ULTRIX Worksystem Software

digital

Guide to the X Toolkit Widgets:
C Language Binding

Order Number: AA-MF09A-TE

ULTRIX Worksystem Software Guide to the X Toolkit Widgets: C Language Binding

Order No. AA-MF09A-TE

ULTRIX Worksystem Software, Version 2.0

Digital Equipment Corporation

This manual was written and produced by the ULTRIX Documentation Group in Nashua, New Hampshire.

Contents

About This Manual

The *Guide to the X Toolkit Widgets: C Language Binding* describes the "Athena" widget set that you can use to write X Toolkit-based application programs. Note that the information provided is specific to the C programming language.

Audience

The audience for this manual is the application programmer who will use the Athena widget set with the Intrinsics to build an X Toolkit-based application.

This manual does not attempt to teach how to write an XUI application, nor does it attempt to teach C programming concepts.

Organization

The *Guide to the X Toolkit Widgets* contains the following:

Chapter 1	Athena Widgets and Intrinsics Provides a general overview of the X Toolkit.
Chapter 2	Using Widgets Describes how to initialize the X toolkit, create a widget, map widgets, destroy widgets, obtain or modify widget resource values, and use callbacks. In addition, it discusses the common arguments (resources) that are associated with all of the Athena widgets.
Chapter 3	Athena Widget Set Describes, in detail, the eleven Athena widgets: Command widget, Label widget, Text widget, Scrollbar widget, Viewport widget, Box widget, VPaned widget, Form widget, Dialog widget, List widget, and Grip widget.
Chapter 4	Creating a Custom Widget Provides programming hints for writing your own X Toolkit-based widget.

Related Documents

XUI Style Guide

Describes the XUI user interface and, hence, the "look and feel" of an XUI application.

Guide to the XUI Toolkit: C Language Binding

Describes the XUI Toolkit widget set that you can use to write your XUI-based application.

Guide to the Xlib Library: C Language Binding

Describes the low-level C functions that you can use to write your X-based application.

X Window System Protocol: X Version 11

Describes the precise semantics of the X11 protocol specification.

Conventions

The following typeface conventions are used in this manual:

special	In text, all function names, events, errors, constant names, and pathnames are presented in this type.
UPPERCASE	Although the ULTRIX system differentiates between lowercase and uppercase characters, uppercase is used intentionally in this manual where it is applicable.
boldface	The primary occurrence for a given index entry is in this type.

In addition, each function is introduced by a general discussion that distinguishes it from other functions. The function declaration itself follows, and each argument is specifically explained. General discussion of the function, if any is required, follows the arguments.

Athena Widgets and Intrinsics 1

The Athena widget set and the Intrinsics make up the X Toolkit. In the X Toolkit, a widget is the combination of an X window or subwindow and its associated input and output semantics. The Athena widgets provide the base functionality necessary to build a wide variety of application environments. Because the Intrinsics mask implementation details from the widget and application programmer, the Athena widgets and the application environments built with them are fully compatible with the other widget sets built with the Intrinsics. For information about the Intrinsics, see the *Guide to the XUI Toolkit Intrinsics*.

The Athena widget set is a library package layered on top of the Intrinsics and Xlib. This layer extends the basic abstractions provided by X and provides the next layer of functionality primarily by supplying a cohesive set of sample widgets.

To the extent possible, the X Toolkit is policy free. The application environment, not the X Toolkit, defines, implements, and enforces:

- Policy
- Consistency
- Style

Each individual widget implementation defines its own policy. The X Toolkit design allows for but does not necessarily encourage the free mixing of radically differing widget implementations.

1.1 Introduction to the X Toolkit Library

The X Toolkit library provides tools that simplify the design of application user interfaces in the X Window System programming environment. It assists application programmers by providing a set of common underlying user-interface functions. It also lets widget programmers modify existing widgets or add new widgets. By using the X Toolkit library in their applications, programmers present a similar user interface across applications to all workstation users.

The X Toolkit consists of:

- A set of Intrinsics functions for building widgets
- An architectural model for constructing widgets
- A sample interface (widget set) for programming

While the majority of the Intrinsics functions are intended for the widget programmer, a subset of the Intrinsics functions are to be used by application programmers (see the *Guide to the XUI Toolkit Intrinsics*). The architectural model lets the widget programmer design new widgets by using the Intrinsics and by combining other widgets. The application interface layers built on top of the X Toolkit include a coordinated set of widgets and composition policies. Some of these widgets and policies are specific to an application domain, and others are common across a number of application domains.

The X Toolkit also can implement one or more application interface layers to:

- Verify the toolkit architecture
- Provide a base set of widgets and composition policies that can be incorporated in other application interface layers
- Make the X Toolkit immediately usable by those application programmers who find that a supplied application interface layer meets their needs

The remainder of this chapter discusses the X Toolkit:

- Terminology
- Model
- Design principles and philosophy

1.2 Terminology

In addition to the terms already defined for X programming (see the *Guide to the Xlib Library*), the following terms are specific to the Intrinsics and used throughout this book.

Application programmer

A programmer who uses the X Toolkit to produce an application user interface.

Child

A widget that is contained within another ("parent") widget.

Class

The general group to which a specific object belongs.

Client

A function that uses a widget in an application or for composing other widgets.

Full name

The name of a widget instance appended to the full name of its parent.

Instance

A specific widget object as opposed to a general widget class.

Method

The functions or procedures that a widget class implements.

Name

The name that is specific to an instance of a widget for a given client.

Object

A software data abstraction consisting of private data and private and public functions that operate on the private data. Users of the abstraction can interact with the object only through calls to the object's public functions. In the X Toolkit, some of the object's public functions are called directly by the application, while others are called indirectly when the application calls the common Intrinsics functions. In general, if a function is common to all widgets, an application uses a single Intrinsics function to invoke the function for all types of widgets. If a function is unique to a single widget type, the widget exports the function as another "Xt" function.

Parent

A widget that contains at least one other ("child") widget. A parent widget is also known as a composite widget.

Resource

A named piece of data in a widget that can be set by a client, by an application, or by user defaults.

Superclass

A larger class of which a specific class is a member. All members of a class are also members of the superclass.

User

A person interacting with a workstation.

Widget

An object providing a user-interface abstraction (for example, a Scrollbar widget).

Widget class

The general group to which a specific widget belongs, otherwise known as the type of the widget.

Widget programmer

A programmer who adds new widgets to the X Toolkit.

1.3 Underlying Model

The underlying architectural model is based on the following premises:

Widgets are X windows

Every user-interface widget is contained in a unique X window. The X window ID for a widget is readily available from the widget ID, so standard Xlib window manipulation procedures can operate on widgets.

Information hiding

The data for every widget is private to the widget and its subclasses. That is, the data is neither directly accessible nor visible outside of the module implementing the widget. All program interaction with the widget is performed by a set of operations (methods) that are defined for the widget.

Widget semantics and widget layout geometry

Widget semantics are clearly separated from widget layout geometry. Widgets are concerned with implementing specific user-interface semantics. They have little control over issues such as their size or placement relative to other widget peers. Mechanisms are provided for associating geometric managers with widgets and for widgets to make suggestions about their own geometry.

1.4 Design Principles and Philosophy

The X Toolkit follows two design principles throughout, which cover languages and language bindings as well as widget IDs.

1.4.1 Languages and Language Bindings

The X Toolkit facilitates access from objective languages. However, the X Toolkit library is conveniently usable by application programs written in nonobjective languages. Procedural interface guidelines are required when the X Toolkit is used with nonobjective languages.

The guidelines for the procedural interfaces are:

- Strings are passed as null-terminated character arrays.
- Most other arrays are passed using two parameters: a size and a pointer to the first element.
- Most numeric arguments are passed by value.
- Structures as arguments are avoided, unless a method for building them is provided for languages without pointers. Pointers embedded in structures are allowed, but they should be avoided if an equivalent alternative is available.
- Pointers are not recommended as return arguments, unless they will never have to be dereferenced by the caller. If they need to be dereferenced, the caller should allocate storage and pass the address to the procedure to fill in.
- Procedures can be passed as parameters.
- The ownership of dynamically allocated storage is determined on a case-by-case basis. The application is also permitted to replace the standard memory allocation and freeing routines used by the library at build time.

1.4.2 Widget IDs

All references to widgets use a unique identifier that is known as the widget ID. The widget ID is returned to the client by the **XtCreateWidget** function. From an application programmer's perspective, a widget ID is an opaque data type; no particular interpretation can be assigned to it. Given a widget ID, you can retrieve the corresponding X window ID, the **Display** and **Screen** structures, and other information by using Intrinsics functions.

From a widget programmer's perspective, the widget ID actually is a pointer to a data structure known as the widget instance record. Several parts of the data structure are common to all widget types, while other parts are unique to a particular widget type. The widget's private data that is associated with a particular widget instance normally is included directly in the widget instance record.

Using Widgets 2

Widgets serve as the primary tools for building a user interface or application environment. The widget set consists of primitive widgets (for example, a command button) and composite widgets (for example, a Dialog widget).

The remaining chapters of this guide explain the widgets and the geometry managers that work together to provide a set of user-interface components. These user-interface components serve as a default interface for application programmers who do not want to implement their own widgets. In addition, they serve as examples or a starting point for those widget programmers who, using the Intrinsics mechanisms, want to implement alternative application programming interfaces.

This chapter discusses the common features of the X Toolkit widgets.

2.1 Initializing the Toolkit

You must invoke the toolkit initialization function **XtInitialize** before invoking any other toolkit routines. **XtInitialize** opens the X server connection, parses standard parts of the command line, and creates an initial widget that is to serve as the root of a tree of widgets that will be created by this application.

```
Widget XtInitialize(shell_name, application_class, options,
                        num_options, argc, argv)
        String shell_name;
        String application_class;
        XrmOptionDescRec options[];
        Cardinal num_options;
        Cardinal *argc;
        String argv[];
```

shell_name Specifies the name of the application shell widget instance, which usually is something generic like "main".

application_class
Specifies the class name of this application, which usually is the generic name for all instances of this application. By

convention, the class name is formed by reversing the case of the application's first significant letter. For example, an application named "xterm" would have a class name of "XTerm".

options	Specifies how to parse the command line for any application-specific resources. The options argument is passed as a parameter to XrmParseCommand. For further information, see the *Guide to the Xlib Library*.
num_options	Specifies the number of entries in the options list.
argc	Specifies a pointer to the number of command line parameters.
argv	Specifies the command line parameters.

For further information about this function, see the *Guide to the XUI Toolkit Intrinsics*.

2.2 Creating a Widget

Creating a widget is a three-step process. First, the widget instance is allocated, and various instance-specific attributes are set by using XtCreateWidget. Second, the widget's parent is informed of the new child by using XtManageChild. Finally, X windows are created for the parent and all its children by using XtRealizeWidget and specifying the top-most widget. The first two steps can be combined by using XtCreateManagedWidget. In addition, XtRealizeWidget is automatically called when the child becomes managed if the parent is already realized.

To allocate and initialize a widget, use XtCreateWidget.

```
Widget XtCreateWidget(name, widget_class, parent, args,
                                num_args)
      String name;
      WidgetClass widget_class;
      Widget parent;
      ArgList args;
      Cardinal num_args;
```

name	Specifies the instance name for the created widget that is used for retrieving widget resources.
widget_class	Specifies the widget class pointer for the created widget.
parent	Specifies the parent widget ID.
args	Specifies the argument list. The argument list is a variable-length list composed of name and value pairs that contain information pertaining to the specific widget instance being

	created. For further information, see Section 2.7.2.
num_args	Specifies the number of arguments in the argument list. When the num_args is zero, the argument list is never referenced.

When a widget instance is successfully created, the widget identifier is returned to the application. If an error is encountered, the XtError routine is invoked to inform the user of the error.

For further information, see the *Guide to the XUI Toolkit Intrinsics.*

2.3 Common Arguments in the Widget Argument List

Although a widget can have unique arguments that it understands, all widgets have common arguments that provide some regularity of operation. The common arguments allow arbitrary widgets to be managed by higher-level components without regards to the individual widget type. All widgets ignore any argument that they do not understand.

The following resources are retrieved from the argument list or from the resource database by all X Toolkit widgets:

Name	**Type**	**Default**
XtNbackground	Pixel	XtDefaultBackground
XtNbackgroundPixmap	Pixmap	None
XtNborderColor	Pixel	XtDefaultForeground
XtNborderPixmap	Pixmap	None
XtNborderWidth	Dimension	1
XtNdestroyCallback	XtCallbackList	NULL
XtNheight	Dimension	Widget dependent
XtNmappedWhenManaged	Boolean	True
XtNsensitive	Boolean	True
XtNtranslations	TranslationTable	None
XtNwidth	Dimension	Widget dependent
XtNx	Position	0
XtNy	Position	0

XtNbackground	Specifies the window's background color.
XtNbackgroundPixmap	Specifies the window's background pixmap.
XtNborderColor	Specifies the window's border color.
XtNborderPixmap	Specifies the window's border pixmap.

XtNborderWidth	Specifies the width of the border in pixels.
XtNdestroyCallback	Specifies the callback for XtDestroyWidget.
XtNheight	Specifies the height of the widget.
XtNmappedWhenManaged	Specifies whether XtMapWidget is automatic.
XtNsensitive	Specifies whether the widget should receive input.
XtNtranslations	Specifies the event-to-action translations.
XtNwidth	Specifies the width of the widget.
XtNx	Specifies the x coordinate within the parent.
XtNy	Specifies the y coordinate within the parent.

The following additional resources are retrieved from the argument list or from the resource database by many X Toolkit widgets:

Name	Type	Default
XtNcallback	XtCallbackList	NULL
XtNcursor	Cursor	None
XtNforeground	Pixel	XtDefaultForeground

XtNcallback	Specifies the callback functions and client data.
XtNcursor	Specifies the pointer cursor.
XtNforeground	Specifies the foreground color.

The value for the XtNcursor resource can be specified in the resource database as a string, which can be specified as one of the following:

- A standard X cursor name from <X11/cursorfont.h>
- FONT font-name glyph-index [[font-name] glyph-index]
- A relative or absolute file name

The first font and glyph specify the cursor source pixmap. The second font and glyph specify the cursor mask pixmap. The mask font defaults to the source font, and the mask glyph index defaults to the source glyph index.

If a relative or absolute file name is specified, that file is used to create the source pixmap. Then the string "Mask" is appended to locate the cursor mask pixmap. If the "Mask" file does not exist, the suffix "msk" is tried. If "msk" fails, no cursor mask will be used. If a relative file name is used, the directory specified by the resource name bitmapFilePath or class BitmapFilePath is added to the beginning of the file name. If the bitmapFilePath resource is not defined, the default directory on a UNIX-based system is /usr/include/X11/bitmaps.

2.4 Realizing a Widget

The XtRealizeWidget function performs two tasks:

- Creates an X window for the widget and, if it is a composite widget, for each of its managed children.
- Maps each window onto the screen.

```
void XtRealizeWidget(w)
      Widget w;
```

w Specifies the widget.

For further information about this function, see the *Guide to the XUI Toolkit Intrinsics*.

2.5 Standard Widget Manipulation Functions

After a widget has been created, a client can interact with that widget by calling either of the following:

- One of the standard widget manipulation routines that provide functions that all widgets support
- A widget class-specific manipulation routine

The X Toolkit provides generic routines to provide the application programmer access to a set of standard widget functions. These routines let an application or composite widget manipulate widgets without requiring explicit knowledge of the widget type. The standard widget manipulation functions let you:

- Control the location, size and mapping of widget windows
- Destroy a widget instance
- Obtain an argument value
- Set an argument value

2.5.1 Mapping Widgets

By default, widget windows automatically are mapped (made viewable) by XtRealizeWidget. This behavior can be changed by using XtSetMappedWhenManaged, and it then is the client's responsibility to use the XtMapWidget function to make the widget viewable.

```
void XtSetMappedWhenManaged(w, map_when_managed)
      Widget w;
      Boolean map_when_managed;
```

w Specifies the widget.

map_when_managed
Specifies the new value. If map_when_managed is True, the widget is mapped automatically when it is realized. If map_when_managed is False, the client must call XtMapWidget or make a second call to XtSetMappedWhenManaged to cause the child window to be mapped.

The definition for XtMapWidget is:

```
XtMapWidget(w)
     Widget w;
```

w Specifies the widget.

When you create several children in sequence for a common parent after it has been realized, it is generally more efficient to construct a list of children as they are created and use XtManageChildren to inform their parent of them all at once, instead of causing each child to be managed separately. By managing a list of children at one time, the parent can avoid wasteful duplication of geometry processing and the associated "screen flash".

```
void XtManageChildren(children, num_children)
      WidgetList children;
      Cardinal num_children;
```

children Specifies a list of children to add.

num_children Specifies the number of children to add.

If the parent is already visible on the screen, it is especially important to batch updates so that the minimum amount of visible window reconfiguration is performed.

For further information about these functions, see the *Guide to the XUI Toolkit Intrinsics.*

2.5.2 Destroying Widgets

To destroy a widget instance of any type, use XtDestroyWidget.

```
void XtDestroyWidget(w)
      Widget w;
```

w	Specifies the widget.

XtDestroyWidget destroys the widget and recursively destroys any children that it may have, including the windows created by its children. After calling XtDestroyWidget, no further references should be made to the widget or to the widget IDs of any children that the destroyed widget may have had.

2.5.3 Retrieving Widget Resource Values

To retrieve the current value of a resource attribute associated with a widget instance, use XtGetValues.

```
void XtGetValues(w, args, num_args)
      Widget w;
      ArgList args;
      Cardinal num_args;
```

w	Specifies the widget.
args	Specifies a variable-length argument list of name and address pairs that contain the resource name and the address into which the resource value is stored.
num_args	Specifies the number of arguments in the argument list.

The arguments and values passed in the argument list are dependent on the widget. Note that the caller is responsible for allocating space into which the returned resource value is copied; the ArgList contains a pointer to this storage. The caller must allocate storage of the type as represented in the widget. For example, x and y must be allocated as Position and so on. For further information, see the *Guide to the XUI Toolkit Intrinsics.*

2.5.4 Modifying Widget Resource Values

To modify the current value of a resource attribute associated with a widget instance, use XtSetValues.

```
void XtSetValues(w, args, num_args)
      Widget w;
      ArgList args;
      Cardinal num_args;
```

w	Specifies the widget.
args	Specifies a variable-length argument list of name and value pairs that contain the arguments to be modified and their new values.
num_args	Specifies the number of arguments in the argument list.

The arguments and values passed in the argument list depend on the widget being modified. Some widgets may not allow certain resources to be modified after the widget instance has been created or realized. No notification is given if any part of a XtSetValues request is ignored.

For further information about these functions, see the *Guide to the XUI Toolkit Intrinsics.*

Note

The argument list entry for XtGetValues specifies the address to which the caller wants the value copied. The argument list entry for XtSetValues, however, contains the new value itself if the size of value is less than sizeof(XtArgVal) (architecture dependent, but at least sizeof(long)); otherwise, it is a pointer to the value. String resources are always passed as pointers, regardless of the length of the string.

2.6 Using the Client Callback Interface

Widgets communicate changes in their state to their clients by means of a callback facility. The format for a client's callback handler is:

```
void CallbackProc(w, client_data, call_data)
      Widget w;
      caddr_t client_data;
      caddr_t call_data;
```

w	Specifies widget for which the callback is registered.
client_data	Specifies arbitrary client-supplied data that the widget should pass back to the client when the widget executes the client's callback procedure. This is a way for the client registering the callback to also register client-specific data: a pointer to additional information about the widget, a reason for invoking the callback, and so on. It is perfectly normal to have client_data of NULL if all necessary information is in the widget. This field is also frequently known as the *closure*.
call_data	Specifies any callback-specific data the widget wants to pass to the client. For example, when Scrollbar executes its

jumpProc callback list, it passes the current position of the thumb in the call_data argument.

Callbacks can be registered with widgets in one of two ways. When the widget is created, a pointer to a list of callback procedure and data pairs can be passed in the argument list to XtCreateWidget. The list is of type XtCallbackList:

```
typedef struct {
        XtCallbackProc callback;
        caddr_t closure;
} XtCallbackRec, *XtCallbackList;
```

The callback list must be allocated and initialized before calling XtCreateWidget. The end of the list is identified by an entry containing NULL in callback and closure. Once the widget is created, the client can change or de-allocate this list; the widget itself makes no further reference to it. The closure field contains the client_data passed to the callback when the callback list is executed.

The second method for registering callbacks is to use XtAddCallback after the widget has been created.

```
void XtAddCallback(w, callback_name, callback, client_data)
        Widget w;
        String callback_name;
        XtCallbackProc callback;
        caddr_t client_data;
```

w Specifies the widget to add the callback to.

callback_name
Specifies the callback list within the widget to append to.

callback Specifies the callback procedure to add.

client_data Specifies the data to be passed to the callback when it is invoked.

XtAddCallback adds the specified callback to the list for the named widget.

All widgets provide a callback list named XtNdestroyCallback where clients can register procedures that are to be executed when the widget is destroyed. The destroy callbacks are executed when the widget or an ancestor is destroyed. The call_data argument is unused for destroy callbacks.

The Intrinsics provide additional functions for further manipulating a callback list. For information about these functions, see XtCallCallbacks, XtRemoveCallback, XtRemoveCallbacks, and XtRemoveAllCallbacks in the *Guide to the XUI Toolkit Intrinsics.*

2.7 Programming Considerations

This section provides some guidelines to set up an application program that uses the X Toolkit. This section discusses:

- Writing applications
- Creating argument lists

2.7.1 Writing Applications

When writing an application that uses the toolkit, you should make sure that your application performs the following:

1. Include <X11/Intrinsic.h> in your application programs. This header file automatically includes <X11/Xlib.h>, so all Xlib functions also are defined.
2. Include the widget-specific header files for each widget type that you need to use. For example, <X11/Label.h> and <X11/Command.h>.
3. Call the XtInitialize function before invoking any other toolkit or Xlib functions. For further information, see Section 2.1 and the *Guide to the XUI Toolkit Intrinsics.*
4. To pass attributes to the widget creation routines that will over-ride any site or user customizations, set up argument lists. In this document, a list of valid argument names that start with XtN is provided in the discussion of each widget.

   For further information, see Section 2.7.2.
5. When the argument list is set up, create the widget by using the XtCreateWidget function. For further information, see Section 2.2 and the *Guide to the XUI Toolkit Intrinsics.*
6. If the widget has any callback routines, which are usually defined by the XtNcallback argument or the XtAddCallback function, declare these routines within the application.
7. After a widget has been created, use XtManageChild to manage it. If there is no manipulation of the widget between XtCreateWidget and XtManageChild, you can do this in a single step by using XtCreateManagedWidget. For further information about these functions, see the *Guide to the XUI Toolkit Intrinsics.*
8. After creating the initial widget hierarchy, windows must be created for each widget by calling XtRealizeWidget on the top level widget.
9. Most applications now sit in a loop processing events using XtMainLoop, for example:

   XtCreateManagedWidget(*name*, *class*, *parent*, *args*, *num_args*);
   XtRealizeWidget(*parent*);

```
XtMainLoop( );
```

For information about this function, see the *Guide to the XUI Toolkit Intrinsics*.

10. Link your application with libXaw.a (the Athena widgets), libXmu.a (miscellaneous utilities), libXt.a (the Intrinsics), and libX11.a (the core X library). The following provides a sample command line:

    cc -o *application application*.c –lXaw –lXmu –lXt –lX11

2.7.2 Creating Argument Lists

To set up an argument list for the inline specification of widget attributes, you can use one of the four approaches discussed in this section. You should use whichever approach fits the needs of the application and you are most comfortable with. In general, argument lists should be kept as short as possible to allow widget attributes to be specified through the resource database. Whenever a client inserts a specific attribute value in an argument list, the user is prevented from customizing the behavior of the widget. Resource names in the resource database, by convention, correspond to their symbolic names that are used in argument list without the XtN prefix. For example, the resource name for XtNforeground is "foreground". For further information, see the *Guide to the XUI Toolkit Intrinsics*.

The Arg structure contains:

```
typedef struct {
        String name;
        XtArgVal value;
} Arg, *ArgList;
```

The first approach lets you statically initialize the argument list. For example:

```
static Arg arglist[] = {
        {XtNwidth, (XtArgVal) 400},
        {XtNheight, (XtArgVal) 300},
};
```

This approach makes it easy to add or delete new elements. The XtNumber macro can be used to compute the number of elements in the argument list, thus preventing simple programming errors. The following provides an example:

```
        XtCreateWidget(name, class, parent, arglist, XtNumber(arglist));
```

The second approach lets you use the XtSetArg macro. For example:

```
Arg arglist[10];
XtSetArg(arglist[1], XtNwidth, 400);
XtSetArg(arglist[2], XtNheight, 300);
```

To make it easier to insert and delete entries, you also can use a variable index, as in this example:

```
Arg arglist[10];
Cardinal i=0;
XtSetArg(arglist[i], XtNwidth,  400);        i++;
XtSetArg(arglist[i], XtNheight, 300);        i++;
```

The i variable can then be used as the argument list count in the widget create function. In this example, XtNumber would return 10, not 2, and therefore is not useful.

Note

> You should not use auto-increment or auto-decrement within the first argument to XtSetArg. As it is currently implemented, XtSetArg is a macro that dereferences the first argument twice.

The third approach lets you individually set the elements of the argument list array, one piece at a time. For example:

```
Arg arglist[10];
arglist[0].name  = XtNwidth;
arglist[0].value = (XtArgVal) 400;
arglist[1].name  = XtNheight;
arglist[1].value = (XtArgVal) 300;
```

Note that in this example, as in the previous example, XtNumber would return 10, not 2, and therefore is not useful.

The fourth approach lets you use a mixture of the first and third approaches: you can statically define the argument list but modify some entries at runtime. For example:

```
static Arg arglist[] = {
        {XtNwidth, (XtArgVal) 400},
        {XtNheight, (XtArgVal) NULL},
};
arglist[1].value = (XtArgVal) 300;
```

In this example, XtNumber can be used, as in the first approach, for easier code maintenance.

2.7.3 Sample Program

The following program creates one command button that, when pressed, causes the program to exit. This example is a complete program that illustrates:

- Toolkit initialization
- Optional command-line arguments
- Widget creation
- Callback routines

```
#include <stdio.h>
#include <X11/Intrinsic.h>
#include <X11/Command.h>

static XrmOptionDescRec options[] = {
{"-label", "*button.label", XrmoptionSepArg, NULL}
};

Syntax(call)
    char *call;
{
    fprintf(stderr, "Usage: %s\n", call);
}

void Activate(w, client_data, call_data)
    Widget w;
    caddr_t client_data;  /* unused */
    caddr_t call_data;    /* unused */
{
    printf("button was activated.\n");
    exit(0);
}

void main(argc, argv)
    unsigned int argc;
    char **argv;
{
    Widget toplevel;
    static XtCallbackRec callbacks[] = {
        { Activate, NULL },
        { NULL, NULL },
    };

    static Arg args[] = {
      { XtNcallback, (XtArgVal)callbacks },
    };

    toplevel = XtInitialize("main", "Demo", options,
                                  XtNumber(options),
    &argc, argv );
    if (argc != 1) Syntax(argv[0]);
```

(continued on next page)

```
    XtCreateManagedWidget("button",commandWidgetClass,toplevel,
    args, XtNumber(args));

    XtRealizeWidget(toplevel);
    XtMainLoop();
}
```

Athena Widget Set 3

This chapter describes the following Athena widgets:

- Command
- Label
- Text
- Scrollbar
- Viewport
- Box
- VPaned
- Form
- Dialog
- List
- Grip

3.1 Command Widget

The Command widget is a rectangular button that contains a text or pixmap label. When the pointer cursor is on the button, the button border is highlighted to indicate that the button is available for selection. Then, when a pointer button is pressed and released the button is selected, and the application's callback routine is invoked.

The class variable for the Command widget is commandWidgetClass.

When creating a Command widget instance, the following resources are retrieved from the argument list or from the resource database:

Name	Type	Default
XtNbackground	Pixel	XtDefaultBackground
XtNbackgroundPixmap	Pixmap	None
XtNbitmap	Pixmap	None
XtNborderColor	Pixel	XtDefaultForeground
XtNborderPixmap	Pixmap	None

Name	Type	Default
XtNborderWidth	Dimension	1
XtNcallback	XtCallbackList	NULL
XtNcursor	Cursor	None
XtNdestroyCallback	XtCallbackList	NULL
XtNfont	XFontStruct*	XtDefaultFont
XtNforeground	Pixel	XtDefaultForeground
XtNheight	Dimension	Text height
XtNhighlightThickness	Dimension	2
XtNinsensitiveBorder	Pixmap	Gray
XtNinternalHeight	Dimension	2
XtNinternalWidth	Dimension	4
XtNjustify	XtJustify	XtJustifyCenter
XtNlabel	String	Button name
XtNmappedWhenManaged	Boolean	True
XtNresize	Boolean	True
XtNsensitive	Boolean	True
XtNtranslations	TranslationTable	see below
XtNwidth	Dimension	Text width
XtNx	Position	0
XtNy	Position	0

For an explanation of the common widget resources associated with the Command widget, see Section 2.3. The new resources associated with the Command widget are:

XtNbitmap	Specifies a bitmap to display in place of the text label. See the description of this resource in the Label widget for further details.
XtNfont	Specifies the label font.
XtNheight	Specifies the height of the Command widget. The default value is the minimum height that will contain: XtNinternalheight + height of XtNlabel + XtNinternalHeight If the specified height is larger than the minimum, the label string is centered vertically.
XtNhighlightThickness	Specifies the width of border that is to be highlighted.

XtNinsensitiveBorder	Specifies the border when it is not sensitive.
XtNinternalHeight	Represents the distance in pixels between the top and bottom of the label text or bitmap and the horizontal edges of the Command widget. HighlightThickness can be larger or smaller than this value.
XtNinternalWidth	Represents the distance in pixels between the ends of the label text or bitmap and the vertical edges of the Command widget. HighlightThickness can be larger or smaller than this value.
XtNjustify	Specifies left, center, or right alignment of the label string within the Command widget. If it is specified within an ArgList, one of the values XtJustifyLeft, XtJustifyCenter, or XtJustifyRight can be specified. In a resource of type "string", one of the values "left", "center", or "right" can be specified.
XtNlabel	Specifies the text string that is to be displayed in the Command widget if no bitmap is specified. The default is the widget name of the Command widget.
XtNresize	Specifies whether the Command widget should attempt to resize to its preferred dimensions whenever XtSetValues is called for it. The default is True.
XtNsensitive	If set to False, the Command widget will change its window border to XtNinsensitiveBorder and will stipple the label string.
XtNwidth	Specifies the width of the Command widget. The default value is the minimum width that will contain: XtNinternalWidth + width of XtNlabel + XtNinternalWidth If the width is larger or smaller than the minimum, XtNjustify determines how the label string is aligned.

The Command widget supports the following actions:

- Switching the button between the foreground and background colors with **set** and **unset**
- Processing application callbacks with **notify**
- Switching the internal border between highlighted and unhighlighted states with **highlight** and **unhighlight**

The following are the default translation bindings that are used by the Command widget:

<EnterWindow>:	highlight( )
<LeaveWindow>:	reset( )
<Btn1Down>:	set( )
<Btn1Up>:	notify( ) unset( )

With these bindings, the user can cancel the action before releasing the button by moving the pointer out of the Command widget.

The full list of actions supported by Command is:

highlight( )	Displays the internal highlight border in the XtNforeground color.
unhighlight( )	Displays the internal highlight border in the XtNbackground color.
set( )	Enters the "set" state, in which **notify** is possible and displays the interior of the button, including the highlight border, in the foreground color. The label is displayed in the background color.
unset( )	Cancels the "set" state and displays the interior of the button, including the highlight border, in the background color. The label is displayed in the foreground color.
reset( )	Cancels any **set** or **highlight** and displays the interior of the button in the background color, with the label displayed in the foreground color.
notify( )	Executes the XtNcallback callback list if executed in the **set** state. The value of the call_data argument is undefined.

To create a Command widget instance, use XtCreateWidget and specify the class variable commandWidgetClass.

To destroy a Command widget instance, use XtDestroyWidget and specify the widget ID of the button.

The Command widget supports two callback lists: XtNdestroyCallback and XtNcallback. The notify action executes the callbacks on the XtNcallback list. The call_data argument is unused.

3.2 Label Widget

A Label is an noneditable text string or pixmap that is displayed within a window. The string is limited to one line and can be aligned to the left, right, or center of its window. A Label can neither be selected nor directly edited by the user.

The class variable for the Label widget is labelWidgetClass.

When creating a Label widget instance, the following resources are retrieved from the argument list or from the resource database:

Name	**Type**	**Default**
XtNbackground	Pixel	XtDefaultBackground
XtNbackgroundPixmap	Pixmap	None
XtNbitmap	Pixmap	None
XtNborderColor	Pixel	XtDefaultForeground
XtNborderPixmap	Pixmap	None
XtNborderWidth	Dimension	1
XtNcursor	Cursor	None
XtNdestroyCallback	XtCallbackList	NULL
XtNfont	XFontStruct*	XtDefaultFont
XtNforeground	Pixel	XtDefaultForeground
XtNheight	Dimension	text height
XtNinsensitiveBorder	Pixmap	Gray
XtNinternalHeight	Dimension	2
XtNinternalWidth	Dimension	4
XtNjustify	XtJustify	XtJustifyCenter
XtNlabel	String	label name
XtNmappedWhenManaged	Boolean	True
XtNresize	Boolean	True
XtNsensitive	Boolean	True
XtNwidth	Dimension	text width
XtNx	Position	0
XtNy	Position	0

For an explanation of the common widget resources associated with the Label widget, see Section 2.3. The new resources associated with the Label widget are:

XtNbitmap	Specifies a bitmap to display in place of the text label. The bitmap can be specified as a string in the resource data base. The StringToPixmap converter will interpret the string as the name of a file in the bitmap utility format that is to be loaded into a pixmap.

	The string can be an absolute or a relative file name. If a relative file name is used, the directory specified by the resource name bitmapFilePath or th resource class BitmapFilePath is add to the beginnin of the specified file name. If the bitmapFilePath resource is not defined, the default directory on a UNIX-based system is /usr/include/X11/bitmaps.
XtNfont	Specifies the label font.
XtNheight	Specifies the height of the Label widget. The default value is the minimum height that will contain: XtNinternalheight + height of XtNlabel + XtNinternalHeight If the specified height is larger than the minimum, the label string is centered vertically.
XtNinsensitiveBorder	Specifies the border when the widget is not sensitive.
XtNinternalHeight	Represents the distance in pixels between the top and bottom of the label text or bitmap and the horizontal edges of the Label widget.
XtNinternalWidth	Represents the distance in pixels between the ends of the label text or bitmap and the vertical edges o the Label widget.
XtNjustify	Specifies left, center, or right alignment of the label string within the Label widget. If it is specified within an ArgList, one of the values XtJustifyLeft, XtJustifyCenter, or XtJustifyRight can be specified. I a resource of type "string", one of the values "left' "center", or "right" can be specified.
XtNlabel	Specifies the text string that is to be displayed in the button if no bitmap is specified. The default is the widget name of the Label widget.
XtNresize	Specifies whether the Label widget should attempt t resize to its preferred dimensions whenever XtSetValues is called for it.
XtNsensitive	If set to False, the Label widget will change its window border to XtNinsensitiveBorder and will stipp the label string.

XtNwidth	Specifies the width of the Label widget. The default value is the minimum width that will contain: XtNinternalWidth + width of XtNlabel + XtNinternalWidth If the width is larger or smaller than the minimum, XtNjustify determines how the label string is aligned.

To create a Label widget instance, use XtCreateWidget and specify the class variable labelWidgetClass.

To destroy a Label widget instance, use XtDestroyWidget and specify the widget ID of the label.

The Label widget supports only the XtNdestroyCallback callback list.

3.3 Text Widget

A Text widget is a window that provides a way for an application to display one or more lines of text. The displayed text can reside in a file on disk or in a string in memory. An option also lets an application display a vertical Scrollbar in the Text window, letting the user scroll through the displayed text. Other options allow an application to let the user modify the text in the window.

The Text widget is divided into three parts:

- Source
- Sink
- Text widget

The idea is to separate the storage of the text (source) from the painting of the text (sink). The Text widget coordinates the sources and sinks. Clients usually will use AsciiText widgets that automatically create the source and sink for the client. A client can, if it so chooses, explicitly create the source and sink before creating the Text widget.

The source stores and manipulates the text. The X Toolkit provides string and disk file sources. The source determines what editing functions may be performed on the text.

The sink obtains the fonts and the colors in which to paint the text. The sink also computes what text can fit on each line. The X Toolkit provides a single-font, single-color ASCII sink.

If a disk file is used to display the text, two edit modes are available:

- Append
- Read-only

Append mode lets the user enter text into the window, while read-only mode does not. Text may only be entered if the insertion point is after the last character in the window.

If a string in memory is used, the application must allocate the amount of space needed. If a string in memory is used to display text, three types of edit mode are available:

- Append-only
- Read-only
- Editable

The first two modes are the same as displaying text from a disk file. Editable mode lets the user place the cursor anywhere in the text and modify the text at that position. The text cursor position can be modified by using the key strokes or pointer buttons defined by the event bindings.

Many standard keyboard editing facilities are supported by the event bindings. The following actions are supported:

Cursor Movement
- forward-character
- backward-character
- forward-word
- backward-word
- forward-paragraph
- backward-paragraph
- beginning-of-line
- end-of-line
- next-line
- previous-line
- next-page
- previous-page
- beginning-of-file
- end-of-file
- scroll-one-line-up
- scroll-one-line-down

New Line
- newline-and-indent
- newline-and-backup
- newline

Kill
- kill-word
- backward-kill-word
- kill-selection
- kill-to-end-of-line
- kill-to-end-of-paragraph

Delete
- delete-next-character
- delete-previous-character
- delete-next-word
- delete-previous-word
- delete-selection

Selection
- select-word
- select-all
- select-start
- select-adjust
- select-end
- extend-start
- extend-adjust
- extend-end

Miscellaneous
- redraw-display
- insert-file
- do-nothing

Unkill
- unkill
- stuff
- insert-selection

Note

1. A page corresponds to the size of the Text window. For example, if the Text window is 50 lines in length, scrolling forward one page is the same as scrolling forward 50 lines.
2. The **delete** action deletes a text item. The **kill** action deletes a text item and puts the item in the kill buffer (X cut buffer 1).
3. The **unkill** action inserts the contents of the kill buffer into the text at the current position. The **stuff** action inserts the contents of the paste buffer (X cut buffer 0) into the text at the current position. The **insert-selection** action retrieves the value of a specified X selection or cut buffer, with fall-back to alternative selections or cut buffers.

The default event bindings for the Text widget are:

```
char defaultTextTranslations[] = "\
        Ctrl<Key>F:                  forward-character( ) \n\
        Ctrl<Key>B:                  backward-character( ) \n\
        Ctrl<Key>D:                  delete-next-character( ) \n\
        Ctrl<Key>A:                  beginning-of-line( ) \n\
        Ctrl<Key>E:                  end-of-line( ) \n\
        Ctrl<Key>H:                  delete-previous-character( ) \n\
        Ctrl<Key>J:                  newline-and-indent( ) \n\
        Ctrl<Key>K:                  kill-to-end-of-line( ) \n\
        Ctrl<Key>L:                  redraw-display( ) \n\
        Ctrl<Key>M:                  newline( ) \n\
        Ctrl<Key>N:                  next-line( ) \n\
        Ctrl<Key>O:                  newline-and-backup( ) \n\
        Ctrl<Key>P:                  previous-line( ) \n\
        Ctrl<Key>V:                  next-page( ) \n\
        Ctrl<Key>W:                  kill-selection( ) \n\
        Ctrl<Key>Y:                  unkill( ) \n\
        Ctrl<Key>Z:                  scroll-one-line-up( ) \n\
        Meta<Key>F:                  forward-word( ) \n\
        Meta<Key>B:                  backward-word( ) \n\
        Meta<Key>I:                  insert-file( ) \n\
        Meta<Key>K:                  kill-to-end-of-paragraph( ) \n\
        Meta<Key>V:                  previous-page( ) \n\
        Meta<Key>Y:                  stuff( ) \n\
        Meta<Key>Z:                  scroll-one-line-down( ) \n\
        :Meta<Key>d:                 delete-next-word( ) \n\
        :Meta<Key>D:                   kill-word( )  \n\
        :Meta<Key>h:                   delete-previous-word( )  \n\
        :Meta<Key>H:                   backward-kill-word( )  \n\
        :Meta<Key>\<:                  beginning-of-file( )  \n\
        :Meta<Key>\>:                  end-of-file( )  \n\
        :Meta<Key>]:                   forward-paragraph( )  \n\
        :Meta<Key>[:                   backward-paragraph( )  \n\
        ~Shift Meta<Key>Delete:        delete-previous-word( )  \n\
         Shift Meta<Key>Delete:        backward-kill-word( )  \n\
        ~Shift Meta<Key>Backspace:     delete-previous-word( )  \n\
         Shift Meta<Key>Backspace:     backward-kill-word( )  \n\
        <Key>Right:                    forward-character( )  \n\
        <Key>Left:                     backward-character( )  \n\
        <Key>Down:                     next-line( )  \n\
        <Key>Up:                       previous-line( )  \n\
        <Key>Delete:                   delete-previous-character( )  \n\
        <Key>BackSpace:                delete-previous-character( )  \n\
        <Key>Linefeed:                 newline-and-indent( )  \n\
        <Key>Return:                   newline( )  \n\
        <Key>:                         insert-char( )  \n\
        <FocusIn>:                     focus-in( )  \n\
        <FocusOut>:                    focus-out( )  \n\
        <Btn1Down>:                    select-start( )  \n\
        <Btn1Motion>:                  extend-adjust( )  \n\
        <Btn1Up>:                      extend-end(PRIMARY, CUT_BUFFER0)  \n\
        <Btn2Down>:                    insert-selection(PRIMARY, CUT_BUFFER0)  \n\
        <Btn3Down>:                    extend-start( )  \n\
        <Btn3Motion>:                  extend-adjust( )  \n\
        <Btn3Up>:                      extend-end(PRIMARY, CUT_BUFFER0)  \
```

A user-supplied resource entry can use application-specific bindings, a subset of the supplied default bindings, or both. The following is an example of a user-supplied resource entry that uses a subset of the default bindings:

```
Xmh*Text.Translations:  \
        <Key>Right:        forward-character( )  \n\
        <Key>Left:         backward-character( )  \n\
         Meta<Key>F:       forward-word( )  \n\
         Meta<Key>B:       backward-word( )  \n\
        :Meta<Key>]:       forward-paragraph( )  \n\
        :Meta<Key>[:       backward-paragraph( )  \n\
        <Key>:             insert-char( )
```

An augmented binding that is useful with the xclipboard utility is:

```
*Text.Translations: #override \
        Button1 <Btn2Down>:         extend-end(CLIPBOARD)
```

A Text widget lets both the user and the application take control of the text being displayed. The user takes control with the scroll bar or with key strokes defined by the event bindings. The scroll bar option places the scroll bar on the left side of the window and can be used with any editing mode. The application takes control with procedure calls to the Text widget to:

- Display text at a specified position
- Highlight specified text areas
- Replace specified text areas

The text that is selected within a Text window may be assigned to an X selection or copied into a cut buffer and can be retrieved by the application with the Intrinsics XtGetSelectionValue or the Xlib XFetchBytes functions respectively. Several standard selection schemes (e.g. character/word/paragraph with multi-click) are supported through the event bindings.

The class variable for the Text widget is textWidgetClass.

To create a Text string widget, use XtCreateWidget and specify the class variable asciiStringWidgetClass.

To create a Text file widget, use XtCreateWidget and specify the class variable asciiDiskWidgetClass.

Note

If you want to create an instance of the class textWidgetClass, you must provide a source and a sink when the widget is created. The Text widget cannot be instantiated without both.

When creating a Text widget instance, the following resources are retrieved from the argument list or from the resource database:

Name	Type	Default
XtNbackground	Pixel	XtDefaultBackground
XtNbackgroundPixmap	Pixmap	None
XtNborderColor	Pixel	XtDefaultForeground
XtNborderPixmap	Pixmap	None
XtNborderWidth	Dimension	4
XtNcursor	Cursor	XC_xterm
XtNdialogHOffset	int	10
XtNdialogVOffset	int	10

Name	Type	Default
XtNdestroyCallback	XtCallbackList	NULL
XtNdisplayPosition	int	0
XtNeditType	XtEditType	XttextRead
XtNfile	char*	tmpnam( )
XtNforeground	Pixel	Black
XtNfont	XFontStruct*	Fixed
XtNheight	Dimension	Font height
XtNinsertPosition	int	0
XtNleftMargin	Dimension	2
XtNlength	int	String length
XtNmappedWhenManaged	Boolean	True
XtNselectTypes	XtTextSelectType*	See below
XtNsensitive	Boolean	True
XtNstring	char*	Blank
XtNtextOptions	int	None
XtNtextSink	XtTextSink	None
XtNtextSource	XtTextSource	None
XtNtranslations	TranslationTable	See above
XtNwidth	Dimension	100
XtNx	Position	0
XtNy	Position	0

For an explanation of the common widget resources associated with the Text widget, see Section 2.3. The new resources associated with the Text widget are:

XtNdialogHOffset XtNdialogVOffset	Specified the horizontal and vertical offsets for the insert file dialog box.
XtNdisplayPosition	Specifies the character position of first line.
XtNediType	Specifies the edit mode (see Notes).
XtNfile	Specifies the file for asciiDiskClass.
XtNfont	Specifies the font name.
XtNinsertPosition	Specifies the character position of the caret.
XtNleftMargin	Specifies the left margin in pixels.
XtNlength	Specifies the size of the string buffer.
XtNselectTypes	Specifies the selection units for multiclicks.

XtNstring	Specifies the string for asciiStringWidgetClass.
XtNtextOptions XtNtextSink XtNtextSource	See Notes.

Note

1. You cannot use XtNeditType, XtNfile, XtNlength, and XtNfont with the XtTextSetValues and the XtTextGetValues calls.
2. The XtNeditType attribute has one of the values XttextAppend, XttextEdit, or XttextRead.
3. If asciiStringWidgetClass is used, the resource XtNstring specifies a buffer containing the text to be displayed and edited. AsciiStringWidget does not copy this buffer but uses it in-place.

The options for the XtNtextOptions attribute are:

editable	Specifies whether or not the user is allowed to modify the text.
resizeHeight	Makes a request to the parent widget to lengthen the window if all the text cannot fit in the window.
resizeWidth	Makes a request to the parent widget to widen the window if the text becomes too long to fit on one line.
scrollHorizontal	Puts a scroll bar on the top of the window.
scrollOnOverflow	Automatically scrolls the text up when new text is entered below the bottom (last) line.
scrollVertical	Puts a scroll bar on the left side of the window.
wordBreak	Starts a new line when a word does not fit on the current line.

These options can be ORed together to set more than one at the same time.

XtNselectionTypes is an array of entries of type XtTextSelectType and is used for multiclick. As the pointer button is clicked in rapid succession, each click highlights the next "type" described in the array.

XtselectAll	Selects the contents of the entire buffer.

XtselectChar	Selects text characters as the pointer moves over them.
XtselectLine	Selects the entire line.
XtselectNull	Indicates the end of the selection array.
XtselectParagraph	Selects the entire paragraph (delimited by newline characters).
XtselectPosition	Selects the current pointer position.
XtselectWord	Selects whole words (delimited by whitespace) as th pointer moves onto them.

The default selectType array is:

{XtselectPosition, XtselectWord, XtselectLine, XtselectParagraph, XtselectAll, XtselectNull}

For the default case, two rapid pointer clicks highlight the current word, three clicks highlight the current line, four clicks highlight the current paragraph, and five clicks highlight the entire text. If the timeout value is exceeded, the next pointer click returns to the first entry in the selection array. The selection array is not copied by the Text widget. The client must allocate space for the array and cannot deallocate or change it until the Text widget is destroyed or until a new selection array is set.

3.3.1 Selection Actions

The Text widget fully supports the X selection and cut buffer mechanisms. The following actions can be used to specify button bindings that will cause Text to assert ownership of one or more selections, to store the selected text into a cut buffer, and to retrieve the value of a selection or cut buffer and insert it into the text value.

insert-selection(*name*[,*name*,...])
Retrieves the value of the first (left-most) named selection that exists or the cut buffer that is not empty and inserts it into the input stream. The specified name can be that of any selection (for example, PRIMARY or SECONDARY) or a cut buffer (i.e. CUT_BUFFER0 through CUT_BUFFER7). Note that case matters.

select-start( ) Unselects any previously selected text and begins selecting new text.

select-adjust( )
extend-adjust( )
Continues selecting text from the previous start position.

start-extend( ) Begins extending the selection from the farthest (left or right) edge.

select-end(*name*[,*name*,...])
extend-end(*name*[,*name*,...])

Ends the text selection, asserts ownership of the specified selection(s) and stores the text in the specified cut buffer(s). The specified name can be that of a selection (for example, PRIMARY or SECONDARY) or a cut buffer (i.e. CUT_BUFFER0 through CUT_BUFFER7). Note that case is significant. If CUT_BUFFER0 is listed, the cut buffers are rotated before storing into buffer 0.

3.3.2 Selecting Text

To enable an application to select a piece of text, use XtTextSetSelection.

```
typedef long XtTextPosition;

void XtTextSetSelection(w, left, right)
     Widget w;
     XtTextPosition left, right;
```

w	Specifies the window ID.
left	Specifies the character position at which the selection begins.
right	Specifies the character position at which the selection ends.

If redisplay is not disabled, this function highlights the text and makes it the PRIMARY selection.

3.3.3 Unhighlighting Text

To unhighlight previously highlighted text in a window, use XtTextUnsetSelection.

```
void XtTextUnsetSelection(w)
     Widget w;
```

3.3.4 Getting Selected Text Character Positions

To enable the application to get the character positions of the selected text, use XtTextGetSelectionPos.

```
void XtTextGetSelectionPos(w, pos1, pos2)
     Widget w;
     XtTextPosition *pos1, *pos2;
```

w	Specifies the window ID.
pos1	Specifies a pointer to the location to which the beginning character position of the selection is returned.
pos2	Specifies a pointer to the location to which the ending character position of the selection is returned.

If the returned values are equal, there is no current selection.

3.3.5 Replacing Text

To enable an application to replace text, use XtTextReplace.

```
int XtTextReplace(w, start_pos, end_pos, text)
     Widget w;
     XtTextPosition start_pos, end_pos;
     XtTextBlock *text;
```

w	Specifies the window ID.
start_pos	Specifies the starting character position of the text replacement.
end_pos	Specifies the ending character position of the text replacement.
text	Specifies the text to be inserted into the file.

The XtTextReplace function deletes text in the specified range (startPos, endPos) and inserts the new text at startPos. The return value is XawEditDone if the replacement is successful, XawPositionError if the edit mode is XttextAppend and startPos is not the last character of the source, or XawEditError if either the source was read-only or the range to be deleted is larger than the length of the source.

The XtTextBlock structure defined in <X11/Text.h> contains:

```
typedef struct {
        int firstPos;
        int length;
        char *ptr;
        Atom format;
} XtTextBlock, *TextBlockPtr;
```

The firstPos field is the starting point to use within the ptr field. The value is usually zero. The length field is the number of characters that are transferred from the ptr field. The number of characters transferred is usually the number of characters in ptr. The format field is not currently used, but should be specified as FMT8BIT. The XtTextReplace arguments **start_pos** and **end_pos** represent the text source character positions for the existing text that is to be replaced by the text in the XtTextBlock structure. The characters from start_pos up to but not including end_pos are deleted, and the characters that are specified by the text block are inserted in their place. If start_pos and end_pos are equal, no text is deleted and the new text is inserted after start_pos.

Note

Only ASCII text is currently supported, and only one font can be used for each Text widget.

3.3.6 Redisplaying Text

To redisplay a range of characters, use XtTextInvalidate.

```
void XtTextInvalidate(w, from, to)
      Widget w;
      XtTextPosition from, to;
```

The XtTextInvalidate function causes the specified range of characters to be redisplayed immediately if redisplay is enabled or the next time that redisplay is enabled.

To enable redisplay, use XtTextEnableRedisplay.

```
void XtTextEnableRedisplay(w)
      Widget w;
```

The XtTextEnableRedisplay function flushes any changes due to batched updates when XtTextDisableRedisplay was called and allows future changes to be reflected immediately.

To disable redisplay while making several changes, use XtTextDisableRedisplay.

```
void XtTextDisableRedisplay(w)
       Widget w;
```

The XtTextDisableRedisplay function causes all changes to be batched until XtTextDisplay or XtTextEnableRedisplay is called.

To display batched updates, use XtTextDisplay.

```
void XtTextDisplay(w)
     Widget w;
```

The **XtTextDisplay** function forces any accumulated updates to be displayed.

To notify the source that the length has been changed, use **XtTextSetLastPos**.

```
void XtTextSetLastPos(w, last);
      Widget w;
      XtTextPosition last;
```

The **XtTextSetLastPos** function notifies the text source that data has been added to or removed from the end of the source.

3.3.7 Changing Resources

The following procedures are convenience procedures that replace calls to **XtSetValues** or **XtGetValues** when only a single resource is to be modified or retrieved.

To assigns a new value to XtNtextOptions resource, use **XtTextChangeOptions**.

```
void XtTextChangeOptions(w, options)
      Widget w;
      int options;
```

To obtain the current value of XtNtextOptions for the specified widget, use **XtTextGetOptions**.

```
int XtTextGetOptions(w)
      Widget w;
```

To obtain the character position of the left-most character on the first line displayed in the widget (that is, the value of XtNdisplayPosition), use **XtTextTopPosition**.

```
XtTextPosition XtTextTopPosition(w)
      Widget w;
```

To move the insertion caret to the specified source position, use **XtTextSetInsertionPoint**.

```
void XtTextSetInsertionPoint(w, position)
      Widget w;
      XtTextPosition position;
```

The text will be scrolled vertically if necessary to make the line containing the insertion point visible. The result is equivalent to setting the XtNinsertPosition resource.

To obtain the current position of the insertion caret, use **XtTextGetInsertionPoint**.

```
XtTextPosition XtTextGetInsertionPoint(w)
      Widget w;
```

The result is equivalent to retrieving the value of the XtNinsertPosition resource.

To replace the text source in the specified widget, use **XtTextSetSource**.

```
void XtTextSetSource(w, source, position)
      Widget w;
      XtTextSource source;
      XtTextPosition position;
```

A display update will be performed if redisplay has not been disabled.

To obtain the current text source for the specified widget, use **XtTextGetSource**.

```
XtTextSource XtTextGetSource(w)
      Widget w;
```

3.3.8 Creating Sources and Sinks

The following functions for creating and destroying text sources and sinks are called automatically by **AsciiStringWidget** and **AsciiDiskWidget** and it is therefore only necessary for the client to use them when creating an instance of **textWidgetClass**.

To create a new ASCII text sink, use **XtAsciiSinkCreate**.

```
XtTextSink XtAsciiSinkCreate(w, args, num_args)
      Widget w;
      ArgList args;
      Cardinal num_args;
```

The resources required by the sink are qualified by the name and class of the parent and the sub-part name XtNtextSink and class XtCTextSink.

To deallocate an ASCII text sink, use XtAsciiSinkDestroy.

```
void XtAsciiSinkDestroy(sink)
      XtTextSink sink;
```

The sink must not be in use by any widget or an error will result.

To create a new text disk source, use XtDiskSourceCreate.

```
XtTextSource XtDiskSourceCreate(w, args, num_args)
      Widget w;
      ArgList args;
      Cardinal num_args;
```

The resources required by the source are qualified by the name and class of the parent and the sub-part name XtNtextSource and class XtCTextSource.

To deallocate a text disk source, use XtDiskSourceDestroy.

```
void XtDiskSourceDestroy(source)
      XtTextSource source;
```

The source must not be in use by any widget or an error will result.

To create a new text string source, use XtStringSourceCreate.

```
XtTextSource XtStringSourceCreate(w, args, num_args)
      Widget w;
      ArgList args;
      Cardinal num_args;
```

The resources required by the source are qualified by the name and class of the parent and the sub-part name XtNtextSource and class XtCTextSource.

To deallocate a text string source, use XtStringSourceDestroy.

```
void XtStringSourceDestroy(source)
      XtTextSource source;
```

The source must not be in use by any widget or an error will result.

3.4 Scrollbar Widget

The Scrollbar widget is a rectangular area that contains a slide region and a thumb (slide bar). A Scrollbar can be used alone, as a valuator, or it can be used within a composite widget (for example, a Viewport). A Scrollbar can be aligned either vertically or horizontally.

When a Scrollbar is created, it is drawn with the thumb in a contrasting color. The thumb is normally used to scroll client data and to give visual feedback on the percentage of the client data that is visible.

Each pointer button invokes a specific scroll bar action. That is, given either a vertical or horizontal alignment, the pointer button actions will scroll or return data as appropriate for that alignment. Pointer buttons 1 and 3 do not perform scrolling operations by default. Instead, they return the pixel position of the cursor on the scroll region. When pointer button 2 is clicked, the thumb moves to the current pointer position. When pointer button 2 is held down and the pointer pointer is moved, the thumb follows the pointer.

The cursor in the scroll region changes depending on the current action. When no pointer button is pressed, the cursor appears as an arrow that points in the direction that scrolling can occur. When pointer button 1 or 3 is pressed, the cursor appears as a single-headed arrow that points in the logical direction that the client will move the data. When pointer button 2 is pressed, the cursor appears as an arrow that points to the thumb.

While scrolling is in progress, the application receives notification from callback procedures. For both scrolling actions, the callback returns the Scrollbar widget ID, the client_data, and the pixel position of the pointer when the button was released. For smooth scrolling, the callback routine returns the scroll bar window, the client data, and the current relative position of the thumb. When the thumb is moved using pointer button 2, the callback procedure is invoked continuously. When either button 1 or 3 is pressed, the callback procedure is invoked only when the button is released and the client callback procedure is responsible for moving the thumb.

The class variable for the Scrollbar widget is scrollbarWidgetClass.

When creating a Scrollbar widget instance, the following resources are retrieved from the argument list or from the resource database:

Name	Type	Default
XtNbackground	Pixel	white
XtNbackgroundPixmap	Pixmap	None
XtNborderColor	Pixel	XtDefaultForeground
XtNborderPixmap	Pixmap	None
XtNborderWidth	Dimension	1
XtNdestroyCallback	XtCallbackList	NULL
XtNforeground	Pixel	black
XtNheight	Dimension	See below
XtNjumpProc	XtCallbackList	NULL
XtNlength	Dimension	None
XtNmappedWhenManaged	Boolean	True
XtNorientation	XtOrientation	XtorientVertical
XtNscrollDCursor	Cursor	XC_sb_down_arrow
XtNscrollHCursor	Cursor	XC_sb_h_double_arrow
XtNscrollLCursor	Cursor	XC_sb_left_arrow
XtNscrollProc	XtCallbackList	NULL
XtNscrollRCursor	Cursor	XC_sb_right_arrow
XtNscrollUCursor	Cursor	XC_sb_up_arrow
XtNscrollVCursor	Cursor	XC_sb_v_double_arrow
XtNsensitive	Boolean	True
XtNshown	float	NULL
XtNthickness	Dimension	14
XtNthumb	Pixmap	Grey
XtNtop	float	NULL
XtNtranslations	TranslationTable	See below
XtNwidth	Dimension	See below
XtNx	Position	NULL
XtNy	Position	NULL

For an explanation of the common widget resources associated with the Scrollbar widget, see Section 2.3. The new resources associated with the Scrollbar widget are:

XtNjumpProc	Specifies the callback procedure for thumb selection.
XtNlength	Specifies the major dimension, which is the height c XtorientVertical.
XtNorientation	Specifies the vertical or horizontal orientation of the widget.

XtNscrollDCursor	Specifies the cursor that is to be used when scrolling down.
XtNscrollHCursor	Specifies the idle horizontal cursor.
XtNscrollLCursor	Specifies the cursor that is to be used when scrolling left.
XtscrollProc	Specifies the callback procedure for the slide region.
XtNscrollRCursor	Specifies the cursor that is to be used when scrolling right.
XtNscrollUCursor	pecifies the cursor that is to be used when scrolling up.
XtNscrollVCursor	Specifies the idle vertical cursor.
XtNshown	Specifies the percentage that the thumb covers.
XtNthickness	Specifies the minor dimension, whcih is the height of XtorientHorizontal.
XtNthumb	Specifies the pixmap that is to be used for the thumb.
XtNtop	Specifies the position on the scroll bar.

Note that the class for all cursor resources is XtCCursor.

You can set the dimensions of the Scrollbar two ways. As for all widgets, you can use the XtNwidth and XtNheight resources. In addition, you can use an alternative method that is independent of the vertical or horizontal orientation:

XtNlength	Specifies the height for a vertical Scrollbar and the width for a horizontal Scrollbar.
XtNthickness	Specifies the width for a vertical Scrollbar and the height for a horizontal Scrollbar.

To create a Scrollbar widget instance, use XtCreateWidget and specify the class variable scrollbarWidgetClass.

To destroy a Scrollbar widget instance, use XtDestroyWidget and specify the widget ID for the Scrollbar.

The arguments to the XtNscrollProc callback procedure are:

```
void ScrollProc(scrollbar, client_data, position)
     Widget scrollbar;
     caddr_t client_data;
     caddr_t position;      /* int */
```

scrollbar	Specifies the ID of the Scrollbar.
client_data	Specifies the client data.
position	Returns the pixel position of the thumb in integer form.

The XtNscrollProc callback is used for incremental scrolling and is called by the **NotifyScroll** action. The position argument is a signed quantity and should be cast to an int when used. Using the default button bindings, button 1 returns a positive value, and button 3 returns a negative value. In both cases, the magnitude of the value is the distance of the pointer in pixels from the top (or left) of the Scrollbar. The value will never be less than zero or greater than the length of the Scrollbar.

The arguments to the XtNjumpProc callback procedure are:

```
void JumpProc(scrollbar, client_data, percent)
     Widget scrollbar;
     caddr_t client_data;
     caddr_t percent_ptr;     /* float* */
```

scrollbar	Specifies the ID of the scroll bar window.
client_data	Specifies the client data.
percent_ptr	Specifies the floating point position of the thumb (0.0 – 1.0).

The XtNjumpProc callback is used to implement smooth scrolling and is called by the **NotifyThumb** action. Percent_ptr must be cast to a pointer to float before use; i.e.

```
float percent = *(float*)percent_ptr;
```

With the default button bindings, button 2 moves the thumb interactively, and the XtNjumpProc is called on each new position of the pointer.

Note

An older interface used XtNthumbProc and passed the percentage by value rather than by reference. This interface is not portable across machine architectures and therefore is no longer supported but is still implemented for those (non-portable) applications which used it.

To set the position and length of a Scrollbar thumb, use XtScrollbarSetThumb.

```
void XtScrollbarSetThumb(w, top, shown)
     Widget w;
     float top;
     float shown;
```

w	Specifies the Scrollbar widget ID.
top	Specifies the position of the top of the thumb as a fraction of the length of the Scrollbar.
shown	Specifies the length of the thumb as a fraction of the total length of the Scrollbar.

XtScrollbarThumb moves the visible thumb to position (0.0 – 1.0) and length (0.0 – 1.0). Either the top or shown arguments can be specified as −1.0, in which case the current value is left unchanged. Values greater than 1.0 are truncated to 1.0.

If called from XtNjumpProc, XtScrollbarSetThumb has no effect.

The actions supported by the Scrollbar widget are:

StartScroll(*value***)**
The possible values are Forward, Backward, or Continuous. This must be the first action to begin a new movement.

NotifyScroll(*value***)**
The possible values are Proportional or FullLength. If the argument to StartScroll was Forward or Backward, NotifyScroll executes the XtNscrollProc callbacks and passes either the position of the pointer if its argument is Proportional or the full length of the scroll bar if its argument is FullLength. If the argument to StartScroll was Continuous, NotifyScroll returns without executing any callbacks.

EndScroll( ) This must be the last action after a movement is complete.

MoveThumb( ) Repositions the scroll bar thumb to the current pointer location.

NotifyThumb( )
Calls the XtNjumpProc callbacks and passes the relative position of the pointer as a percentage of the scroll bar length.

The default bindings for Scrollbar are:

```
<Btn1Down>:        StartScroll(Forward)
<Btn2Down>:        StartScroll(Continuous) MoveThumb( ) NotifyThumb( )
<Btn3Down>:        StartScroll(Backward)
<Btn2Motion>:      MoveThumb( ) NotifyThumb( )
<BtnUp>:           NotifyScroll(Proportional) EndScroll( )
```

Examples of additional bindings a user might wish to specify in a resource file are:

```
*Scrollbar.Translations: \
        ˜Meta<KeyPress>space:     StartScroll(Forward) NotifyScroll(FullLength) \n\
         Meta<KeyPress>space:     StartScroll(Backward) NotifyScroll(FullLength) \n\
                                  EndScroll( )
```

3.5 Viewport Widget

The Viewport widget consists of a frame window, one or two Scrollbars, and an inner window. The frame window is determined by the viewing size of the data that is to be displayed and the dimensions to which the Viewport is created. The inner window is the full size of the data that is to be displayed and is clipped by the frame window. The Viewport widget controls the scrolling of the data directly. No application callbacks are required for scrolling.

When the geometry of the frame window is equal in size to the inner window, or when the data does not require scrolling, the Viewport widget automatically removes any scroll bars. The forceBars option causes the Viewport widget to display any scroll bar permanently.

The class variable for the Viewport widget is viewportWidgetClass.

When creating a Viewport widget instance, the following resources are retrieved from the argument list or from the resource database:

Name	Type	Default
XtNallowHoriz	Boolean	False
XtNallowVert	Boolean	False
XtNbackground	Pixel	XtDefaultBackground
XtNbackgroundPixmap	Pixmap	None
XtNborderColor	Pixel	XtDefaultForeground
XtNborderPixmap	Pixmap	None
XtNborderWidth	Dimension	1
XtNdestroyCallback	XtCallbackList	NULL
XtNforceBars	Boolean	False
XtNheight	Dimension	height of child
XtNmappedWhenManaged	Boolean	True

Name	Type	Default
XtNsensitive	Boolean	True
XtNtranslations	TranslationTable	None
XtNuseBottom	Boolean	False
XtNuseRight	Boolean	False
XtNwidth	Dimension	width of child
XtNx	Position	0
XtNy	Position	0

For an explanation of the common widget resources associated with the Viewport widget, see Section 2.3 The new resources associated with the Viewport widget are:

XtNallowHoriz	Specifies whether horizontal scroll bars are to be allowed.
XtNallowVert	Specifies whether vertical scroll bars are to be allowed.
XtNforceBars	Specifies whether to force the display of scroll bars.
XtNuseBottom	Specifies whether to use top or bottom bars.
XtNuseRight	Specifies whether to use right or left bars.

The Viewport widget manages a single child widget. When the size of the child is larger than the size of the Viewport, the user can interactively move the child within the Viewport by repositioning the Scrollbars.

The default size of the Viewport before it is realized is the width and/or height of the child. After it is realized, the viewport will allow its child to grow vertically or horizontally if XtNallowVert or XtNallowHoriz were set, respectively. If the corresponding vertical or horizontal scrolling were not enabled, the viewport will propagate the geometry request to its own parent and the child will be allowed to change size only if the (grand) parent allows it. Regardless of whether or not scrolling was enabled in the corresponding direction, if the child requests a new size smaller than the viewport size, the change will be allowed only if the parent of the viewport allows the viewport to shrink to the appropriate dimension.

To create a Viewport widget instance, use XtCreateWidget and specify the class variable viewportWidgetClass.

To insert a child into a Viewport widget, use XtCreateWidget and specify the widget ID of the previously created Viewport as the parent.

To remove a child from a Viewport widget, use XtUnmanageChild or XtDestroyWidget and specify the widget ID of the child.

To delete the inner window, any children, and the frame window, use XtDestroyWidget and specify the widget ID of the Viewport widget.

3.6 Box Widget

The Box widget provides geometry management of arbitrary widgets in a box of a specified dimension. The children are rearranged when resizing events occur either on the Box or when children are added or deleted. The Box widget always attempts to pack its children as closely as possible within the geometry allowed by its parent.

Box widgets are commonly used to manage a related set of Command widgets and are frequently called ButtonBox widgets, but the children are not limited to buttons.

The children are arranged on a background that has its own specified dimensions and color.

The class variable for the Box widget is boxWidgetClass.

When creating a Box widget instance, the following resources are retrieved from the argument list or from the resource database:

Name	**Type**	**Default**
XtNbackground	Pixel	XtDefaultBackground
XtNbackgroundPixmap	Pixmap	None
XtNborderColor	Pixel	XtDefaultForeground
XtNborderPixmap	Pixmap	None
XtNborderWidth	Dimension	1
XtNdestroyCallback	XtCallbackList	NULL
XtNhSpace	Dimension	4
XtNheight	Dimension	see below
XtNmappedWhenManaged	Boolean	True
XtNtranslations	TranslationTable	None
XtNvSpace	Dimension	4
XtNwidth	Dimension	width of widest child
XtNx	Position	0
XtNy	Position	0

For an explanation of the common widget resources associated with the Box widget, see Section 2.3. The new resources associated with the Box widget are:

XtNhSpace	Specifies the left and right distance between children.
XtNvSpace	Specifies the top and bottom distance between children.

The Box widget positions its children in rows with XtNhSpace pixels to the left and right of each child and XtNvSpace pixels between rows. If the Box width is not specified, the Box widget uses the width of the widest child. Each time a child is managed or unmanaged, the Box widget will attempt to reposition the remaining children to compact the box. Children are positioned in order left to right, top to bottom. When the next child does not fit on the current row, a new row is started. If a child is wider than the width of the box, the box will request a larger width from it parent and will begin the layout process from the beginning if a new width is granted. After positioning all children, the Box widget attempts to shrink its own size to the minimum dimensions required for the layout.

To create a box widget instance, use XtCreateWidget and specify the class variable boxWidgetClass.

To add a child to the Box, use XtCreateWidget and specify the widget ID of the Box as the parent of the new widget.

To remove a child from the Box, use XtUnmanageChild or XtDestroyWidget and specify the widget ID of the child.

To destroy a Box widget instance, use XtDestroyWidget and specify the widget ID of the Box widget. All the children of this box are automatically destroyed at the same time.

3.7 VPaned Widget

The VPaned widget manages children in a vertically tiled fashion. A region, called a grip, appears on the border between each child. When the pointer is positioned on a grip and pressed, an arrow is displayed that indicates the significant pane that is being resized. While keeping the pointer button down, the user can move the pointer up or down. This, in turn, changes the window borders, causing one pane to shrink and some other pane to grow. The cursor indicates the pane that is of interest to the user; some other pane in the opposite direction will be chosen to grow or shrink an equal amount. The choice of alternate pane is a function of the XtNmin, XtNmax and XtNskipAdjust constraints on the other panes. With the default bindings, button 1 resizes the pane above the selected grip, button 3 resizes the pane below the selected grip and button 2 repositions the border between two panes only.

The class variable for the VPaned widget is vPanedWidgetClass.

When creating a VPaned widget instance, the following resources are retrieved from the argument list or from the resource database:

Name	**Type**	**Default**
XtNbackground	Pixel	XtDefaultBackground
XtNbackgroundPixmap	Pixmap	None
XtNbetweenCursor	Cursor	XC_sb_left_arrow
XtNborderColor	Pixel	XtDefaultForeground
XtNborderPixmap	Pixmap	None
XtNborderWidth	Dimension	1
XtNdestroyCallback	XtCallbackList	NULL
XtNforeground	Pixel	Black
XtNgripCursor	Cursor	XC_sb_v_double_arrow
XtNgripIndent	Position	10
XtNgripTranslations	TranslationTable	internal
XtNheight	Dimension	sum of child heights
XtNlowerCursor	Cursor	XC_sb_down_arrow
XtNmappedWhenManaged	Boolean	True
XtNrefigureMode	Boolean	On
XtNsensitive	Boolean	True
XtNtranslations	TranslationTable	None
XtNupperCursor	Cursor	XC_sb_up_arrow
XtNwidth	Dimension	width of widest child
XtNx	Position	0
XtNy	Position	0

For an explanation of the common widget resources associated with the VPaned widget, see Section 2.3. The new resources associated with the VPaned widget are:

XtNbetweenCursor	Specifies the cursor that is to be used for changing the boundary.
XtNgripCursor	Specifies the cursor that is to be used when the gri is not active.
XtNgripIndent	Specifies the offset of the grip, in pixels, from the margin.
XtNgripTranslations	Specifies the button bindings for the grip.
XtNlowerCursor	Specifies the cursor that is to be used when resizin a pane below the grip.

XtNrefigureMode	Specifies whether children should be adjusted.
XtNupperCursor	Specifies the cursor that is to be used when resizing a pane above the grip.

To create a VPaned widget instance, use XtCreateWidget and specify the class variable vPanedWidgetClass.

Once the parent frame is created, you then add panes to it. Any type of widget can be paned.

To add a child pane to a VPaned frame, use XtCreateWidget and specify the widget ID of the VPaned widget as the parent of each new child pane.

During the creation of a child pane, the following resources, by which the VPaned widget controls the placement of the child, can be specified in the argument list or retrieved from the resource database:

Name	Type	Default
XtNallowResize	Boolean	False
XtNmax	Dimension	unlimited
XtNmin	Dimension	1
XtNskipAdjust	Boolean	False

XtNallowResize	If False, specifies to ignore child resize requests.
XtNmax	Specifies the maximum height for a pane.
XtNmin	Specifies the minimum height for a pane.
XtNskipAdjust	If True, the VPaned widget should not automatically resize the pane.

To delete a pane from a vertically paned window frame, use XtUnmanageWidget or XtDestroyWidget and specify the widget ID of the child pane.

To enable or disable a child's request for pane resizing, use XtPanedAllowResize.

```
void XtPanedAllowResize(w, allow_resize)
     Widget w;
     Boolean allow_resize;
```

w	Specifies the widget ID of the child widget pane.
allow_resize	Enables or disables a pane window for resizing requests.

If allow_resize is True, VPane allows geometry requests from the child to change the pane's height. If allow_resize is False, VPane ignores geometry requests from the child to change the pane's height. The default state is True before the VPane is realized and False after it is realized. This procedure is equivalent to changing the XtNallowResize resource for the child.

To change the minimum and maximum height settings for a pane, use XtPanedSetMinMax.

```
void XtPanedSetMinMax(w, min, max)
     Widget w;
     int min, max;
```

w	Specifies the widget ID of the child widget pane.
min	New minimum height of the child, expressed in pixels.
max	New maximum height of the child, expressed in pixels.

This procedure is equivalent to setting the XtNmin and XtNmax resources for the child.

To enable or disable automatic recalculation of pane sizes and positions, use XtPanedSetRefigureMode.

```
void XtPanedSetRefigureMode(w, mode)
     Widget w;
     Boolean mode;
```

w	Specifies the widget ID of the VPaned widget.
mode	Enables or disables refiguration.

You should set the mode to FALSE if you add multiple panes to or remove multiple panes from the parent frame after it has been realized, unless you can arrange to manage all the panes at once using XtManageChildren. After all the panes are added, set the mode to TRUE. This avoids unnecessary geometry calculations and "window dancing".

To delete an entire VPaned widget and all associated data structures, use XtDestroyWidget and specify the widget ID of the VPaned widget. All the children of the VPaned widget are automatically destroyed at the same time.

3.8 Form Widget

The Form widget can contain an arbitrary number of children or subwidgets. The Form provides geometry management for its children, which allows individual control of the position of each child. Any combination of children can be added to a Form. The initial positions of the children may be computed relative to the positions of other children. When the Form is resized, it computes new positions and sizes for its children. This computation is based upon information provided when a child is added to the Form.

The class variable for a Form widget is formWidgetClass.

When creating a Form widget instance, the following resources are retrieved from the argument list or from the resource database:

Name	Type	Default
XtNbackground	Pixel	XtDefaultBackground
XtNbackgroundPixmap	Pixmap	None
XtNborderColor	Pixel	XtDefaultForeground
XtNborderPixmap	Pixmap	None
XtNborderWidth	Dimension	1
XtNdefaultDistance	int	4
XtNdestroyCallback	XtCallbackList	NULL
XtNheight	Dimension	computed at realize
XtNmappedWhenManaged	Boolean	True
XtNsensitive	Boolean	True
XtNtranslations	TranslationTable	None
XtNwidth	Dimension	computed at realize
XtNx	Position	NULL
XtNy	Position	NULL

For an explanation of the common widget resources associated with the Form widget, see Section 2.3. The new resources associated with the Form widget are:

XtNdefaultDistance	Specifies the default distance for XtNhorizDistance and XtNvertDistance.

To create a Form widget instance, use XtCreateWidget and specify the class variable formWidgetClass.

To add a new child to a Form, use XtCreateWidget and specify the widget ID of the previously created Form as the parent of the child.

When creating children that are to be added to a Form, the following additional resources are retrieved from the argument list or from the resource database:

Name	Type	Default
XtNbottom	XtEdgeType	XtRubber
XtNfromHoriz	Widget	NULL
XtNfromVert	Widget	NULL
XtNhorizDistance	int	XtdefaultDistance
XtNleft	XtEdgeType	XtRubber
XtNresizable	Boolean	FALSE
XtNright	XtEdgeType	XtRubber
XtNtop	XtEdgeType	XtRubber
XtNvertDistance	int	XtdefaultDistance

If XtNresizable is True, the child is allowed to be resized.

When a widget is added to a Form, constraints can be specified to the Form to indicate where the child should be positioned within the Form.

The resources XtNhorizDistance and XtNfromHoriz let the widget position itself a specified number of pixels horizontally away from another widget in the form. As an example, XtNhorizDistance could equal 10 and XtNfromHoriz could be the widget ID of another widget in the Form. The new widget will be placed 10 pixels to the right of the widget defined in XtNfromHoriz. If XtNfromHoriz equals NULL, then XtNhorizDistance is measured from the left edge of the Form.

Similarly, the resources XtNvertDistance and XtNfromVert let the widget position itself a specified number of pixels vertically away from another widget in the Form. If XtNfromVert equals NULL, then XtNvertDistance is measured from the top of the Form. Form provides a StringToWidget conversion procedure. Using this procedure, the resource database may be used to specify the XtNfromHoriz and XtNfromVert resources by widget name rather than widget id. The string value must be the name of a child of the same Form widget parent.

The XtNtop, XtNbottom, XtNleft, and XtNright resources tell the Form where to position the child when the Form is resized. XtEdgeType is defined in <X11/Form.h> and is one of XtChainTop, XtChainBottom, XtChainLeft, XtChainRight or XtRubber.

The values XtChainTop, XtChainBottom, XtChainLeft, and XtChainRight specify that a constant distance from an edge of the child to the top, bottom, left, and right edges respectively of the Form is to be maintained. The value XtRubber specifies that a proportional distance from the edge of the child to the left or top edge of the Form is to be maintained when the form is resized. The proportion is determined from the initial position of the child and the initial size of the Form. Form provides a StringToEdgeType conversion procedure to allow the resize constraints to be easily specified in a resource file.

The default width of the Form is the minimum width needed to enclose the children after computing their initial layout, with a margin of XtNdefaultDistance at the right and bottom edges. If a width and height is assigned to the Form that is too small for the layout, the children will be clipped by the right and bottom edges of the Form.

To remove a child from a Form, use XtUnmanageChild or XtDestroyWidget and specify the widget ID of the child widget.

To destroy a Form widget instance, use XtDestroyWidget and specify the widget ID of the Form. All children of the Form are automatically destroyed at the same time.

When a new child becomes managed or an old child unmanaged, Form will recalculate the positions of its children according to the values of the XtNhorizDistance, XtNfromHoriz, XtNvertDistance and XtNfromVert constraints at the time the change is made. No re-layout is performed when a child makes a geometry request.

To force or defer a re-layout of the Form, use XtFormDoLayout.

```
void XtFormDoLayout(w, do_layout)
      Widget w;
      Boolean do_layout;
```

w	Specifies the Form widget.
do_layout	Enables (if True) or disables (if False) layout of the Form widget.

When making several changes to the children of a Form widget after the Form has been realized, it is a good idea to disable re-layout until all changes have been made, then allow the layout. Form increments an internal count each time XtFormDoLayout is called with do_layout False and decrements the count when do_layout is True. When the count reaches 0, Form performs a re-layout.

3.9 Dialog Widget

The Dialog widget implements a commonly used interaction semantic to prompt for auxiliary input from a user. For example, you can use a Dialog widget when an application requires a small piece of information, such as a file name, from the user. A Dialog widget is simply a special case of the Form widget that provides a convenient way to create a "preconfigured form".

The typical Dialog widget contains three areas. The first line contains a description of the function of the Dialog widget, for example, the string "Filename:". The second line contains an area into which the user types input. The third line can contain buttons that let the user confirm or cancel the Dialog input.

The class variable for the Dialog widget is dialogWidgetClass.

When creating a Dialog widget instance, the following resources are retrieved from the argument list or from the resource database:

Name	Type	Default
XtNbackground	Pixel	XtDefaultBackground
XtNbackgroundPixmap	Pixmap	None
XtNborderColor	Pixel	XtDefaultForeground
XtNborderPixmap	Pixmap	None
XtNborderWidth	Dimension	1
XtNdestroyCallback	XtCallbackList	NULL
XtNheight	Dimension	computed at create
XtNlabel	String	Label name
XtNmappedWhenManaged	Boolean	True
XtNmaximumLength	int	256
XtNsensitive	Boolean	True
XtNtranslations	TranslationTable	None
XtNvalue	char*	NULL
XtNwidth	Dimension	computed at create
XtNx	Position	NULL
XtNy	Position	NULL

For an explanation of the common widget resources associated with the Dialog widget, see Section 2.3. The new resources associated with the Dialog widget are:

XtNlabel	Specifies the label string that is to be displayed.
XtNmaximumLength	Specifies the maximum number of input characters.

XtNvalue	Specifies a pointer to the input string.

The instance name of the label widget within the Dialog widget is "label", and the instance name of the Dialog value widget is "value".

To create a Dialog widget instance, you can use XtCreateWidget and specify the class variable dialogWidgetClass.

To add a child button to the Dialog box, use XtCreateWidget and specify widget ID of the previously created Dialog box as the parent of each child. When creating buttons, you do not have to specify form constraints. The Dialog box will automatically add the constraints.

To return the character string in the text field, use XtDialogGetValueString.

```
char *XtDialogGetValueString(w)
    Widget w;
```

w	Specifies the widget ID of the Dialog box.

If a string was specified in the XtNvalue resource, Dialog will store the input directly into the string.

To remove a child button from the Dialog box, use XtUnmanageChild or XtDestroyWidget and specify the widget ID of the child.

To destroy a Dialog widget instance, use XtDestroyWidget and specify the widget ID of the Dialog widget. All children of the Dialog are automatically destroyed at the same time.

3.10 List Widget

The List widget is a rectangle that contains a list of strings formatted into rows and columns. When one of the strings is selected, it is highlighted, and an application callback routine is invoked.

The class variable for the List widget is listWidgetClass.

When creating a List widget instance, the following resources are retrieved from the argument list or from the resource database:

Name	Type	Default
XtNbackground	Pixel	XtDefaultBackground
XtNbackgroundPixmap	Pixmap	None
XtNborderColor	Pixel	XtDefaultForeground
XtNborderPixmap	Pixmap	None
XtNborderWidth	Dimension	1

Name	Type	Default
XtNcallback	XtCallbackList	NULL
XtNcolumnSpacing	Dimension	6
XtNcursor	Cursor	left_ptr
XtNdefaultColumns	int	2
XtNdestroyCallback	XtCallbackList	NULL
XtNfont	XFontStruct*	XtDefaultFont
XtNforceColumns	Boolean	False
XtNforeground	Pixel	XtDefaultForeground
XtNheight	Dimension	Contains list exactly
XtNinsensitiveBorder	Pixmap	Gray
XtNinternalHeight	Dimension	2
XtNinternalWidth	Dimension	4
XtNlist	String *	List name
XtNlongest	int	Longest item
XtNmappedWhenManaged	Boolean	
XtNnumberStrings	int	Number of strings
XtNpasteBuffer	Boolean	False
XtNrowSpacing	Dimension	4
XtNsensitive	Boolean	True
XtNtranslations	TranslationTable	None
XtNverticalList	Boolean	False
XtNwidth	Dimension	Contains list exactly
XtNx	Position	0
XtNy	Position	0

For an explanation of the common widget resource associated with the List widget, see Section 2.3. The new resources associated with the List widget are:

XtNcolumnSpacing XtNrowSpacing	Specify the amount of space between each of the rows and columns in the list.
XtNdefaultColumns	Specifies the default number of columns, which is used when neither the width nor the height of the List widget is specified or when XtNforceColumns is True.
Xtfont	Specifies the font for the list text.

XtNforceColumns	Specifies that the default number of columns is to be used no matter what the current size of the List widget is.
XtNheight	Specifies the height of the List widget. The default value is the minimum height that will contain the entire list with the spacing values specified. If the specified height is larger than the minimum, the list is put in the upper left corner.
XtNinsensitiveBorder	Specifies the border to use when it is not sensitive.
XtNinternalHeight	Represents a margin, in pixels, between the top and bottom of the list and the edges of the List widget.
XtNinternalWidth	Represents a margin, in pixels, between the left and right edges of the list and the edges of the List widget.
XtNlist	Specifies the array of text strings that is to displayed in the List widget. If the default for XtNnumberStrings is used, the list must be null-terminated. If a value is not specified for the list, the number of strings is set to 1, and the name of the widget is used as the list.
XtNlongest	Specifies the length of the longest string in the current list in pixels. If the client knows the length, it should specify it. The List widget will compute a default length by searching through the list.
XtNnumberStrings	Specifies the number of strings in the current list. If a value is not specified, the list must be null-terminated.
XtNpasteBuffer	If this is True, then the value of the string selected will be put into X cut buffer 0.
XtNsensitive	If set to False, the List widget will change its window border to XtNinsensitiveBorder and display all items in the list as stippled strings. While the List widget is insensitive, no item in the list can be selected or highlighted.

XtNverticalList	If this is True, the elements in the list are arranged vertically; if False, the elements are arranged horizontally.
XtNwidth	Specifies the width of the List widget. The default value is the minimum width that will contain the entire list with the spacing values specified. If the specified width is larger than the minimum, the list is put in the upper left corner.

The List widget has three predefined actions: Set, Unset, and Notify. Set and Unset allow switching the foreground and background colors for the current list item. Notify allows processing application callbacks.

The following is the default translation table used by the List Widget:

```
        <Btn1Down>,<Btn1Up>:             Set( ) Notify( )
```

To create a List widget instance, use XtCreateWidget and specify the class variable listWidgetClass.

To destroy a List widget instance, use XtDestroyWidget and specify the widget ID of the List widget.

The List widget supports two callback lists:

- XtNdestroyCallback
- XtNcallback

The notify action executes the callbacks on the the XtNcallback list.

The call_data argument passed to callbacks on the XtNcallback list is a pointer to an XtListReturnStruct structure, defined in <X11/List.h>:

```
typedef struct _XtListReturnStruct {
        String string;              /* string shown in the list. */
        int index;                  /* index of the item selected. */
} XtListReturnStruct;
```

3.10.1 Changing the List

To change the list that is displayed, use XtListChange.

```
void XtListChange(w, list, nitems, longest, resize)
     Widget w;
     String * list;
     int nitems, longest;
     Boolean resize;
```

w	Specifies the widget ID.
list	Specifies the new list for the list widget to display.
nitems	Specifies the number of items in the list. If a value less than 1 is specified, list must be null terminated.
longest	Specifies the length of the longest item in the list in pixels. If a value less than 1 is specified, the List widget calculates the value for you.
resize	Specifies a Boolean value that indicates whether the List widget should try to resize itself (True) or not (False) after making the change. Note that the constraints of the parent of this widget are always enforced, regardless of the value specified.

XtListChange changes the list of strings that the List widget is to display.

3.10.2 Highlighting an Item

To highlight an item in the list use, XtListHighlight

```
void XtListHighlight(w, item);
     Widget w;
     int item;
```

w	Specifies the widget ID.
item	Specifies the index into the current list that indicates the item to be highlighted.

Only one item can be highlighted at a time. If an item is already highlighted when XtListHighlight is called, the highlighted item is immediately unhighlighted and the new item is highlighted.

3.10.3 Unhighlighting an Item

To unhighlight the currently highlighted item in the list, use XtListUnhighlight.

```
void XtListUnhightlight(w);
     Widget w;
```

w Specifies the widget ID.

3.10.4 Retrieving the Currently Selected Item

To retrieve an item in the list use, XtListShowCurrent

```
XtListReturnStruct *XtListShowCurrent(w);
     Widget w;
```

w Specifies the widget ID.

The XtListShowCurrent function returns a pointer to an XtListReturnStruct structure, contains the currently highlighted item. If the value of the index member is XT_LIST_NONE, the string member is undefined, which indicates that no item is currently selected.

3.11 Grip Widget

The Grip widget provides a small region in which user input events (such as ButtonPressor ButtonRelease) may be handled. The most common use for the grip is as an attachment point for visually repositioning an object, such as the pane border in a VPaned widget.

The class variable for the Grip widget is gripWidgetClass.

When creating a Grip widget instance, the following resources are retrieved from the argument list or from the resource database:

Name	Type	Default
XtNborderColor	Pixel	XtDefaultForeground
XtNborderPixmap	Pixmap	None
XtNborderWidth	Dimension	0
XtNcallback	XtCallbackList	None
XtNcursor	Cursor	None
XtNdestroyCallback	XtCallbackList	NULL
XtNforeground	Pixel	XtDefaultForeground
XtNheight	Dimension	8
XtNmappedWhenManaged	Boolean	True
XtNsensitive	Boolean	True
XtNtranslations	TranslationTable	None
XtNwidth	Dimension	8
XtNx	Position	0
XtNy	Position	0

For an explanation of the common widget resources associated with the Grip widget, see Section 2.3. Note that the Grip widget displays its region with the foreground pixel only.

The Grip widget does not declare any default event translation bindings, but it does declare a single action routine named GripAction in its action table. The client specifies an arbitrary event translation table giving parameters to the GripAction routine.

The GripAction action executes the callbacks on the XtNcallback list, passing as call_data a pointer to a GripCallData structure, defined in <X11/Grip.h>

```
typedef struct _GripCallData {
        XEvent *event;
        String *params;
        Cardinal num_params;
} GripCallDataRec, *GripCallData;
```

In this structure, the event field is a pointer to the input event that triggered the action, and params and num_params give the string parameters specified in the translation table for the particular event binding.

The following is an example of a GripAction translation table:

```
        <Btn1Down>:          GripAction(press)
        <Btn1Motion>:        GripAction(move)
        <Btn1Up>:            GripAction(release)
```

For a complete description of the format of action routines, see the *Guide to the XUI Toolkit Intrinsics.*

To create a Grip widget instance, use XtCreateWidget and specify the class variable gripWidgetClass.

To destroy a Command button widget instance, use XtDestroyWidget and specify the ID of the Grip widget.

Creating a Custom Widget 4

Although the task of creating a new widget may at first appear a little daunting, there is a basic simple pattern that all widgets follow. The Athena widget library contains three files that are intended to assist in writing a custom widget.

The reasons for writing a custom widget include:

- Convenient access to resource management procedures to obtain fonts, colors, and so on, even if user customization is not desired.
- Convenient access to user input dispatch and translation management procedures.
- Access to callback mechanism for building higher-level application libraries.
- Customizing the interface or behavior of an existing widget to suit a special application need.
- Desire to allow user customization of resources such as fonts, colors, and so on, or to allow convenient rebinding of keys and buttons to internal functions.
- Converting a non-X Toolkit application to use the X Toolkit.

In each of these cases, the operation needed to create a new widget is to "subclass" an existing one. If the desired semantics of the new widget are similar to an existing one, then the implementation of the existing widget should be examined to see how much work would be required to create a subclass that will then be able to share the existing class methods. Much time will be saved in writing the new widget if an existing widget class Expose, Resize and/or GeometryManager method can be shared by the subclass.

Note that some trivial uses of a "bare-bones" widget may be achieved by simply creating an instance of the Core widget. The class variable to use when creating a Core widget is widgetClass. The geometry of the Core widget is determined entirely by the parent widget.

It is very often the case than an application will have a special need for a certain set of functions and that many copies of these functions will be needed. For example, when converting an older application to use the X

Toolkit, it may be desirable to have a "Window Widget" class that might have the following semantics:

- Allocate two drawing colors in addition to a background color
- Allocate a text font
- Execute an application-supplied function to handle exposure events
- Execute an application-supplied function to handle user input events

It is obvious that a completely general-purpose WindowWidgetClass could be constructed that would export all class methods as callbacks lists, but such a widget would be very large and would have to choose some arbitrary number of resources such as colors to allocate. An application that used many instances of the general-purpose widget would therefore unnecessarily waste many resources.

In this section, an outline will be given of the procedure to follow to construct a special-purpose widget to address the items listed above. The reader should refer to the appropriate sections of this manual for complete details of the material outlined here. In addition, Section 1.4 of the *Intrinsics* should be read in conjunction with this section.

All Athena widgets have three separate files associated with them:

- A "public" header file containing declarations needed by applications programmers
- A "private" header file containing additional declarations needed by the widget and any subclasses
- A source code file containing the implementation of the widget

This separation of functions into three files is suggested for all widgets, but nothing in the X Toolkit actually requires this format. In particular, a private widget created for a single application may easily combine the "public" and "private" header files into a single file or merge the contents into another application header file. Similarly, the widget implementation can be merged into other application code.

In the following example, the public header file <X11/Template.h>, the private header file <X11/TemplateP.h> and the source code file <X11/Template.c> will be modified to produce the "WindowWidget" described above. In each case, the files have been designed so that a global string replacement of "Template" and "template" with the name of your new widget, using the appropriate case, can be done.

4.1 Public Header File

The public header file contains declarations that will be required by any application module that needs to refer to the widget; whether to create an instance of the class, to perform an XtSetValues operation, or to call a public routine implemented by the widget class.

The contents of the Template public header file, <X11/Template.h>, are:

```
#include <X11/copyright.h>

/* XConsortium: Template.h,v 1.2 88/10/25 17:22:09 swick Exp $ */
/* Copyright Massachusetts Institute of Technology 1987, 1988 */

#ifndef _Template_h
#define _Template_h

/****************************************************************
 *
 * Template widget
 *
 ****************************************************************/

/* Resources:

 Name                Class               RepType      Default Value
 ----                -----               -------      -------------
 background          Background          Pixel        XtDefaultBackground
 border              BorderColor         Pixel        XtDefaultForeground
 borderWidth         BorderWidth         Dimension    1
 destroyCallback     Callback            Pointer      NULL
 height              Height              Dimension    0
 mappedWhenManaged   MappedWhenManaged   Boolean      True
 sensitive           Sensitive           Boolean      True
 width               Width               Dimension    0
 x                   Position            Position     0
 y                   Position            Position     0

*/

/* define any special resource names here that are not in <X11/StringDefs.h> */

#define XtNtemplateResource          "templateResource"

#define XtCTemplateResource          "TemplateResource"
```

```
/* declare specific TemplateWidget class and instance datatypes */

typedef struct _TemplateClassRec*       TemplateWidgetClass;
typedef struct _TemplateRec*            TemplateWidget;

/* declare the class constant */

extern WidgetClass templateWidgetClass;

#endif  _Template_h
```

Note that most of this file is documentation. The crucial parts are the last 8 lines where macros for any private resource names and classes are defined and where the widget class datatypes and class record pointer are declared.

For the "WindowWidget", we want two drawing colors, a callback list for user input and an **XtNexposeCallback** callback list, and we will declare three convenience procedures, so we need to add the following:

```
/* Resources:
     ...
 callback           Callback        Callback        NULL
 drawingColor1      Color           Pixel           XtDefaultForeground
 drawingColor2      Color           Pixel           XtDefaultForeground
 exposeCallback     Callback        Callback        NULL
 font               Font            XFontStruct*    XtDefaultFont
     ...
 */

#define XtNdrawingColor1                "drawingColor1"
#define XtNdrawingColor2                "drawingColor2"
#define XtNexposeCallback               "exposeCallback"

extern Pixel WindowColor1( /* Widget */);
extern Pixel WindowColor2( /* Widget */);
extern Font  WindowFont( /* Widget */);
```

Note that we have chosen to call the input callback list by the generic name, XtNcallback, rather than a specific name. If widgets that define a single user-input action all choose the same resource name then there is greater possibility for an application to switch between widgets of different types.

4.2 Private Header File

The private header file contains the complete declaration of the class and instance structures for the widget and any additional private data that will be required by anticipated subclasses of the widget. Information in the private header file is normally hidden from the application and is designed to be accessed only through other public procedures, for example, **XtSetValues**.

The contents of the Template private header file, <**X11/TemplateP.h**>, are:

```
#include <X11/copyright.h>

/* XConsortium: TemplateP.h,v 1.2 88/10/25 17:31:47 swick Exp $ */
/* Copyright Massachusetts Institute of Technology 1987, 1988 */

#ifndef _TemplateP_h
#define _TemplateP_h

#include "Template.h"
/* include superclass private header file */
#include <X11/CoreP.h>

/* define unique representation types not found in <X11/StringDefs.h> */

#define XtRTemplateResource            "TemplateResource"

typedef struct {
    int empty;
} TemplateClassPart;

typedef struct _TemplateClassRec {
    CoreClassPart        core_class;
    TemplateClassPart    template_class;
} TemplateClassRec;

extern TemplateClassRec templateClassRec;

typedef struct {
    /* resources */
    char* resource;
    /* private state */
} TemplatePart;

typedef struct _TemplateRec {
    CorePart             core;
    TemplatePart        template;
} TemplateRec;

#endif  _TemplateP_h
```

The private header file includes the private header file of its superclass, thereby exposing the entire internal structure of the widget. It may not always be advantageous to do this; your own project development style will dictate the appropriate level of detail to expose in each module.

The "WindowWidget" needs to declare two fields in its instance structure to hold the drawing colors, a resource field for the font and a field for the expose and user input callback lists:

```
typedef struct {
    /* resources */
    Pixel color_1;
    Pixel color_2;
    XFontStruct* font;
    XtCallbackList expose_callback;
    XtCallbackList input_callback;
    /* private state */
    /* (none) */
} WindowPart;
```

4.3 Widget Source File

The source code file implements the widget class itself. The unique part of this file is the declaration and initialization of the widget class record structure and the declaration of all resources and action routines added by the widget class.

The contents of the Template implementation file, <X11/Template.c>, are:

```
#include <X11/copyright.h>

/* XConsortium: Template.c,v 1.2 88/10/25 17:40:25 swick Exp $ */
/* Copyright Massachusetts Institute of Technology 1987, 1988 */

#include <X11/IntrinsicP.h>
#include <X11/StringDefs.h>
#include "TemplateP.h"

static XtResource resources[] = {
#define offset(field) XtOffset(TemplateWidget, template.field)
```

```
    /* {name, class, type, size, offset, default_type, default_addr}, */
    { XtNtemplateResource, XtCTemplateResource, XtRTemplateResource, sizeof(char*),
                        offset(resource), XtRString, "default" },
#undef offset
};

static void TemplateAction(/* Widget, XEvent*, String*, Cardinal* */);

static XtActionsRec actions[] =
{
    /* {name,           procedure}, */
    {"template",        TemplateAction},
};

static char translations[] =
"   <Key>:              template( ) \n\
";

TemplateClassRec templateClassRec = {
  { /* core fields */
    /* superclass          */           (WidgetClass) &widgetClassRec,
    /* class_name          */           "Template",
    /* widget_size         */           sizeof(TemplateRec),
    /* class_initialize    */           NULL,
    /* class_part_initialize            */      NULL,
    /* class_inited        */           FALSE,
    /* initialize          */           NULL,
    /* initialize_hook     */           NULL,
    /* realize             */           XtInheritRealize,
    /* actions             */           actions,
    /* num_actions         */           XtNumber(actions),
    /* resources           */           resources,
    /* num_resources       */           XtNumber(resources),
    /* xrm_class           */           NULLQUARK,
    /* compress_motion */               TRUE,
    /* compress_exposure                */      TRUE,
    /* compress_enterleave */           TRUE,
    /* visible_interest    */           FALSE,
    /* destroy             */           NULL,
    /* resize              */           NULL,
    /* expose              */           NULL,
    /* set_values          */           NULL,
    /* set_values_hook */               NULL,
    /* set_values_almost                */      XtInheritSetValuesAlmost,
    /* get_values_hook  */            NULL,
    /* accept_focus     */            NULL,
    /* version          */            XtVersion,
    /* callback_private */            NULL,
    /* tm_table         */            translations,
    /* query_geometry   */            XtInheritQueryGeometry,
    /* display_accelerator            */    XtInheritDisplayAccelerator,
    /* extension        */            NULL
  },
  { /* template fields */
    /* empty            */            0
  }
};

WidgetClass templateWidgetClass = (WidgetClass)&templateClassRec;
```

The resource list for the "WindowWidget" might look like the following:

```
static XtResource resources[] = {
#define offset(field) XtOffset(WindowWidget, window.field)
    /* {name, class, type, size, offset, default_type, default_addr}, */
    { XtNdrawingColor1, XtCColor, XtRPixel, sizeof(Pixel),
                        offset(color_1), XtRString, XtDefaultForeground },
    { XtNdrawingColor2, XtCColor, XtRPixel, sizeof(Pixel),
                        offset(color_2), XtRString, XtDefaultForeground },
    { XtNfont, XtCFont, XtRFontStruct, sizeof(XFontStruct*),
                        offset(font), XtRString, XtDefaultFont },
    { XtNexposeCallback, XtCCallback, XtRCallback, sizeof(XtCallbackList),
                        offset(expose_callback), XtRCallback, NULL },
    { XtNcallback, XtCCallback, XtRCallback, sizeof(XtCallbackList),
                        offset(input_callback), XtRCallback, NULL },
#undef offset
};
```

The user input callback will be implemented by an action procedure which passes the event pointer as call_data. The action procedure is declared as:

```
/* ARGSUSED */
static void InputAction(w, event, params, num_params)
    Widget w;
    XEvent *event;
    String *params;                /* unused */
    Cardinal *num_params;          /* unused */
{
    XtCallCallbacks(w, XtNcallback, (caddr_t)event);
}

static XtActionsRec actions[] =
{
    /* {name,            procedure}, */
    {"input",            InputAction},
};
```

The default input binding will be to execute the input callbacks on **KeyPress** and **ButtonPress**:

```
static char translations[] =
"   <Key>:               input() \n\
    <BtnDown>:           input() \
";
```

In the class record declaration and initialization, the only field that is different from the Template is the expose procedure:

```
/* ARGSUSED */
static void Redisplay(w, event, region)
    Widget w;
    XEvent *event;       /* unused */
    Region region;
{
    XtCallCallbacks(w, XtNexposeCallback, (caddr_t)region);
}

WindowClassRec windowClassRec = {

    ...

    /* expose            */                  Redisplay,
```

The "WindowWidget" will also declare three public procedures to return the drawing colors and the font id, saving the application the effort of constructing an argument list for a call to **XtGetValues**:

```
Pixel WindowColor1(w)
    Widget w;
{
    return ((WindowWidget)w)->window.color_1;
}

Pixel WindowColor2(w)
    Widget w;
{
    return ((WindowWidget) w)->window.color_2;
}

Font WindowFont(w)
    Widget w;
{
    return ((WindowWidget) w)->window.font->fid;
}
```

The "WindowWidget" is now complete. The application can retrieve the two drawing colors from the widget instance by calling either XtGetValues, or the **WindowColor** functions. The actual window created for the "WindowWidget" is available by calling the XtWindow function.

To test the new "WindowWidget", you may substitute "window" for "command" in the sample program given in Section 2.7.3.

Index

HOW TO ORDER ADDITIONAL DOCUMENTATION

DIRECT TELEPHONE ORDERS

In Continental USA
and New Hampshire,
Alaska or Hawaii
call **800-DIGITAL**

In Canada
call **800-267-6215**

DIRECT MAIL ORDERS (U.S. and Puerto Rico*)

DIGITAL EQUIPMENT CORPORATION
P.O. Box CS2008
Nashua, New Hampshire 03061

DIRECT MAIL ORDERS (Canada)

DIGITAL EQUIPMENT OF CANADA LTD.
100 Herzberg Road
Kanata, Ontario K2K 2A6
Attn: Direct Order Desk

INTERNATIONAL

DIGITAL EQUIPMENT CORPORATION
PSG Business Manager
c/o Digital's local subsidiary
or approved distributor

Internal orders should be placed through the Software Distribution Center (SDC), Digital Equipment Corporation, Westminster, Massachusetts 01473

***Any prepaid order from Puerto Rico must be placed with the Local Digital Subsidiary: 809–754–7575**

ULTRIX
Worksystem Software
Guide to the X Toolkit Widgets:
C Language Binding
AA-MF09A-TE

Reader's Comments

Note: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement. ______________________________

____

____

____

Did you find errors in this manual? If so, specify the error and the page number.

____

____

____

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) ______________________________

Name ______________________________ Date ______________

Organization ______________________________

Street ______________________________

City ______________________________ State ______ Zip Code or Country ______________

· Do Not Tear - Fold Here and Tape

digital™

No Postage
Necessary
if Mailed
in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

Digital Equipment Corporation
Documentation Manager
ULTRIX Documentation Group
ZKO3-3/X18
110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

- Do Not Tear - Fold Here

ULTRIX
Worksystem Software
Guide to the X Toolkit Widgets:
C Language Binding

Reader's Comments

AA-MF09A-TE

Note: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement. ______________________________

____

____

____

Did you find errors in this manual? If so, specify the error and the page number.

____

____

____

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) ______________________________

Name ______________________________ Date ______________________________

Organization ______________________________

Street ______________________________

City ______________________________ State __________ Zip Code or Country ______________

- Do Not Tear - Fold Here and Tape

digital™

No Postage
Necessary
if Mailed
in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

Digital Equipment Corporation
Documentation Manager
ULTRIX Documentation Group
ZKO3-3/X18
110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

- Do Not Tear - Fold Here