a resource descriptor for the CORBA environment.

Synopsis

```
#include <uxmethod.h>
Resource_t *UxEnvArgResource(void);
```

Arguments

None.

Return Value

Returns a resource descriptor for the CORBA environment.

Description

`UxEnvArgResource()` returns the resource descriptor for the CORBA environment that is needed by `UxCreateMethodSignature()` when creating the signature for Corba1 or Corba2 methods. The resource descriptor is stored in a static variable. The same one is returned for every call. The caller should not free the return value.

See Also

UxCreateMethodSignature(), UxGetArgResource()

UxFixed_class_method()

Adds a new class method for the specified swidget class.

Synopsis

```
#include <veos.h>

binptr UxFixed_class_method(char *name, Class_t cl,
   int data_type, int offset);
```

Arguments

name	Name of the method.
cl	Swidget class being initialized.
data_type	ID of data type.
offset	Offset of the method-handle within the class structure

Return Value

The return value is the id of the new method and should be stored in the `UxM_`*MethodName* global variable.

Example

```
UxM_UxWidgetCannotAcceptChildren =
   UxFixed_class_method(
   "UxWidgetCannotAcceptChildren",
   UxC_RectObject,
   T_PNTR,
   Offset(UxRectObjectClass,
   RectObject._UxWidgetCannotAcceptChildren));
```

See Also

UxInit_method()

UxFixed_class_prop()

Registers a new class property for a swidget class.

Synopsis

```
#include <veos.h>

binptr UxFixed_class_prop(char *name, Class_t cl,
   int data_type, int offset);
```

Arguments

name	The name of the property.
cl	The swidget class for which the property is being registered.
data_type	Identifies the data type of the property. For properties, this parameter is always `T_PNTR`, since the actual field in the class structure is a `Resource_t*`pointer. The `T_` types are defined in *uimx_directory*`/custom/include/vtypes.h`.
offset	Offset of the property within the class structure.

Return Value

Returns the ID of the new property. This value should be stored in a `UxP_`*PropertyName* global variable.

Example

```
UxP_CoreRD_background =
UxFixed_class_prop("RD_background",
   UxC_Core,
   T_PNTR,
   Offset(UxCoreClass,
   Core.RD_background));
```

See Also

UxPutClassResource()

UxGET_*type*()

These functions are only used in defining the DESIGN_TIME versions of the UxGet macros in the swidget class public header file.

Synopsis

```
#include <stdgetput.h>

type UxGET_type(swidget swgt, binptr bp, char *name);
```

Arguments

swgt Swidget to get value from.

bp ID of property.

name Name of property (used in error message if the property does not exist for the swidget class).

Description

The UxGET_*type*() functions call the "get_function' for the specified property to get the current property value from the specified swidget. UIM/X displays an error dialog box (by calling UxGUIError) if the property does not exist for the swidget class of the specified swidget.

One UxGET_*type*() function exists for each utype:

Function Name	Utype	Return Value
UxGET_int	UxUT_int	int
UxGET_float	UxUT_float	float
UxGET_short	UxUT_short	short
UxGET_long	UxUT_long	long
UxGET_string	UxUT_string	char *
UxGET_char	UxUT_char	char
UxGET_voidFunction	UxUT_voidFunction	void (*)()
UxGET_cardFunction	UxUT_cardFunction	Cardinal (*)()
UxGET_visualPointer	UxUT_visualPointer	Visual *
UxGET_stringTable	UxUT_stringTable	char **
UxGET_XmTextSource	UxUT_XmTextSource	XmTextSource

Example

```
alignment = UxGET_string(swgt,
   UxP_LabelRD_alignment, "alignment");
```

See Also

UxDefineResource(), UxPUT_*type*()

UxGetArgResource()

Creates a resource descriptor for an argument in a method signature.

Synopsis

```
#include <uxmethod.h>

Resource_t *UxGetArgResource(char *name, int utype,
   char *defvalue, void *validator, void *valuesof);
```

Arguments

name	The argument name as it appears in the Connection Editor.
utype	The utype of the argument.
defvalue	The default value in the Connection Editor.
validator	The Validator function.
valuesof	The ValuesOf function.

Return Value

Returns the resource descriptor for the argument.

Description

`UxGetArgResource()` is a convenience function that creates a resource descriptor tailored to the needs of a method signature. `UxGetArgResource()` can create a resource descriptor and initialize the necessary fields for one argument in a call to `UxCreateMethodSignature()`.

The return value should never be freed. It is the responsibility of the caller to supply a “name” argument that will be valid until the termination of the executable.

See Also

UxDefineResource() , UxEnvArgResource()

UxGetComponentRef()

Retrieves the component reference from an adapter swidget.

Synopsis

```
#ifdef XT_CODE
   #include <UxXt.h>
#else
   #include <UxLib.h>
#endif

void* UxGetComponentRef( swidget sw );
```

Arguments

sw An adapter swidget.

Return Value

A pointer to the external component. `UxGetComponentRef()` returns `NULL` if the swidget is not a subclass of the adapter swidget class.

Description

`UxGetComponentRef()` and `UxPutComponentRef()` are accessors for the component reference field of an adapter swidget. `UxAdapterSwidget()` sets the component reference field when it creates the adapter swidget.

The component reference field stores a pointer to an external component (a third-party object) integrated with UIM/X.

Example

The wrapper methods for the components use `UxGetComponentRef()` to get the component from an adapter swidget:

```
static int _CheckBox__set_Alignment(swidget UxThis,
   Environment *pEnv, int val)
{
   VwCheck *pCmpnt =
      (VwCheck*)UxGetComponentRef(UxThis);
   if (pEnv)
      pEnv->major(CORBA::NO_EXCEPTION);
      if (pCmpnt) {
         return((int) pCmpnt->PutAlignment(
         (VwToggleAlignment)val));
      }
   return ERROR;
```

```
}
```

See Also UxPutComponentRef()

UxGetProp()

Gets the current value of a swidget property.

Synopsis

```
#include <UxLib.h>
XtArgVal UxGetProp(swidget sw, char *name);
```

Arguments

sw	A swidget.
name	The name of the property (an XmNor XtN constant).

Return Value

UxGetProp() returns the requested property value if successful, and 0 otherwise.

Description

UxGetProp() is part of the Ux Convenience Library. This function is used to define the run-time versions of the UxGetProperty macros (in the swidget class public header files).

UxGetProp() is used when run-time conversion of property values is not required. Run-time conversion is required when the data type of the values expected by the swidget and a widget are not the same. UxDDGetProp() is used to define run-time UxGetPropert()y macros for properties that require run-time conversion.

Example

```
#define UxGetTitle(o) (char *)UxGetProp(o, XmNtitle)
```

See Also

UxDDGetProp(), UxDDPutProp(),UxPutProp(), “Run-Time Macros” in Chapter 2, “Integrating Widgets”

UxGetResourceSet()

Gets the list of resource descriptors defined for a swidget class.

Synopsis

```
#include <plist.h>
#include <swidget.h>
PList_c* UxGetResourceSet( object* obj );
```

Arguments

obj — An instance of a swidget class (a `swidget` type variable) or a swidget class ID (a `Class_t` type variable).

Return Value

`UxGetResourceSet()` returns a pointer to the swidget class' PList of resource descriptors.

Description

Each swidget class has a PList of resource descriptors named `ResourceSet`. This function returns that PList of `Resource_t*` pointers. The entries in the resource set are the defined properties for the swidget class.

The entries correspond to the properties that appear in the Property Editor for an instance of the swidget class, with the exception of the Constraints properties.

The order of the resource set is undefined.

UxGlobalInstanceResource()

Registers a global resource descriptor for a component property.

Synopsis

#include <resource.h>

void UxGlobalInstanceResource(char *propname, Resource_t *res);

Arguments

propname The property name.

res A resource descriptor.

Return Value

None.

Description

`UxGlobalInstanceResource()` registers a global resource descriptor for a component property. The registration is global because it applies to a property of that name in any component.

You can create component properties by adding arguments to the component constructor. These properties will appear in the Core category of the Property Editor.

You can also create `get` and `set` property accessor methods in the Method Editor. These properties will appear in the Specific category of the Property Editor.

For example, suppose you have built a set of components, each of which declares an interface function with an argument named `textBackground`. All instances of these components have a `textBackground` property. You can register a resource descriptor for this common property with a single call to `UxGlobalInstanceResource()` (otherwise `UxInstanceResource()` would have to be called once for each Component).

Note: Note that it is the name of the property `textBackground` that is common to all instances. The type of the argument is not necessarily the same in each component constructor. You are responsible for ensuring that the type of the component property matches the type of value expected by the Validator function (if any) in the resource descriptor.

By registering resource descriptors for the properties of instances, you can install specialized popup editors (such as the Color Viewer), option menus, and input validation routines. By default, the Property Editor only allows you to use the Text Editor to set the initial values of the properties of an instance.

When you load an instance into the Property Editor, UIM/X tries to find a registered resource descriptor for each of the properties of the instance.

For example, suppose you load an instance of a component named `formField` into the Property Editor, and that the instance has a property named `textBackground` (which appears in the Property Editor as TextBackground).

First, UIM/X tries to find a resource descriptor registered under the names `formField` and textBackground by `UxInstanceResource()`. If no such resource descriptor is found, UIM/X then tries to find a global resource descriptor registered under the name `textBackground` by `UxGlobalInstanceResource()`.

Note: Resource descriptors registered with `UxInstanceResource()` take precedence over those registered with `UxGlobalInstanceResource()`.

If UIM/X finds a registered resource descriptor, it copies the `ValuesOf`, `Validator`, and `EditFunction` fields of the registered resource descriptor into the actual resource descriptor used by the property:

The `EditFunction` field specifies a function that pops up a specialized editor.

The `ValuesOf` field can be used to install an option menu.

The `Validator`field specifies the input validation routine for the property.

Typically, you would use `UxDefineResource` to create and initialize the resource descriptors that you register. To register an existing resource descriptor, you would use one of the `UxGetRD_`*property* macros.

Note: You must register a resource descriptor for a component argument before you create the component.

Example

This example registers a resource descriptor for the property named `textBackground`. The code that does this is placed in a function called `SetupInstanceResources`.

```
#include <resource.h>
#include <uxgui.h>
#include <textF.cl.h>

/* It is assumed that these functions have been
   written. */
extern int ValuesOfTextBackground();
```

```
extern int ValidateTextBackground();


void SetupInstanceResources()
{
   /*
   * Install the Color Viewer,
   ValuesOfTextBackground, and
   * ValidateTextBackground as the defaults for the
   * textBackground property of all Instances.
   *
   * UxInstanceResource can be used to replace these
   defaults
   * for instances of a given Component.
   */


   Resource_t *res = UxDefineResource(
      /* You must supply a name. */
      RD_NAME, "TextBackground",
      RD_EDITFUNCTION, (void (*)())
      UxGUIPopupColorView,
      RD_VALUESOF, ValuesOfTextBackground,
      RD_VALIDATOR, ValidateTextBackground,RD_END);
   UxGlobalInstanceResource( "textBackground", res
      );
}
```

See Also UxDefineResource(), UxInstanceResource(), UxValuesOfXtype(), UxValidateXtype()

UxInheritedMethodUnregister()

Unregisters a method you inherit from one of your base classes.

Synopsis

```
#include <uxmethod.h>
void UxInheritedMethodUnregister(int clsCode, char
   *mname);
```

Arguments

clsCode A class code obtained from `UxNewSubclassId()`.

mname The method name.

Return Value

None.

Description

Usually, a method implemented by a base class is inherited by all its derived classes. You might find cases where you would like to implement a method in a base class, but it should not be inherited by all its derived classes. At design time, you can turn off inheritance of the method by calling `UxInheritedMethodUnregister()` with the derived class code and method name.

The Connection Editor builds its list of methods by searching for all registered method signatures on the target class and all its base classes. If it finds that a method was unregistered, it will remove it from the list.

See Also

UxMethodRegister(), UxMethodSignatureRegister()

UxInheritResources()

Gives a swidget class the properties defined by its superclasses.

Synopsis

```
#include <swidget.h>
void UxInheritResources(Class_t cl);
```

Arguments

cl A swidget class ID.

Return Value

None.

Description

`UxInheritResources()` initializes the swidget class' PLists of resource descriptors. A swidget class has two PLists of resource descriptors: its resource set and its constraint set. `UxInheritResources()` gives a swidget class its own copies of the superclass' resource and constraint sets. The actual resource descriptors, however, are shared by the classes.

`UxInheritResources()` must be called when you register a new swidget class. Until `UxInheritResources()` is called, a class and its superclass also share the same copy of the resource and constraint sets.

Note: After `UxInheritResources()` has been called, `UxPutClassResource()` can be used to give a derived class its own resource descriptor for an inherited property.

See Also

"Defining New Xtypes" in Chapter 2, "Integrating Widgets", UxPutClassResource()

UxInit_method()

Installs a function as a class method for a swidget class.

Synopsis

```
#include <veos.h>
void UxInit_method( Class_t cl, binptr method_id,
   void (*fcn)() );
```

Arguments

cl Swidget class being initialized.

method_id ID of method (the return value from `UxFixed_class_method`).

fcn Function to be used.

Example

```
UxInit_method( UxC_label, UxM_Init, init_label );
```

See Also

`UxFixed_class_method()`

UxInstanceResource()

Registers a resource descriptor for a property of a given component.

Synopsis

```
#include <resource.h>
void UxInstanceResource(char *component, char
   *propname, Resource_t *res);
```

Arguments

component	The name of the Component.
propname	The property name.
res	A resource descriptor.

Return Value

None.

Description

`UxInstanceResource()` registers a resource descriptor for a property of a given component. The descriptor will apply to all instances of that component.

You can create component properties by adding arguments to the component constructor. These properties will appear in the Core category of the Property Editor.

You can also create `get` and `set` property accessor methods in the Method Editor. These properties will appear in the Specific category of the Property Editor.

By registering resource descriptors for the properties of instances, you can install specialized popup editors (such as the Color Viewer), option menus, and input validation routines. The Property Editor only allows you to use the Text Editor to set the initial values of these properties.

When you load an Instance into the Property Editor, UIM/X tries to find a registered resource descriptor for each of the properties of the instance.

For example, suppose you load an instance of a component named formField into the Property Editor, and that the Instance has a Specific property named `textBackground` (which appears in the Property Editor as TextBackground).

First, UIM/X tries to find a resource descriptor registered under the names formField and textBackground by `UxInstanceResource()`. If no such resource descriptor is found, UIM/X then tries to find a global resource descriptor registered under the name `textBackground`by `UxGlobalInstanceResource()`.

Note: Resource descriptors registered with `UxInstanceResource()` take precedence over those registered with `UxGlobalInstanceResource()`.

If UIM/X finds a registered resource descriptor, it copies the `ValuesOf`, `Validator`, and `EditFunction` fields of the registered resource descriptor into the actual resource descriptor used by the property:

- The `EditFunction` field specifies a function that pops up a specialized editor.
- The `ValuesOf` field can be used to install an option menu.
- The `Validator`field specifies the input validation routine for the property.

Typically, you would use `UxDefineResource()` to create and initialize the resource descriptors that you register. To register an existing resource descriptor, you would use one of the `UxGet_RD` macros.

Note: You must register a resource descriptor for a Component argument before you create the Component.

Example

This example registers resource descriptors for two of the properties of a Component named formField. The code that does this is placed in a function called `SetupInstanceResources`.

```
#include <resource.h>
#include <textF.cl.h>

static int ValuesOfLabels();

void SetupInstanceResources()
{
   /* Install the Color Viewer for the textColor
   argument
   * of the formField Component. The EditFunction
   field of the
   * resource descriptor returned by
   UxGetRD_background holds
   * the popup function for the Color Viewer.
   *
   * The Validator and ValuesOf fields of the
   background
   * property are appropriate for our textBackground
```

```
   property
   * as well, so we'll use the background resource
   descriptor.
    */

   Resource_t *res = UxGetRD_background(
   UxC_textField );
   UxInstanceResource( "formField",
   "textBackground", res );


/* Install an option menu for the labelString
   argument. */

res = UxDefineResource(
   /* You must supply a name. */
   RD_NAME, "labelString",
   RD_VALUESOF, ValuesOfLabels,RD_END);
UxInstanceResource( "formField", "labelString", res
   );
}

static int ValuesOfLabels( char *** menuItems, int*
   numItems )
{
   static char *Items[] = { "Name", "Date", "Phone"
      };
   *menuItems = Items;
   *numItems = XtNumber( Items );


   return *numItems;
}
```

See Also UxDefineResource(), UxGlobalInstanceResource(), UxValuesOfXtype(), UxValidateXtype()

UxIsInterface()

This function detects top-level swidgets.

Synopsis

```
#include <swidget.h>
int UxIsInterface (swidget sw);
```

Arguments

sw An adapter swidget.

Return Value

This function returns `true` if `sw` is the top-level swidget in an interface; `false` otherwise.

Description

A swidget is the top-level swidget in an interface if it is one of the following:

- a subclass of Shell
- a convenience dialog created from the Dialog creation menu
- a manager with an implicit shell supplied by UIM/X

The USER widgets for which `UxIsInterface()` returns `true` correspond to the interfaces shown in the project window; the name of the shell swidget is the name of the interface window; the name of top-level swidget appears on the interface icon.

See Also

`UxGetParent` in the *UIM/X Reference Manual*

UxIsSubclass()

This function determines if an object is an instance of a given class or a subclass of that class.

Synopsis

```
#include <veos.h>
int UxIsSubclass (object* obj, Class_t cl);
```

Description

This function returns `true` if the argument is an instance of the class `cl` or a class derived from `cl`; `false` otherwise.

Example

```
UxIsSubclass (sw, UxC_manager);
```

returns true if and only if the given swidget sw is an instance of any manager class (such as bulletinBoard, form, or fileSelectionBox).

UxLoadGlobalInclude()

Load an include file into the global reference environment.

Synopsis

```
#include <UxLib.h>
void UxLoadGlobalInclude( char *include_file );
```

Arguments

include_fileThe name of the include file.

Return Value

None.

Description

`UxLoadGlobalInclude()` includes a file in the reference translation unit, which is a collection of definitions shared by all translation units. UIM/X uses the reference translation unit to include the standard UIM/X, Motif, Xt, X, and system header files for use in all translation units.

Example

```
#ifdef DESIGN_TIME
UxLoadGlobalInclude("xkcheck.h");
#endif
```

UxMethodLookup()

Retrieves the implementation of a method for a given class.

Synopsis

```
#ifdef XT_CODE
   #include <UxXt.h>
#else
   #include <UxLib.h>
#endif
void* UxMethodLookup(swidget sw, int mid, char
   *mname);
```

Arguments

sw	A swidget.
mid	A method ID obtained from `UxMethodRegister()`.
mname	The method name.

Return Value

Returns the function pointer for the method implementation registered for the swidget's interface class.

If the lookup fails at run time, `UxMethodLookup()` returns a pointer to a function that always returns `0`. If the lookup fails at design time, `UxMethodLookup()` returns `0`.

Description

`UxMethodLookup()` finds the implementation of a method for a given class. The class is obtained from the swidget `sw`, which is typically the top-level swidget of an interface.

If there is no entry in the method table for the given class, `UxMethodLookup()` searches the method table for a superclass version of the method.

Note that you can call `UxMethodLookup()` without a method ID. If you pass `-1` as the method ID, `UxMethodLookup()` searches an internal table of method names for the method ID corresponding to `mname`.

Example

Generated C code defines method invocation macros that use `UxMethodLookup()`:

```
#ifndef bulletinBoard1__set_Background
#define bulletinBoard1__set_Background( UxThis,
   pEnv, color ) \
   ((int(*)())UxMethodLookup(UxThis, \
```

```
    UxbulletinBoard1__set_Background_Id,\
    UxbulletinBoard1__set_Background_Name)) \
    ( UxThis, pEnv, color )
#endif
```

See Also UxMethodRegister(), UxNewInterfaceClassId(), UxNewSubclassId()

UxMethodRegister()

Registers a method for a given class.

Synopsis

```
#ifdef XT_CODE
   #include <UxXt.h>
#else
   #include <UxLib.h>
#endif

int* UxMethodRegister(int clsCode, char *mname,
   void (*function)() );
```

Arguments

clsCode	A class code obtained from `UxNewInterfaceClassId()` or `UxNewSubclassId()`.
mname	The method name.
function	The method implementation.

Return Value

Returns a unique ID.

Description

`UxMethodRegister()` registers a method by storing the function pointer in a method table. This table is indexed by class code and method ID.

`UxMethodRegister()` maps each method name to a unique method ID. So when you register the same method for different classes (for example, when you override an inherited method), `UxMethodRegister()` returns the same method ID.

Note that `UxMethodRegister()` stores the method names in an internal table, and uses the method ID as an index into the table.

Example

In generated C code, the source file defines two variables for each method:

```
int UxbulletinBoard1__set_Background_Id = -1;
char* UxbulletinBoard1__set_Background_Name =
   "_set_Background";
```

The `Id` variable holds the ID returned by `UxMethodRegister()`, and the `Name` variable holds the method name. The method is actually registered in the interface's generated Interface Function:

```
static int _UxIfClassId;
```

```
swidget create_bulletinBoard1( swidget _UxUxParent )
{
   swidget rtrn;
   _UxCbulletinBoard1 *UxContext;
   static int _Uxinit = 0;

   UxBulletinBoard1Context = UxContext =
      (_UxCbulletinBoard1 *) UxNewContext(
      sizeof(_UxCbulletinBoard1), False );

UxParent = _UxUxParent;
if ( ! _Uxinit )
{
   _UxIfClassId = UxNewInterfaceClassId();
   UxbulletinBoard1__set_Background_Id =
   UxMethodRegister(
      _UxIfClassId,UxbulletinBoard1__set_Background_
      Name,(void(*)())_bulletinBoard1__set_Backgroun
      d );
      _Uxinit = 1;
}
   rtrn = _Uxbuild_bulletinBoard1();
   return(rtrn);
}
```

Note: The method is registered in a one-time only block of code.Note also that method registration is not performed in generated C++ code, as method lookups are not necessary in C++.

See Also UxInheritedMethodUnregister(), UxMethodLookup(), UxNewInterfaceClassId(), UxNewSubclassId()

UxMethodSignatureRegister()

Registers the signature for a method.

Synopsis

```
#include "uxmethod.h"

int UxMethodSignatureRegister(int clsCode, char
   *mname,struct Method_t *signature);
```

Arguments

clsCode — A class code obtained from `UxNewInterfaceClassId()` or `UxNewSubclassId()`.

mname — The method name.

signature — The method signature.

Return Value

Returns the method ID.

Description

A method signature is the description of the arguments of a method and its return type. The builder needs this information at design time to determine how to call compiled-in methods and how to display them in the Connection Editor.

`UxMethodSignatureRegister()` registers the signature that you obtain from `UxCreateMethodSignature()`. This is usually done immediately after you register the method with `UxMethodRegister()`. For backward compatibility, if you do not register a signature, the builder will assume it is a CORBA 1 method, and it will not be available in the Connection Editor.

The builder can automatically generate calls to `UxMethodSignatureRegister` if you generate Ux Integration Code. You might have to edit the generated code if you need your own validators and ValuesOf functions for the method arguments. See `UxGetArgResource()` for details.

Example

```
int cid, mid;

cid = UxNewClassId();

mid = UxMethodRegister(cid, "_set_height",
   _set_height);

UxMethodSignatureRegister(cid, "_set_height",

 UxCreateMethodSignature("_set_height", Corba1,
   NULL,

      UxEnvArgResource(),

     UxGetArgResource("height", UxUT_int, "0",
```

```
            UxValidateInt, UxValuesOfInt),
        NULL);
```

See Also UxCreateMethodSignature(), UxEnvArgResource(), UxGetArgResource(), UxMethodRegister()

UxNewInterfaceClassId()

Registers a new interface class.

Synopsis

```
#ifdef XT_CODE
   #include <UxXt.h>
#else #
   include <UxLib.h>
#endif
int UxNewInterfaceClassId( void );
```

Arguments

None.

Return Value

Returns the class code.

Description

`UxNewInterfaceClassId()` registers an interface class as a subclass of the abstract base class `UxVisualInterface` (sometimes referred to as the Interface class). Every interface class and component in UIM/X is a subclass of

UxVisualInterface.

`UxNewInterfaceClassId()` registers the methods of the Interface class:

- Default versions of standard accessor methods for x, y, width, and height.
- A default `UxManage()` method.

Example

In generated C code, the Interface Function for an interface class calls `UxNewInterfaceClassId()` to get a class code, which it then uses to register methods:

```
_UxIfClassId = UxNewInterfaceClassId();
UxbulletinBoard1__set_Background_Id =
   UxMethodRegister(
   _UxIfClassId,
   UxbulletinBoard1__set_Background_Name,
   _bulletinBoard1__set_Background );
```

See Also

UxMethodLookup(), `UxMethodRegister`(), `UxNewSubclassId`()

UxNewSubclassId()

Registers a new class as a subclass of an existing class.

Synopsis

```
#ifdef XT_CODE
   #include <UxXt.h>
#else
   #include <UxLib.h>
#endif

int UxNewSubclassId( int super );
```

Arguments

super The class code for the superclass of the new subclass.

Return Value

Returns the class code for the new subclass.

Description

You use `UxNewSubclassId()` to register a class as a subclass of another class. The subclass inherits the methods of its superclass.

This means that if you pass an instance of the new subclass to `UxMethodLookup()`, UIM/X invokes the superclass' version of the method.

See Also

UxNewInterfaceClassId()

UxPUT_*type*()

These functions are only used in defining the DESIGN_TIMEversions of the UxPut macros in the swidget class public header file.

Synopsis

```
#include <stdgetput.h>

int UxPUT_type (swidget sw, binptr bp, char* name, type
   value);
```

Arguments

sw	Swidget to put value on.
bp	ID of property.
name	Name of property. Used in error message if the property does not exist for the swidget class.
value	Value to put.

Return Value

UxPUT_*type*() returns NO_ERROR if successful, ERROR otherwise.

Description

The UxPUT_*type*() functions call the put_function for the specified property to put the current property value from the specified swidget. UIM/X displays an error dialog box (by calling UxGUIError) if the property does not exist for the swidget class of the specified swidget.

One UxPUT_*type*() function exists for each utype:

Function Name	Utype	Value Type
UxPUT_int	UxUT_int	int
UxPUT_float	UxUT_float	float
UxPUT_short	UxUT_short	short
UxPUT_long	UxUT_long	long
UxPUT_string	UxUT_string	char *
UxPUT_char	UxUT_char	char
UxPUT_voidFunction	UxUT_voidFunction	void (*)()
UxPUT_cardFunction	UxUT_cardFunction	Cardinal (*)()
UxPUT_visualPointer	UxUT_visualPointer	Visual *
UxPUT_stringTable	UxUT_stringTable	char **
UxPUT_XmTextSource	UxUT_XmTextSource	XmTextSource

Example

```
status = UxPUT_string(swgt, UxP_LabelRD_alignment,
   "alignment", "alignment_end");
```

See Also UxDefineResource(), UxGET_*type*()

UxPutClassResource()

Installs a resource descriptor for a swidget class property.

Synopsis

```
#include <swidget.h>

void UxPutClassResource(Class_t cl, binptr bp,
   Resource_t *res);
```

Arguments

cl	Swidget class ID.
bp	ID of the class property.
res	Resource descriptor to install.

Return Value

None.

Description

`UxPutClassResource()` installs a resource descriptor as a class property of the swidget class `cl`. Existing resource descriptors—for example, the resource descriptors of inherited properties—are replaced for the specified class.

`UxPutClassResource()` should be called once for each resource during class initialization.

Example

The following code installs a new `ValuesOf` function for all pushButtons:

```
#include <pushB.cl.h>

extern int ValuesOfBackgroundColor ( char ***values,
   int *nentries );

UxPutClassResource( UxC_pushButton,
   UxP_CoreRD_background,

   UxDefineResource(

      RD_EXAMPLE, UxGetRD_background( UxC_pushButton
   ),

      RD_VALUESOF, ValuesOfBackgroundColor,

      RD_END ) );
```

UxPutComponentRef()

Sets the component reference for an adapter swidget.

Synopsis

```
#ifdef XT_CODE
   #include <UxXt.h>
#else
   #include <UxLib.h>
#endif

void UxPutComponentRef( swidget sw, void *ref );
```

Arguments

sw	An adapter swidget.
ref	The component reference.

Return Value

None.

Description

`UxGetComponentRef()` and `UxPutComponentRef()` are accessors for the component reference field of an adapter swidget. `UxAdapterSwidget()` sets the component reference field when it creates the adapter swidget.

See Also

UxGetComponentRef()

UxPutIconBitmap()

The macro `UxPutIconBitmap()` specifies the name of the bitmap file to be used to represent a swidget class. It should be called once at class initialization.

Synopsis

```
#include <swidget.cl.h>
void UxPutIconBitmap(Class_t cl, char* bitmap_file);
```

Arguments

cl Swidget class being initialized.

bitmap_fileName of bitmap file.

Return Value

None.

Example

```
UxPutIconBitmap(UxC_label, "stext.bm");
```

UxPutProp()

Sets the current value of a swidget property.

Synopsis

```
#include <UxLib.h>
int UxPutProp(swidget sw, char *prop, XtArgVal
   value);
```

Arguments

sw	A swidget.
prop	The name of the property (an XmNor XtN constant).
value	The property value.

Return Value

Returns NO_ERROR if successful, ERROR otherwise.

Description

UxPutProp() is part of the Ux Convenience Library. This function is used to define the run-time versions of the UxPutProperty() macros (in the swidget class public header files).

UxPutProp() is used when run-time conversion of property values is not required. Run-time conversion is required when the data type of the values expected by the swidget and a widget are not the same. UxDDPutProp() is used to define run-time UxPutProperty() macros for properties that require run-time conversion.

Example

```
#define UxPutIconY(o, v) \
   UxPutProp(o, XmNiconY, ((XtArgVal)(v)))
```

See Also

UxDDGetProp(), UxDDPutProp(), UxGetProp(), “Run-TimeMacros” in Chapter 2, “Integrating Widgets”

UxPutToolKitClass()

Specifies the widget class that corresponds to a swidget class.

Synopsis

```
#include <swidget.cl.h>
void UxPutToolKitClass(Class_t cl, WidgetClass
   wgt_class);
```

Arguments

cl swidget class being initialized

wgt_class corresponding widget class

Return Value

None.

Description

The macro `UxPutToolKitClass()` specifies the widget class that corresponds to a swidget class.

Example

```
UxPutToolKitClass(UxC_label, XmLabelWidgetClass);
```

UxPutUxFilename()

Sets the name of the public header file for a swidget class.

Synopsis

```
#include <veos.h>
void UxPutUxFilename( Class_t cl, char *filename );
```

Arguments

cl	The class ID of a swidget class.
filename	The name of the swidget class' public header file.

Return Value

None.

Description

`UxPutUxFilename()` sets the class property that specifies the name of the public header file for the swidget class. This function must be called in the function which registers and initializes the swidget class.

Example

```
UxPutUxFilename( UxC_label, "<UxLabel.h>" );
```

See Also

"The Class Structure" in Chapter 2, "Integrating Widgets"

UxRegister_class()

Registers a new swidget class.

Synopsis

```
#include <veos.h>

Class_t UxRegister_class(char *name, Class_t
   superclass, int instance_size, int class_size);
```

Arguments

name Name of swidget class.

superclass Swidget superclass.

instance_size Size of instance structure in bytes.

class_size Size of class structure in bytes.

Return Value

Returns the swidget class ID.

Description

Registers a swidget class. If a class already exists with the same name, superclass, instance size, and class size, the class is considered to already be registered.

Example

```
UxC_label = UxRegister_class("label", UxC_primitive,
   sizeof(label), sizeof(UxlabelClass));
```

Ux*Type*_get_op()

Looks up a swidget class method and returns a function pointer.

Synopsis

```
#include <veos_d.h>

void (*UxVoid_get_op(Object_t cl, binptr
   method_id))(void);

int (*UxInt_get_op(Object_t cl, binptr
   method_id))(void);

char* (*UxPNTR_get_op(Object_t cl, binptr
   method_id))(void);
```

Arguments

obj Class or swidget whose method is to be retrieved.

method_id ID of method.

Description

The Ux*Type*_get_op() functions look up the function that was previously installed for a given method and return a pointer to a function returning type.

Examples

```
swidget swgt;


UxVoid_get_op(UxC_label, UxM_Init) (swgt);

char *msg = UxPNTR_get_op
   (UxC_form, UxM_InteractiveChildCreate) (swgt);


swidget swgt_to_recreate =
   (swidget) UxPNTR_get_op
   (UxC_label, UxM_RecreateParentOrChild) (swgt);
```

See Also

UxFixed_class_method(), UxInit_method()

UxValidateXtype()

Validates a property value of a specific xtype.

Synopsis

```
#include <valuesOf.h>
int UxValidateXtype(swidget swgt, type value);
```

Arguments

swgt A swidget.

value A property value.

Return Value

Returns `NO_ERROR` if `value`is valid, and `ERROR` otherwise.

Description

Validation functions are used by UIM/X to check that the values given when setting a property on a swidget instance are valid values. Validation functions are specified (with `RD_VALIDATOR`) when a resource descriptor is defined by `UxDefineResource()`. The name of the validation function is conventionally `UxValidate`Xtype() where Xtype is the name of the property xtype being validated. All validation functions should follow the format shown above.

Example

```
int UxValidatePositiveInt(swidget swgt, int value)
{
if (value <= 0)
return (ERROR);
else
return (NO_ERROR);
}
```

See Also

UxDefineResource(), UxValuesOfXtype()

UxValuesOfXtype()

Describes the allowable values for a property of a given xtype.

Synopsis

```
#include <valuesOf.h>
int UxValuesOfXtype(char ***values, int* num_strings);
```

Arguments

values Array of permissible values or a description of permissible values.

num_strings Number of strings in the array `values`.

Return Value

If the array `values` holds a list of permissible values (as it should for an enumerated type property), the return value should be the number of permissible values. Note that the list of permissible values may be followed by other strings holding additional descriptive information, so that the return value is not necessarily equal to '*num_strings'.

If the `values`array holds strings that describe the permissible values, the return value should be `0`.

Description

ValuesOf functions are used by UIM/X when a description of the permissible values for a property is needed. For enumerated type properties, the ValuesOf function is used to supply the list of choices that appears in the Property Editor. ValuesOf functions are specified (with `RD_VALUESOF`) when a resource descriptor is defined by `UxDefineResource()`. The name of the ValuesOf function is conventionally `UxValuesOf`Xtype() where Xtype is the name of the property `xtype`. All ValuesOf functions should follow the format shown above.

Examples

```
int UxValuesOfPositiveInt(char ***vals, int *n)
{
   **vals = "<positive integer>";
   *n = 1;
   return (0);
}
int UxValuesOfBoolean(char ***vals, int *n)
{
   static char *boolean_values[] = {"true", "false"};

   *vals = boolean_values;
```

```
        *n = XtNumber(boolean_values);
        return (*n);
    }
```

See Also UxDefineResource, UxValidateXtype

Index

Index

A

B

C

D

E

F

G

H

I

Index

V

W

X

Index