Structure of a Motif Program

Motif uses the same event−driven programming model as the X Toolkit Intrinsics. At its core, a Motif application waits for the user to provide input, usually by pressing a key, moving the mouse, or clicking a mouse button. Such an action by the user causes the X server to generate one or more X Window System events. Xt listens for these events and dispatches them to the appropriate Motif widget, usually the widget to which the user directed the input. The widget may take some action as a result of the user input. If the application has asked to be notified of that action, the widget "calls back" to the application−−that is, it invokes an application callback procedure. When both Motif and the application have finished responding to the user input, the application waits for the user to provide more input. This cycle of user−initiated events and application response, called the *event loop*, continues until the user terminates the application.

For simple applications, the Intrinsics and Motif toolkits do everything necessary for dispatching user input to widgets. The application must take the following actions:

- Include the required header files
- Initialize the Intrinsics
- Create one or more widgets
- Define callback procedures and attach them to widgets
- Make the widgets visible
- Enter the event loop

This chapter discusses each of these actions. The following table summarizes these steps and some of the procedures the application needs to call. Note that some of these steps are different when the application uses UIL and MRM. See Chapter 4 for more information.

Table 1 Steps in Writing Widget Programs

Step	*Description*	*Related Functions*
1	Include required header files.	*#include <Xm/Xm.h> #include <Xm/widget.h> or <Xm/XmAll.h>*
2	Initialize Xt Intrinsics. Do steps 3 and 4 for each widget.	*XtAppInitialize()*
3	Create widget.	*XtSetArg() XtCreateManagedWidget() or XtVaCreateManagedWidget() or* **XmCreate***<WidgetName>()* followed by *XtManageChild(widget)*
4	Add callback routines.	*XtAddCallback()*
5	Realize widgets.	*XtRealizeWidget(parent)*
6	Enter event loop.	*XtAppMainLoop()*

Including Header Files

All Motif applications must include the file `<Xm/Xm.h>`. This file contains definitions that all applications need. It also includes the Xt header files `<X11/Intrinsic.h>` and `<X11/StringDefs.h>`.

Each Motif widget also has an *#include* file. An application must include the header files for all widgets it creates. In addition, some groups of Motif routines have their own header files. For example, an application that uses any of the Motif clipboard routines must include the file `<Xm/CutPaste.h>`. Required *#include* files for each Motif widget and routine are documented in the *Motif Programmer's Reference*.

Following is an example of including header files for an application that uses only a Text widget:

```
#include <Xm/Xm.h>
#include <Xm/Text.h>
```

Instead of using a large number of header files for particular widgets and routines, an application can include `<Xm/XmAll.h>`. This file incorporates all documented Motif header files.

Initializing the Intrinsics

The first task of a Motif application is to initialize the Intrinsics. Most applications can perform the initialization by calling the routine **XtAppInitialize**. This is a convenience routine that combines several initialization steps, each of which the application can take separately by calling a specialized Xt routine:

1. Initialize the state of the Intrinsics. An application can also do this by calling **XtToolkitInitialize**.
2. Create an application context. Xt uses this construct to contain the information it associates with each instance of an application. Its purpose is to allow multiple instances of an application to run in a single address space. Most applications need only create an application context and pass it to Intrinsics routines that take an application context as an argument. The data type is **XtAppContext**. An application can create an application context explicitly by calling **XtCreateApplicationContext**.
3. Open a connection to a display and attach it to an application context. When an application uses **XtAppInitialize**, the display specification comes from the command line invoking the application or from the user's environment. After opening the display, Xt builds a resource database by processing resource defaults and command–line options. The construction of this database is described in the next section. An application can perform these steps explicitly by calling **XtOpenDisplay**. If an application already has an open display as a result of calling **XOpenDisplay**, it can attach the display to an application context and build the initial resource database by calling **XtDisplayInitialize**.
4. Create a top–level shell widget for the application. **XtAppInitialize** creates an ApplicationShell and returns it as the function's return value. An application can create a top–level shell by calling **XtAppCreateShell**.

Following is an example of a simple call to **XtAppInitialize**:

```
int main(int argc, char **argv)
{
    Widget         app_shell;
    XtAppContext   app;
    app_shell = XtAppInitialize(&app, "Example",
        (XrmOptionDescList) NULL, 0, &argc, argv,
        (String *) NULL, (ArgList) NULL, 0);
}
```

The Initial Resource Database

The **XtDisplayInitialize** routine builds the initial resource database for the application. An application rarely needs to call this routine directly; it is called by **XtOpenDisplay**, which in turn is called by **XtAppInitialize**.

XtDisplayInitialize builds a separate resource database for each display connection. The initial database combines resource settings from the command line, the display, an application class defaults file, and user defaults files that may be specialized according to the application or the host on which the application is running. The application class defaults and the user's per–application defaults may be further specialized according to the language environment and possibly according to a general–purpose *customization* resource. The resources in the initial database may pertain to particular widgets or widget classes or to the application as a whole. When the application creates widgets, the resource settings from the database are often the source for the initial values of widget resources.

The remainder of this section describes the order in which **XtDisplayInitialize** loads each component of the database and how it derives the location of that component.

File Search Paths

In loading the application class defaults and the user's per–application defaults, **XtDisplayInitialize** calls **XtResolvePathname** to determine which files to read. **XtResolvePathname** uses file search paths. Each path is a set of patterns that may contain special character sequences for which **XtResolvePathname** substitutes runtime values when it searches for a file. It uses the following substitutions in building the path:

- %*N* is replaced by the class name of the application, as specified by the *application_class* argument to **XtAppInitialize**, **XtOpenDisplay**, or **XtDisplayInitialize**.
- %*C* is replaced by the value of the *customization* resource.
- %*L* is replaced by the display's language specification. This may come from the *xnlLanguage* resource, the locale of the application, or an application callback procedure. See Chapter 11 for more information. The format of the language specification is implementation dependent; it may have language, territory, and code set components.
- %*l* is replaced by the language part of the language specification.
- %*t* is replaced by the territory part of the language specification.
- %*c* is replaced by the code set part of the language specification.
- %% is replaced by %.

If the language specification is not defined, or if one of its parts is missing, a % element that references it is replaced by NULL.

The paths contain a series of elements separated by colons. Each element denotes a filename, and the filenames are looked up left−to−right until one of them succeeds. Before doing the lookup, substitutions are performed.

> **Note:** The Intrinsics use the X/Open convention of collapsing multiple adjoining slashes in a filename into one slash.

Initial Database Components

The **XtDisplayInitialize** function loads the resource database by merging in resources from these sources, in order of precedence (that is, each component takes precedence over the following components):

- The application command line
- Per−host user environment resource file on the local host
- Screen−specific resources for the default screen of the display
- Resource property on the server or user preference resource file on the local host
- Application−specific user resource file on the local host
- Application−specific class resource file on the local host

Command−Line Specifications

XtDisplayInitialize calls the X Resource Manager function **XrmParseCommand** to extract resource settings from the command line by which the user invoked the application. The arguments and number of arguments on the command line come from the *argv* and *argc* arguments to **XtAppInitialize**, **XtOpenDisplay**, or **XtDisplayInitialize**. Xt maintains a standard set of command−line options, such as *-background* and *-geometry*, for specifying resource settings. An application can specify additional options in arguments to **XtAppInitialize**, **XtOpenDisplay**, or **XtDisplayInitialize**. The user can supply the *-xrm* option to set any resource in the database.

Per−Host User Resources

To load the per−host user environment resources, **XtDisplayInitialize** uses the filename specified by the **XENVIRONMENT** environment variable. If **XENVIRONMENT** is not defined, **XtDisplayInitialize** looks for the file `$HOME/.Xdefaults-`*host*, where *host* is the name of the host on which the application is running (that is, the name of the client host, not the server host).

Screen−Specific Resources

To load screen−specific resources, **XtDisplayInitialize** looks for a SCREEN_RESOURCES property on the root window of the default screen of the display. The SCREEN_RESOURCES property typically results from invoking the **xrdb** command when some resources are not defined for all screens.

> **Note:** When Xt needs to fetch resources for a screen other than the default screen of the display−−for example, when the application creates a widget on another screen−−it uses the SCREEN_RESOURCES property of that screen instead of the SCREEN_RESOURCES property of the default screen.

Server or User−Preference Resources

To load the server resource property or user preference file, **XtDisplayInitialize** first looks for a RESOURCE_MANAGER property on the root window of the display's screen 0. The RESOURCE_MANAGER property typically results from invoking the **xrdb** command when some resources are defined for all screens. If that property does not exist, **XtDisplayInitialize** looks for the file `$HOME/.Xdefaults`.

User Application File

To load the user's application resource file, **XtDisplayInitialize** performs the following steps:

1. Use **XUSERFILESEARCHPATH** to look up the file, performing appropriate substitutions.
2. If that fails, or if **XUSERFILESEARCHPATH** is not defined, and if **XAPPLRESDIR** is defined, use an implementation−dependent search path containing at least seven entries, in the following order and with the following directory prefixes and substitutions:
   - **$XAPPLRESDIR** with %C, %N, %L or with %C, %N, %l, %t, %c
   - **$XAPPLRESDIR** with %C, %N, %l
   - **$XAPPLRESDIR** with %C, %N

$XAPPLRESDIR with %N, %L or with %N, %l, %t, %c
$XAPPLRESDIR with %N, %l
$XAPPLRESDIR with %N
$HOME with %N

where **$XAPPLRESDIR** is the value of the **XAPPLRESDIR** environment variable and $HOME is the user's home directory.

3. If **XAPPLRESDIR** is not defined, use an implementation–dependent search path containing at least six entries, in the following order and with the following directory prefixes and substitutions:

$HOME with %C, %N, %L or with %C, %N, %l, %t, %c
$HOME with %C, %N, %l
$HOME with %C, %N
$HOME with %N, %L or with %N, %l, %t, %c
$HOME with %N, %l
$HOME with %N

Application Class Resource File

To load the application–specific class resource file, **XtDisplayInitialize** performs the appropriate substitutions on the path specified by the **XFILESEARCHPATH** environment variable. If that fails, or if **XFILESEARCHPATH** is not defined, **XtDisplayInitialize** uses an implementation–dependent search path containing at least six entries, in the following order and with the following substitutions:

%C, %N, %S, %T, %L or %C, %N, %S, %T, %l, %t, %c
%C, %N, %S, %T, %l
%C, %N, %S, %T
%N, %S, %T, %L or %N, %S, %T, %l, %t, %c
%N, %S, %T, %l
%N, %S, %T

where the substitution for *%S* is usually NULL and the substitution for *%T* is usually *app–defaults*.

If no application–specific class resource file is found, **XtDisplayInitialize** looks for any fallback resources that may have been defined by a call to **XtAppInitialize** or **XtAppSetFallbackResources**.

Creating Widgets

The top−level widget returned by **XtAppInitialize** or **XtAppCreateShell** is the root of a program's widget hierarchy for a given display or logical application. After initializing the Intrinsics, the application can proceed to create the remainder of the widget hierarchy it needs to start the program.

Widget creation is a two−stage process. In the first stage, the application creates the widget hierarchy but does not assign windows to the widgets. In the second stage, the application assigns windows and makes them visible. These stages are separate because, otherwise, window geometry might have to be recomputed each time a child is added. This computation can require a great deal of communication with the X server and take a long time. Instead, initial window geometry is computed only once. For more information, see Section 3.5.

The general routine for creating a widget is **XtCreateWidget**. The required arguments to this routine are the widget's name, class, and parent widget. You can also provide initial resource values for the widget, as discussed in the next section. **XtVaCreateWidget** is a version of **XtCreateWidget** that uses a variable−length argument list.

Motif has a convenience routine for creating a widget of each Motif class. The name of such a routine is usually **XmCreate**<*widget*>, where *widget* represents the widget class. For example, the convenience routine for creating a Text widget is **XmCreateText**. These routines do not require the widget−class argument.

Some convenience routines, such as **XmCreateMenuBar**, create specialized widgets. These routines usually set some initial resource values to configure the widget for a particular use−−for example, to configure a RowColumn widget for use as a MenuBar. In some cases, such as **XmCreatePulldownMenu** and **XmCreateScrolledList**, these routines create a widget hierarchy rather than a single widget. The documentation for each convenience routine in the *Motif Programmer's Reference* explains what the routine does.

Using a Motif creation routine is generally preferable to calling **XtCreateWidget**. In addition to creating multiple widgets and setting appropriate resources, these routines sometimes perform optimizations. For example, some convenience routines add **XmNdestroyCallback** procedures to free memory when the widget is destroyed.

> **Note:** Every widget except a top−level widget must have a parent at the time the widget is created.

An application can use **XtDestroyWidget** to destroy a widget. An application can specify values for resources when it creates a widget and anytime thereafter. Note that hardcoding widget resources reduces the ability of the user to customise the widgets. It can retrieve resource values after creating a widget.

Setting Resources during Widget Initialization

When an application creates a widget, the creation routine sets the widget's initial resource values from the following sources, in order (that is, each succeeding component takes precedence over preceding components):

- Default values for resources specified by the widget class and its superclasses
- Resource values from the initial resource database
- Resource values specified by the application in its call to the widget creation routine

Each widget class can have its own *initialize* procedure. After setting the initial resource values, the widget creation routine calls the *initialize* procedure for each class in the widget's class hierarchy, in superclass−to−subclass order. The *initialize* procedure can set new values for resources, possibly based on other resource values in the widget or its ancestors. In some cases, an *initialize* procedure forces a resource to have a particular value, regardless of whether the user or application has specified another value. In other cases, the *initialize* procedure might set a resource value only if the user or application has not specified another value.

The documentation for each widget class in the *Motif Programmer's Reference* lists the data type and default value for each resource. For resources whose default values are computed dynamically, the documentation describes how the default values are determined.

Arguments that Specify Resource Values

To specify initial resource values in a call to a widget creation routine, an application supplies two arguments: a list of elements representing resource settings and an integer specifying the number of elements in the list. Each element in the list is a structure of type *Arg*. This structure has two members: a string representing the name of the resource, and a value specifier representing the resource value. The value specifier is of type **XtArgVal**. This is a data type large enough to hold a *long* or one of several types of pointers to other data. If the resource value is of a type small

enough to fit into an **XtArgVal**, the value specifier contains the resource value itself; otherwise, it contains a pointer to the actual value. For most resources, an application supplies integer values (including such types as *Position* and *Dimension*) directly in the value specifier; otherwise, the application supplies a pointer to the value.

The most common way to set up a list of resource specifications is to declare a list of *Arg* elements large enough to hold all the specifications and then to use **XtSetArg** to insert each specification into the list. An application should always use a sequence of calls to **XtSetArg** in the following way to avoid mistakes in building the list:

```
...
Widget      text;
Arg         args[10];
Cardinal    n;

n = 0;
XtSetArg(args[n], XmNrows, 10);         n++;
XtSetArg(args[n], XmNcolumns, 80);      n++;
text = XmCreateText("text", parent, args, n);
```

Instead of using lists of *Arg* structures, the variable−argument routines that specify resource values take a variable number of pairs of resource names and values as arguments. The resource value in each pair is of type **XtArgVal**, with the same meaning as the value in an *Arg* structure. The application can provide two special strings in place of a resource name. If the name is **XtVaNestedList**, the next argument is interpreted as a nested list of name−value pairs. If the name is **XtVaTypedArg**, the next four arguments supply the resource value and cause it to be converted from one data type to another, as described in the following sections.

Setting Resource Values

To specify resource values after a widget has been created, an application uses **XtSetValues** or **XtVaSetValues**. **XtSetValues** takes a list of resource specifications in the same format as that used when creating a widget:

```
...
Arg         args[10];
Cardinal    n;

n = 0;
XtSetArg(args[n], XmNrows, 10);         n++;
XtSetArg(args[n], XmNcolumns, 80);      n++;
XtSetValues(text, args, n);
```

Each widget class can have its own *set_values* procedure. After setting the values specified in the argument list, **XtSetValues** calls the *set_values* procedure for each class in the widget's class hierarchy, in superclass−to−subclass order. The *set_values* procedure can set new values for resources other than those specified in the arguments to **XtSetValues**. This usually happens when the value of one resource depends on the value of another. Setting a new value for a resource that affects the widget's geometry can also cause Motif to recompute the widget's layout. In some cases a *set_values* procedure forces a resource to have a particular value, regardless of whether the application has specified another value.

Retrieving Resource Values

To retrieve resource values, an application uses **XtGetValues** or **XtVaGetValues**. The arguments are the same as those for **XtSetValues**, except that in place of a value for each resource is an address in which Motif stores the requested value:

```
...
Arg         args[10];
Cardinal    n;
short       nrows, ncolumns;

n = 0;
XtSetArg(args[n], XmNrows, &nrows);             n++;
XtSetArg(args[n], XmNcolumns, &ncolumns);       n++;
XtGetValues(text, args, n);
```

Resource Value Data Types

The documentation for each widget class in the *Motif Programmer's Reference* lists the data types to use when setting and retrieving values for

resources. The user and application do not always have to supply data of the type documented. Motif has routines, called converters, that convert resource values from one data type to another. For example, when a value for the resource database comes from a file or the command line, Motif processes the value as a string. Motif and Xt have routines to convert strings to most common resource types, including *Boolean*, *Dimension*, *Position*, *Pixel*, and **XmFontList**.

When using the standard widget creation routines, **XtSetValues**, and **XtGetValues**, an application must supply resource values or addresses of the types the widget expects. But when using the variable−argument versions of these routines, the application can supply values of any types for which routines exist to convert data of those types into values of the expected types. To provide for a resource conversion, the application supplies **XtVaTypedArg** in place of a resource name in the argument list. In place of the resource value, the application supplies four arguments:

- The resource name
- A string representing the type of the value supplied
- The value itself (of type **XtArgVal**)
- An integer representing the number of bytes in the value

For example, the following call converts the supplied string into the compound string that Motif expects for a PushButton label:

```
...
char *label = "Button";

  XtVaSetValues(button, XtVaTypedArg, XmNlabelString,
                XmRString, label, strlen(label) + 1, NULL);
```

Note that setting resource values with **XtVaTypedArg** will not work properly for certain gadget, text, and VendorShell resources.

Resource Values and Memory Management

The application is responsible for allocating and freeing memory needed for resource values it supplies when initializing a widget or setting new values. For most resources whose values are not immediate data, including strings, compound strings, and font lists, Motif makes copies of values the application supplies when it creates a widget or calls **XtSetValues**. The application can free the allocated memory anytime after the widget creation routine executes or **XtSetValues** returns:

```
...
char     *label = "Button";
XmString  label_cs;

  label_cs = XmStringCreateLocalized(label);
  XtVaSetValues(button, XmNlabelString, label_cs, NULL);
  XmStringFree(label_cs);
```

For resources whose values are not immediate data, **XtGetValues** sometimes makes a copy of values and sometimes does not. For example, Motif always makes copies of compound strings retrieved by **XtGetValues**, but it does not make copies of lists of compound strings (data of type **XmStringTable**). An application should free compound strings retrieved by **XtGetValues**, but in general it should not free values of other types unless the documentation for the particular resource in the *Motif Programmer's Reference* says the application must free that value.

The standard routines an application should use to allocate memory are **XtMalloc** and **XtNew**. The standard routine to free memory is **XtFree**. Some Motif data types have memory−management routines that an application should use instead of the more general Xt routines. For example, use **XmStringFree** to free memory for a compound string. Table 3−2 lists the Motif memory deallocation routines.

Table 1 Motif Memory Deallocation Routines

Routine	*Recovers Memory Used By:*
XmFontListEntryFree	A font list entry
XmFontListFree	A font list
XmFontListFreeFontContext	A font list context
XmImFreeXIC	The widgets associated with a specified X Input Context (XIC)
XmParseMappingFree	A parse mapping
XmParseTableFree	A parse table
XmRenderTableFree	A render table
XmRenditionFree	A rendition
XmStringConcatAndFree	Two input compound strings
XmStringFree	A compound string
XmStringFreeContext	The string scanning context
XmTabFree	A tab
XmTabListFree	A tab list

Adding Callback Procedures

Callback routines are the heart of a Motif application. Many widget classes have resources whose values are lists of callback procedures. When the user acts on a widget—for example, pressing a PushButton—Motif invokes the callback routines in the corresponding callback list. If an application needs to take some action when the user presses a PushButton, it supplies a callback routine and adds that routine to the appropriate callback list.

Callbacks are not the only means Motif uses to notify an application of a user action. An application can also supply its own action routines and event handlers. The main difference between these kinds of procedures is the level of abstraction at which Motif or Xt invokes the procedures:

- The Xt event dispatcher calls an event handler whenever an event of a particular type occurs in a specified widget.
- The Xt translation manager calls an action routine when an event sequence matches an event specification in a widget translation table. In a translation table, actions are associated with event specifications. More than one event sequence can invoke the same action routine.
- A Motif widget invokes callback procedures when user input signifies an action that is meaningful to the widget, such as activating a PushButton. Widgets often invoke callbacks from action routines. More than one action can invoke the same callback list.

Most applications use only callback procedures. Action routines and event handlers are discussed in Chapter 13.

Each callback procedure is a function of type **XtCallbackProc**. The procedure takes three arguments: a widget and two pointers to data. The first pointer is to data that the application has told the widget to pass back to the application when the callback procedure is invoked. The second pointer is to data that the widget passes to all callbacks on the callback list. A callback procedure returns no value.

The application data argument is primarily for passing data that the application maintains separately from the widget itself. The widget data argument for most Motif widgets is a pointer to a structure containing information that varies by widget class. For example, when the user changes the value of a ToggleButton, Motif invokes callback procedures with a pointer to an **XmToggleButtonCallbackStruct** structure as the third argument. This structure has three members:

- An integer indicating the reason for invoking the callback. When the user changes the value, the reason is **XmCR_VALUE_CHANGED**. Usually the reason is identified by a symbol beginning with the characters **XmCR**.
- A pointer to the **XEvent** that triggered the callback.
- An integer that indicates the new state of the ToggleButton, either selected or unselected.

The documentation for each widget class in the *Motif Programmer's Reference* describes any callback structures that the widget passes to callback procedures as widget data. Note that a callback procedure can change the values of some members of these structures. Because the order of procedures in a callback list is unspecified, an application that uses multiple callback procedures in the same list must use caution in changing these values.

Following is a simple callback procedure that an application might use to set the state of a valve when the user changes the value of a ToggleButton. The application data passed in the callback in this example might be a pointer to a valve object associated with the ToggleButton:

```
void ToggleValueChangedCB(Widget toggle, XtPointer app_data,
    XtPointer widget_data)
{
    Valve *valve_p = (Valve *) app_data;
    XmToggleButtonCallbackStruct *toggle_info =
        (XmToggleButtonCallbackStruct *) widget_data;
    ChangeValveState(*valve_p,
        ((Boolean) toggle_info->set == TRUE) ?
                            VALVE_ON: VALVE_OFF);
}
```

To register a callback procedure with a widget, an application uses **XtAddCallback** or **XtAddCallbacks** after declaring the callback procedure and creating the widget. The following code fragment creates a ToggleButton for each valve in a global list of valves:

```
...
 char      name[20];
 Widget    toggles[N_VALVES];
 int       i;
 Valve    *valve_p;

    for(i = 0, valve_p = valves; i < N_VALVES;
                                    i++, valve_p++) {
        sprintf(name, "valve_state_%d", i);
        toggles[i] = XmCreateToggleButton(parent, name,
```

```
            (ArgList) NULL, 0);
        XtAddCallback(toggles[i], XmNvalueChangedCallback,
            (XtCallbackProc) ToggleValueChangedCB,
            (XtPointer) valve_p);
}
```

To remove a callback procedure from a callback list, use **XtRemoveCallback** or **XtRemoveCallbacks**. Because Motif sometimes adds its own callbacks to callback lists, do not use **XtRemoveAllCallbacks** to remove all callbacks from a list.

Making Widgets Visible

Creating a widget does not by itself make the widget visible. Widgets become visible when the following conditions exist:

- The widget and its ancestors are *managed*. A widget is managed when the Xt and Motif geometry managers take account of the widget when computing the positions and sizes of widgets they display.
- The widget and its ancestors are *realized*. A widget is realized when it has an associated window.
- The widget and its ancestors are *mapped*. A widget is mapped when its window is displayed.

An application can manage, realize, and map widgets in separate steps, but each of these actions affects the others.

Managing Widgets

Parent widgets are responsible for managing the geometry of their children. A child can ask the parent to be given some size or position, but the parent decides whether or not to grant the request. A parent can move or resize a child without the child's permission. The process by which parent and child widgets interact to determine widget geometry is described in Chapter 14.

An application tells a widget to manage a child widget's geometry by calling **XtManageChild** or **XtManageChildren**. If the parent is realized, **XtManageChild** calls the parent class's *change_managed* procedure. This procedure can change the size or position of any of the parent's children. After calling the parent's *change_managed* procedure, **XtManageChild** realizes the child and, if the child's **XmNmappedWhenManaged** resource is True, maps it.

If the parent is not realized, **XtManageChild** marks the child as managed. Xt defers calling the parent's *change_managed* procedure until the parent is realized.

When managing more than one child of a realized parent, it is more efficient for an application to call **XtManageChildren** than to call **XtManageChild** separately for each child being managed. Widget layout can be computationally expensive, and **XtManageChild** invokes the parent's *change_managed* procedure each time it is called. **XtManageChildren** calls the parent's *change_managed* procedure only once for all children being managed.

An application tells a widget not to manage a child widget's geometry by calling **XtUnmanageChild** or **XtUnmanageChildren**. By managing and unmanaging widgets, an application can alternately display more than one set of children without having to create and destroy widgets each time the configuration of the application changes. In addition, managing a Motif dialog or PopupMenu causes the widget to pop up, and unmanaging it causes the widget to pop down.

To create a widget and then manage it in the same call, an application can use **XtCreateManagedWidget** or **XtVaCreateManagedWidget**. The Motif routines that create widgets of particular classes return unmanaged widgets. When using these routines, the application must manage the widgets by using **XtUnmanageChild** or **XtUnmanageChildren**.

Realizing Widgets

An application uses **XtRealizeWidget** to realize a widget. This routine does the following:

- In post–order, traverses the tree whose root is the widget and calls the class *change_managed* procedure for any widget in the tree that has managed children.
- Recursively traverses the tree whose root is the widget and calls the class *realize* procedure for any widget in the tree that is managed. The *realize* procedure creates the widget's window.
- Maps the widget's managed children whose **XmNmappedWhenManaged** resource is True. If the widget is a top–level widget whose **XmNmappedWhenManaged** resource is True, **XtRealizeWidget** maps the widget.

Note these implications:

- Geometry negotiation proceeds from the bottom up; then window creation proceeds from the top down.
- After a widget is realized, all its managed descendants are realized and, by default, mapped.
- If no widget in the tree is realized, all geometry negotiation between parents and their managed children takes place before any widget is realized.

When making a widget tree visible for the first time, an application should usually manage all children before realizing any widgets, then realize only the top–level widget. This causes all initial sizing and positioning of children to take place and the overall size of the top–level window to be determined before any windows exist, minimizing interaction with the X server. It also allows the application to realize all widgets with a single call to **XtRealizeWidget**.

Mapping Widgets

Most applications do not explicitly map or unmap widgets' windows. Mapping usually takes place as part of the process of managing or realizing widgets. But it is possible to keep Xt from mapping windows at these times by setting a widget's **XmNmappedWhenManaged** to False. In this case, the application must explicitly use **XtMapWidget** to map the widget. An application can use **XtUnmapWidget** to unmap a widget.

The effect of making a widget managed but unmapped is different from the effect of making a widget unmanaged. When a widget is unmanaged, its parent takes no account of it in laying out its children. When a widget is managed, its parent is likely to leave room for it in the widget layout. When the parent is mapped, the space allocated for a managed but unmapped child is filled with the parent's background rather than the child's window.

Multiple Screens, Displays, and Applications

An application can run on more than one display. In this case, it must use **XOpenDisplay** to open a connection to each display and must then call **XtDisplayInitialize** separately for each display connection. It need not create a separate application context for each display.

> **Note: XtDisplayInitialize** modifies its *argv* and *argc* arguments. If an application needs to call **XtDisplayInitialize** more than once, it must save these arguments before the first call and use a copy of the saved arguments on each call.

The application should use **XtAppCreateShell** to create at least one top–level widget for each display on which it runs. Because Xt maintains a separate resource database for each display, a child widget running on a different display from that of its parent would use incorrect initial resource settings.

An application can also run on more than one screen within a display. Such an application opens and initializes the display only once, no matter how many screens it uses within the display. However, the application also needs a widget on each screen, whose window is a child of the root window for that screen, to serve as the root of the widget hierarchy for the screen.

One approach to using multiple screens is to create a single, unrealized ApplicationShell for the display. The application then creates one TopLevelShell for each screen as a *popup* child of the ApplicationShell. Although a shell normally has only one managed child, it can have more than one popup child. The application uses **XtAppCreateShell** to create the ApplicationShell and **XtCreatePopupShell** to create each TopLevelShell. If no screen is specified for the ApplicationShell, **XtAppCreateShell** sets the **XmNscreen** resource for this widget to the default screen of the display. In the argument list passed to **XtCreatePopupShell**, the application must specify the proper value for **XmNscreen** for each TopLevelShell so that the shell is created on the intended screen.

The application does not manage the TopLevelShells. To realize and map the TopLevelShells, the program must use **XtPopup** with a *grab_kind* argument of **XtGrabNone**.

```
int main(int argc, char **argv)
{
 Widget         app_shell, top_shell;
 XtAppContext   app;
 Display        *display;
 char           name[20];
 Arg            args[5];
 Cardinal       n;
 int            i;

    app_shell = XtAppInitialize(&app, "Example",
        (XrmOptionDescList) NULL, 0, &argc, argv,
        (String *) NULL, (ArgList) NULL, 0);
    display = XtDisplay(app_shell);

    for (i = 0; i < ScreenCount(display); i++) {
        sprintf(name, "top_shell_%d", i);
        n = 0;
        XtSetArg(args[n], XmNscreen,
            ScreenOfDisplay(display, i));    n++;
        top_shell = XtCreatePopupShell(name,
                        topLevelShellWidgetClass, app_shell,
                        args, n);
        /* Create and manage descendants of top shell */
```

```
        ...
        /* Realize and map the top shell */
        XtPopup(top_shell, XtGrabNone);
    }
    ...
}
```

It is possible for a program to have multiple logical applications on the same display. In this case, it can use **XtAppCreateShell** to create a separate top–level widget for each logical application.

Entering the Event Loop

The last step in a Motif application is to enter the event loop. Most applications simply call **XtAppMainLoop**. This routine waits for user input and dispatches the resulting events to the appropriate event–handling procedures, usually in the widget in which the input occurs. **XtAppMainLoop** is an infinite loop; it never returns. An application should provide for a user action to terminate the program and should exit as a result of that action, usually in a callback routine.

Writing Threaded Applications

In general, writing multithreaded Motif applications requires familiarity with:

- General concurrent programming principles.
- Specific threads interfaces.
- New interfaces in X11R6 Xt for multithreaded applications.

Why Multithreading?

In general, there are two ways of specifying program concurrency: through two or more processes or through two or more threads. The former approach is known as multiprocessing and the latter approach is known as multithreading. There are two main reasons for choosing multithreading over multiprocessing: multiprocessing requires expensive interprocess communication, while separate threads can directly share data. Compared to process creation and process context switches, thread creation and thread context switches are inexpensive operations.

Input Processing Loop

Motif applications are based on a paradigm whereby the application operates in an infinite loop, checking for input and dispatching the input to the appropriate place. The input consists of X events arriving from the display server, input from alternate input sources, and timeout values. The code for the input processing loop looks something like this:

```
while(TRUE)     {
        XEvent event;
        XtAppNextEvent(app, &event);
        XtDispatchEvent(&event);
}
```

The *app* refers to an application context in the Motif application. **XtAppNextEvent** removes and returns the event from the head of *app*'s X event queue. If the X event queue is empty, **XtAppNextEvent** waits for an X event to arrive, meanwhile looking at alternate input sources and timeout values and calling any callback procedures triggered by them. After **XtAppNextEvent** returns, **XtDispatchEvent** dispatches the X event to the appropriate place. Dispatching the X event typically involves invoking some event handlers, action procedures, or callback procedures. For the purposes of our discussion, we will henceforth refer to event handlers, action procedures, callback procedures, input callback procedures and timer callback procedures, collectively as "callbacks."

Typically, every Motif application does its useful work in its "callbacks." If the work done in a callback" is time consuming (and indivisible), the processing of the next event on the X event queue may be noticeably delayed, causing degradation in the responsiveness of the application. Multithreading can help avoid the delay in the processing of X events and thus alleviate the problem of poor responsiveness (or interactivity).

However, this is not the only motivation. Others include a more "natural" program structure for inherently concurrent Motif applications and better performance of Motif applications on multiple−processor machines (or on a configuration of multiple machines).

Xt Interfaces For Multithreading

Motif is so closely coupled with the X Toolkit. The X11R6 interfaces for multithreading are required only if the multithreaded application calls Motif/Xt interfaces in multiple threads. It is possible to write multithreaded Motif applications in which Motif/Xt interfaces are invoked only from a single thread. In such applications, the interfaces may not be required. In addition to these interfaces, X11R6 declares functions and data types used by multithreaded programs, in the file `X11/Xthreads.h`. Since threads interfaces are operating system dependent, including this file instead of the system specific file increases portability.

Initializing Xt For Use In Multiple Threads

A Motif application that creates multiple application threads must call **XtToolkitThreadInitialize**, which initializes Xt for use in multiple threads. **XtToolkitThreadInitialize** returns *True* if the given Xt supports multithreading, otherwise it returns *False*.

XtToolkitThreadInitialize may be called before or after **XtToolkitInitialize** and it may be called more than once. However, it must not be called concurrently from multiple threads. An application must call **XtToolkitThreadInitialize** before calling **XtAppInitialize**, **XtSetLanguageProc**, **XtOpenApplication** or **XtCreateApplicationContext**.

Using XtAppLock and XtAppUnlock

Concurrency can be programmed into a Motif application by using a model where each *AppContext* has one thread. This ensures that each event loop within each *AppContext* can operate independently in its own thread. This is important in order to keep the data and event processing in each *AppContext* from becoming corrupted. Enabling only one thread in each *AppContext* requires a locking strategy per *AppContext*.

To lock and unlock an *AppContext* and all widgets and display connections in the *AppContext*, an application must use **XtAppLock** and **XtAppUnlock**.

All Motif and Xt functions that take an *AppContext*, widget or display connection as a parameter, implicitly lock their associated *AppContext* for the duration of the function call. Therefore, with a few exceptions, an application does not need to call **XtAppLock** or **XtAppUnlock**. The first exception is the situation in which an application needs to make a series of Xt function calls atomically. In such a situation, an application can enclose the series of Xt calls within a matching pair of **XtAppLock** and **XtAppUnlock**. For example, if an application wants to check and update the height of a widget atomically, it might do it as follows:

```
XtAppContext app;
Dimension ht = 0;
Widget w;
...
XtAppLock(app);
XtVaGetValues(w, XtNheight, &ht, NULL);
if((int)ht < 10) {
        ht += 10;
        XtVaSetValues(w, XtNheight, ht, NULL);
}
XtAppUnlock(app);
```

Other exceptions are the Motif resources that are returned as live pointers to the actual resource storage within the widget instance. Some examples include:

- **XmNchildren**: returns a pointer to the corresponding Composite field XmNitems
- **XmNselectedItems**: returns pointers to the corresponding List fields
- **XmNsource**: returns a pointer to the corresponding Text field

In these cases, the application again needs to envelop the **XtGetValues** call within *AppLocks* so that another thread operating on the same instance doesn't change the resource value. The application might also want to make a copy of the retrieved resource.

To provide mutually exclusive access to global data structures application writers can use **XtProcessLock** and **XtProcessUnlock**.

Both **XtAppLock** and **XtProcessLock** may be called recursively. To unlock to the top−level, **XtAppUnlock** and **XtProcessUnlock** must be called the same number of times as **XtAppLock** and **XtProcessLock**, respectively. To lock an *AppContext* and Xt's global data at the same time, first call **XtAppLock** and then **XtProcessLock**. To unlock, first call **XtProcessUnlock** and then **XtAppLock**. The order is important to avoid deadlock.

New XtAppMainLoop

As we have seen, the R5 **XtAppMainLoop** is an infinite loop that calls **XtAppNextEvent** and then **XtDispatchEvent**. In a multithreaded−safe Xt, an infinite **XtAppMainLoop** would prevent an input processing thread from exiting its **XtAppMainLoop**, without simultaneously exiting its thread. This situation may lead to the following problems:

- Memory leaks.
- Dangling threads.
- Complicated synchronization between the input processing thread and other threads.

In order to elegantly address these problems, the X11R6 version of Xt reimplemented **XtAppMainLoop**. It is still a conditional loop that calls **XtAppNextEvent** and then **XtDispatchEvent**, but it now checks at the bottom of the input loop to determine if the application is finished with the *AppContext* by checking a new exit flag. If the exit flag is set to *True*, **XtAppMainLoop** exits the input processing loop. The following example shows one implementation of the new **XtAppMainLoop**.

```
XtAppContext app;
do {
        XEvent event;
        XtAppNextEvent(app, &event);
```

```
        XtDispatchEvent(&event);
} while(XtAppGetExitFlag(app) == FALSE);
```

The *AppContext*'s exit flag can be set to *True* by calling **XtAppSetExitFlag**. The value of the exit flag can be obtained through **XtAppGetExitFlag**. The new **XtAppMainLoop** offers an effective way of destroying an *AppContext*, without having to exit from the input processing thread.

Destroying An Application Context

In multithreaded Motif applications, the recommended way of destroying an *AppContext* is: After the input processing thread returns from **XtAppMainLoop**, and after it has been determined that no other thread is referring to the associated *AppContext*, call **XtDestroyApplicationContext**. Use **XtAppGetExitFlag** to synchronize between an input processing thread and other threads. Call **XtAppSetExitFlag** in the exit callback. This causes the associated **XtAppMainLoop** to exit but does not terminate the input thread.

Event Management In Multiple Threads

When multiple threads call into event management functions concurrently, they return from these functions in last−in first−out order. Another way of looking at this is to imagine that if multiple threads get blocked for input in any of the event management functions, they are pushed on a per−application−context stack of blocked threads. Whenever there is some input to process, a thread is popped off the stack and allowed to process the input.

For example, let's assume thread A, the "input processing thread", is blocked for input in **XtAppMainLoop** so that it is on the per−application −context stack of blocked threads. Let us further assume that subsequently one of the "other" threads, thread B, detects some error condition and decides to popup a message box and temporarily take over input processing from thread A. The fact that blocked threads get stacked allows thread B to easily take over input processing from thread A. The way thread B accomplishes this is to execute a form of input processing loop within a matched pair of **XtAppLock** and **XtAppUnlock**, as follows:

```
/* This code is in a thread other than the input processing thread */
XtAppContext app;
Widget error_dialog;
XEvent event;
XtAppLock(app); /* Lock the AppContext */
XtPopup(error_dialog, XtGrabExclusive); /* popup error dialog */
do {/* Take over input processing */
        XtAppNextEvent(app, &event);
        if(/* some boolean condition involving event */)
                XtDispatchEvent(&event);
} while(/* some boolean condition */)
XtAppUnlock(app);
```

Because thread B calls into **XtAppNextEvent** after thread A does, thread B gets stacked above thread A. Subsequently, whenever input arrives, thread B is "popped" off the stack and allowed to process the input.