

X/MOTIF

Amarra, Anna Christine
Dueñas, Jose Roberto
Lucena, Joshua Paolo
Nieva, Jeanie
Sevilla, Anna Mae
Vy, Sally

CS 159.3 A
Mr. William Yu

X/MOTIF

In this era, people wanted everything to be done at the shortest possible time and at the lightest possible work. And when it comes to computers, users wanted to execute processes with the most convenient way. With this, the graphical user interface (GUI) was developed. GUIs provide an easy means of data entry and modification.

One of the GUI toolkits available today is the Motif.

Motif was the first graphical interface offering user-oriented PC-style behavior and screen appearance for applications running on systems that support the X Window System X11R5. It is also the base graphical user interface toolkit for the Common Desktop Environment (CDE); CDE is built on top of Motif. It provides a high level GUI toolkit that adopts common GUI design principles. It has many fully featured GUI objects. For example cut and paste, multi-line text editors, file browsers, drag and drop mechanisms. Motif applications can be assembled by bringing such objects together, thus speeds up GUI program development. This same toolkit enables the development of cross-platform applications, thus helps in protecting valuable investments in software and user training.

HISTORY

We all know that computers became commercially available in the 1950s, but they are unusually large, expensive and at the same time difficult to operate. After years of development, more reliable operating systems were developed, which resulted in smaller and smaller workstations. In these computers, users were only able to interact through forms of early text editors. Then in the early 1970s, Alan Kay's research group at Xerox's Palo Alto Research Center, where two important projects were undertaken: the development of the book-sized personal computer with high resolution color display, a radio link to a worldwide computer network and the inclusion of mailbox, library, telephone and secretarial functions (*Dynabook*) and the production of desk-sized personal workstation used by a single person (*Star*). It was in *Star* where the idea of a graphical user interface came about. From here, Apple developed *Lisa*, which supports GUIs, then followed by other operating systems. But these developments are not able to communicate with other operating systems other than the developers' since every manufacturer had its own proprietary windowing system.

With this problem at hand, the X Window System was developed. The X System was designed such that it is platform independent and network-based. This means that with X, an application written in a single language can be run on different machines with little or no modification needed.

Following the creation of the X Window system, two primary high-level X interface toolkits were developed. These are the Motif, which is a product of the Open Software Foundation (OSF), an organization that originally included DEC, IBM and Hewlett-Packard; and the Open Look/Open Windows, a product of Sun and AT&T. And since Motif was based on IBM's Common User Access (CUA) guidelines, the same guidelines used by Microsoft Windows and OS/2 (known for its user-friendly environment), it is the logical choice of users in migrating applications to UNIX. It was intentionally modeled after CUA specification because there is a proven business model for profiting from an "open systems" philosophy.

The Sun was aware of the booming business with Motif. It then decided to stop developing Open Look and adopted Motif for Sun Workstations instead. With this, the Common Open Software Environment (COSE) united major UNIX producers including Sun, DEC, IBM, Hewlett-Packard and UNIX Systems Laboratories. In effect, there was a significant impact on the endorsement of Motif since it became a standard choice for UNIX and general cross-platform GUI development.

To better achieve the purpose of the creation of Motif, it has undergone five major revisions: Motif 1.0, Motif 1.1, Motif 1.2, Motif 2.0, and Motif 2.1. The newer the version, the more efficient it is, as detected bugs from the older versions were fixed and significant enhancements were done.

As mentioned, the OSF/Motif was developed by Open Software Foundation, for which it got the OSF of its name. It is popularly known as X/Motif since Motif is an important component of the X Windows System. It was developed with the goal of enhancing inter-operability between computers from different manufacturers. Still, it remains as the common native windowing toolkit for all the UNIX platforms fully supported by all the major operating system vendors. Its graphical user interface specification is designed to be independent of the computer on which the application is running.

See the concepts section of this paper for further discussions on Motif specification.

MOTIF PROGRAMMING PRINCIPLES

Even though sometimes, programmers are free to create their program in their own standards, Motif has several programming principles. The most basic ones are as follows:

1. Initializing the toolkit.
2. Widget creation
3. Managing widgets
4. Setting up events and callback functions
5. Displaying the widget hierarchy
6. Enter the main event handling loop

There are several ways to initialize the toolkit. `XtVaOpenApplication()` is one common method. For most of our programs this is the only one that need concern us.

When the `XtVaOpenApplication()` function is called, the following tasks are performed:

- The application is connected to the X display.
- The application is parsed for the standard X command-line arguments.
- Resources are set up.
- A top level window / shell, which handles the application's interaction with the window manager, is created

`XtVaOpenApplication()` has several arguments:

- Application context
- Application class name
- Command line arguments
- Fallback Resources
- Additional Parameters

Initializing a toolkit:

```
Widget toplevel;  
XtAppContext app;
```

```
toplevel = XtVaOpenApplication (&app, name", NULL, 0, &argc,  
argv, NULL, sessionShellWidgetClass, NULL);
```

After which, we need to create the widgets. The basic unit of Motif is the *widget*, the basic building block for the GUI. It is common and beneficial for most GUIs assembled in Motif to look and behave in a similar fashion. Since it is more of a specification, Motif enforces many of these features by

providing default actions for each widget. Motif also prescribes certain other actions that should, whenever possible, be adhered to.¹

And with the many software developed now a days, Motif can be created using Motif Builders. These builders helps the Motif developer create GUIs. It makes the programming task easier since the programmer will just have to drag and drop the widgets needed, just like Visual Café for Java and Visual Basic. The builder also helps in defining the events of the widgets created. The main purpose of these builders is to ease the programming burden, especially to those new to the Motif environment. A list of available builders on sale, their respective vendors and where to get them is provided in Appendix B.

PRIMARY WIDGETS and GADGETS

A widget in Motif may be regarded as a general abstraction for user-interface components. Basically, each widget is defined as a C data structure whose elements define a widget's data attributes, or *resources* and pointers to functions, such as *callbacks*. So, each widget is defined to be of a certain class, and Motif has its own definition of a whole hierarchy of widget classes.

To use the widgets, we need to explicitly call Motif functions for the specific widget, which are found in `#include<Xm/...>` subdirectories. Header files containing definitions of the widget to be used must be included. We must always include the general header of the Motif library.

There are times when we want to put multiple primitive widgets of the same type, or use the properties of the primitive widget but do not want to worry about managing its window properties. Because of the complication, Motif's developers decided to create *gadgets* to try to lessen these types of problems.

Gadgets are basically windowless widgets and, therefore, require lesser resources than a widget. There are gadgets that are equivalent to many of the primitive widgets: ArrowButtonGadgets, SeparatorGadgets, PushButtonGadgets, CascadeButtonGadgets, ToggleButtonGadgets, LabelGadgets, and in Motif 2.1, IconGadgets. The IconGadget is similar to a LabelGadget, except that it can display a label and an image simultaneously. The appearance and behavior of the gadgets are mostly identical to that of the corresponding widgets. Control of the gadget is the responsibility of the parent of the gadget.

The following are some of the widely used widgets, which mostly have corresponding gadgets:

LABELS

Simply props for the stage, labels are not intended to respond to user interaction, although a help callback can be attached in case the HELP key is pressed. It is not uncommon to find Labels (fig.1.2) displaying either text or graphics, yet they cannot display both simultaneously in the conventional sense. It is not always obvious whether to use a Label or a Text widget since the former can display text.

All of the button subclasses of Label inherit the drag source capability, so the text labels for PushButtons and ToggleButtons can also be manipulated using drag and drop.

The only callback routine for the Label widget is the `XmNhelpCallback` associated with all Primitive widgets. If the user presses the HELP key on a Label widget, its help callback is called.

To use Labels, one must include the header file `<Xm/Label.h>`, which defines the `xmLabelWidgetClass` type. This type is a pointer to the actual widget structure used by `XtVaCreateWidget()` or `XtVaCreateManagedWidget()` routines.

¹ http://www.cs.cf.ac.uk/Dave/X_lecture/X_book_caller/X_book_caller.html#ch:motif2.0

A Label widget or gadget can display either text or an image. The `XmNlabelType` resource controls the type of label that is displayed; the resource can be set to `XmSTRING` or `XmPIXMAP`. If you want to display text in a Label, you do not need to set this resource explicitly as the default value is `XmSTRING`. The resource that specifies the string that is displayed in a Label is `XmNlabelString`. The value for this resource must be a Motif compound string; common C character strings are not allowed.

If the `XmNlabelString` resource is not specified, the Label automatically converts its name into a compound string and uses that as its label.

Technically, widget names should only be composed of alphanumerics (letters and numbers), hyphens, and underscores. Characters such as space, dot (`.`), and the asterisk (`*`) are disallowed because they make it impossible for the user to specify these widgets in resource files.

By setting the `XmNlabelType` resource to `XmPIXMAP`, a Label widget or gadget can display an image instead of text. As a result of this resource setting, the Label displays the pixmap specified for the `XmNlabelPixmap` resource.

A Label can be made inactive by setting the `XmNsensitive` resource to `False`. Since Labels are never really active, it may seem frivolous to set a Label insensitive common to associate a Label with another interactive element, such as a List, a TextField, or even a composite item such as RadioBox. In these situations, it is useful to desensitize the Label along with its corresponding user-interface element, to emphasise that the component is inactive. In the same vein, if `XtSetSensitive()` is applied to a Manager widget, the routine sensitizes or desensitizes all of the children of the widget, including Labels.

If a Label is displaying text, setting the widget insensitive causes the text to be greyed out. This effect is achieved by stippling the text label. If a Label is displaying an image, you can specify the `XmNlabelInsensitivePixmap` resource to indicate the image that is displayed when the Label is inactive. By default, the resource is set to the value `XmUNSPECIFIED_PIXMAP`, and the Label will use the `XmNlabelPixmap` resource value, automatically applying an opaque stippling mask operation on the pixmap image concerned.

Within the boundaries of a Label widget or gadget, the text or image that is displayed can be left justified, right justified, or centered. The alignment depends on the value of the `XmNalignment` resource, which can have one of the following values: `XmALIGNMENT_BEGINNING`, `XmALIGNMENT_END`, `XmALIGNMENT_CENTER`. The default value is `XmALIGNMENT_CENTER`, which causes the text or pixmap to be centered vertically and horizontally within the widget or gadget. The `XmALIGNMENT_BEGINNING` and `XmALIGNMENT_END` values refer to the left and right edges of the widget or gadget when the value for `XmNlayoutDirection` is set to `XmLEFT_TO_RIGHT`.

If the text used within a Label is read from left-to-right (the default), the beginning of the string is on the left. However, if the text used is read from right-to-left, the alignment values are inverted, as should be the value for `XmNlayoutDirection`.

Multi-line and multi-font labels are also possible. The fonts used within a Label are directly associated with the rendition element tags used in the compound string specified for the `XmNlabelString` resource. The `XmNrenderTable*` resource for a Label specifies the mapping between rendition tags and fonts that are used when displaying the text. Since a compound string may use multiple character sets, a Label can display any number of fonts, as specified in the `XmNlabelString` for the Label. A compound string may also contain embedded newlines and tabs, and color specifications.

To create a simple label:

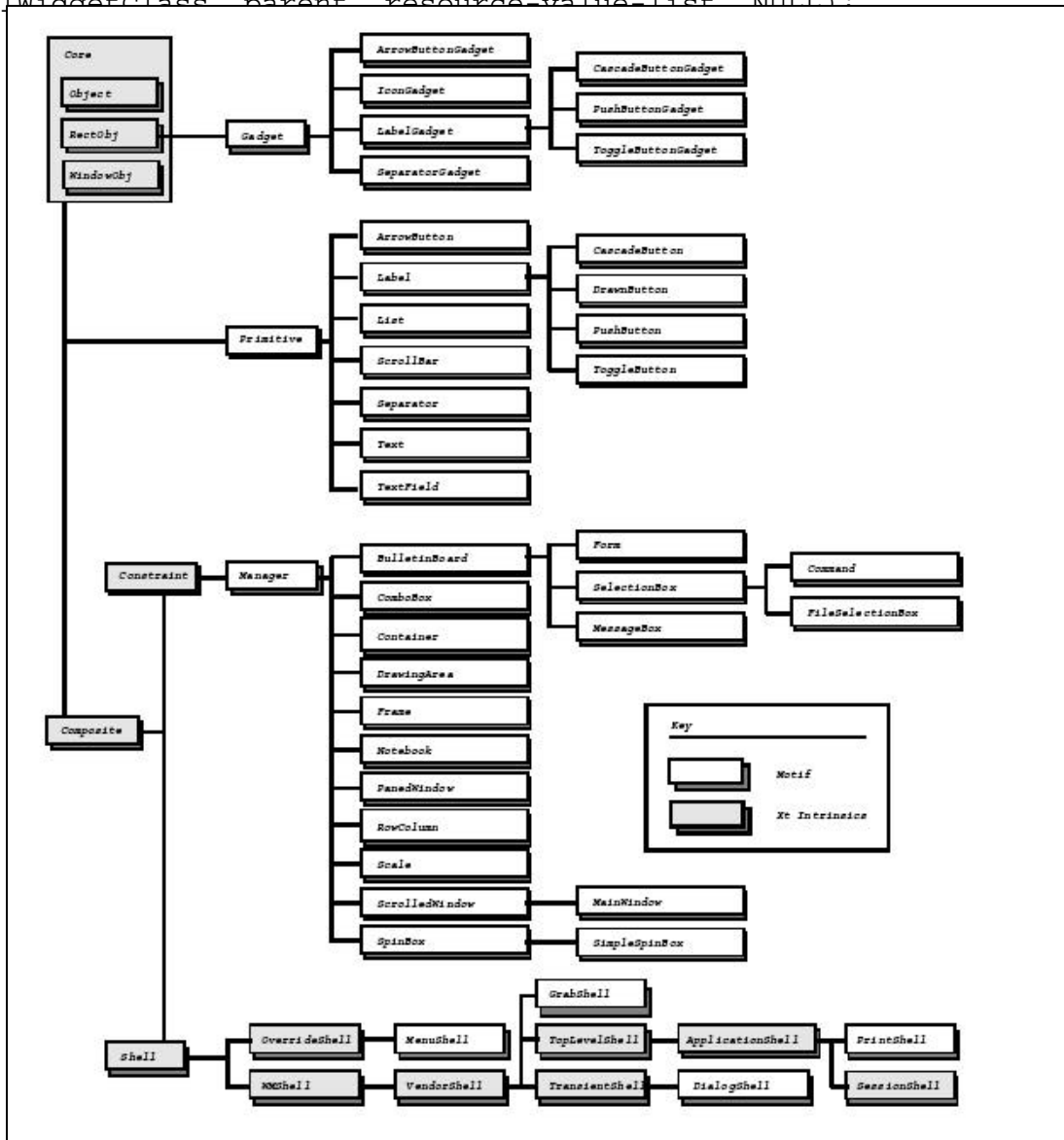
```

Widget label = XmCreateLabel (parent, "name", resource-value-
array, resource-value-count);
...
XtManageChild (label);

Widget label = XtVaCreateWidget ("name", xmLabelWidgetClass,
parent, resource-value-list, NULL);
...
XtManageChild (label);

Widget label = XtVaCreateManagedWidget ("name",
xmLabelWidgetClass, parent, resource-value-list, NULL);

```



More specifically, text labels can be created through the following:

```
Widget label;
```

```

Arg args[...];
int n= 0;

XmString str = XmStringCreateLocalized ("A Label");

XtSetArg (args[n], XmNlabelString, str); n++;
label = XmCreateLabel (parent, "label", args, n);
XmStringFree (str);

```

If the `XmNlabelString` resource is not specified, the Label automatically converts its name into a compound string and uses that as its label. Therefore, the previous example could also be implemented as follows:

```

Widget label = XmCreateLabel (parent, "A Label", NULL, 0);

```



Fig. 1.2 A Label Widget

SCROLLBARS

The `ScrolledWindow` widget provides a viewing area into another, usually larger, visual object. The viewport may be adjusted by the user through the use of `ScrollBars` (fig. 1.3) that are attached to the `ScrolledWindow`. The `Motif MainWindow`, `ScrolledList`, and `ScrolledText` objects use `ScrolledWindows` to implement scrolling for their respective contents. The `ScrolledWindow` can also be used independently to provide a viewport into another large object, such as a `DrawingArea` or a manager widget that contains a large group of widgets.

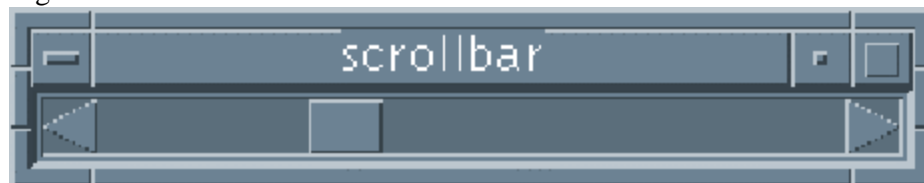


Fig. 1.3 A Scrollbar Widget

The user always interacts with a `ScrolledWindow` through `ScrollBars`. Internally, however, there are several ways to implement what the user sees. These methods are based on two different scrolling models: automatic scrolling and application-defined scrolling. In either case, the application gives the `ScrolledWindow` a *work window* that contains the visual data to be viewed. Although the two models are different, they share many of the same concepts and features.

In automatic scrolling mode, the `ScrolledWindow` operates entirely on its own, adjusting the viewport as necessary in response to `ScrollBar` activity. The application simply creates the desired data, such as a `Label` widget that contains a large pixmap, and makes that widget the work window for the

ScrolledWindow. When the user operates the ScrollBars to change the visible area, the ScrolledWindow adjusts the Label so that the appropriate portion is visible.

With application-defined scrolling, the ScrolledWindow operates under the assumption that the work window is not complete. The widget assumes that another entity, application or the internals of another widget, controls the data within the work window and that the data may change dynamically as the user scrolls. In order to control scrolling, the application must control all aspects of the ScrollBars. This level of control is necessary when it is impossible or impractical for an application to provide the ScrolledWindow with a sufficiently large work window (or the data for it) at any one time.

Most of the time, the ScrolledWindow widget is used in automatic scrolling mode. When it is used in this mode, the ScrolledWindow contains at most three internal widgets: two ScrollBars and a *clip window*. The ScrolledWindow creates these widgets automatically. The work area is an external widget (specified by the `XmNworkWindow` resource) that is clipped by the clip window. This work window is a child of the ScrolledWindow that is provided by the application; it is not created automatically by the ScrolledWindow. When the user interacts with the ScrollBars, the work window is adjusted so that the appropriate part is visible through the clip window.

The work window can be almost any widget, but there can be only one work window per ScrolledWindow. If you want to have more than one widget inside of a ScrolledWindow, you can place all of the widgets in a manager widget and make that manager the work window. The clip window is always the size of the viewport portion of the ScrolledWindow, which is the size of the ScrolledWindow minus the size of the ScrollBars and any borders and margins. The clip window is not adjusted in size unless the ScrolledWindow is resized. The clip window is always positioned at the origin, which means that you cannot use `XtMoveWidget()` or change its `XmNx` and `XmNy` resources to reposition it in the ScrolledWindow. The internals of the ScrolledWindow are solely responsible for changing the view in the clip window, although you can affect this behavior. While you can get a handle to the clip window, you must not remove it or replace it with another window.

To be able to use the scrollbar widget, we should include `<Xm/ScrollBar.h>`. And to instantiate this widget:

```
Widget scrollbar = XmCreateScrollBar (parent, name, resource-
value-array, resource-value-count);
or
Widget scrollbar = XtCreateWidget (name, xmScrollBarWidgetClass,
parent, resource-value-array, resource-value-count);
```

TOGGLEBUTTONS

A ToggleButton (Fig. 1.4) is a simple user-interface element that represents application state in some way, usually a Boolean value. The widget consists of an indicator (a square, diamond, or circle) with either text or a pixmap on one side of it*. The indicator is optional, however, since the text or pixmap itself can provide the state information of the button. The ToggleButton widget is subclassed from Label, so ToggleButtons can have their labels set to compound strings or pixmaps and can be aligned in the same ways and under the same restrictions as Label widgets.

Individually, a ToggleButton might be used to indicate whether a file should be opened in overwrite mode or append mode, or whether a mail application should update a folder upon process termination. But for the most part, it is when ToggleButtons are grouped together that they become interesting components of a user interface. A RadioBox is a group of ToggleButtons in which only one may be on at any given time. Like the old AM car radios, when one button is pressed in, all of the others are popped out. A CheckBox is a group of ToggleButtons in which each ToggleButton may be set independently of the others. In a RadioBox the selection indicator is represented by a diamond shape, and

in a CheckBox it is represented by a square. In either case, when the button is on, the indicator appears to be pressed in, and when it is off, the indicator appears to be popped out. The indicator can also be configured to internally display a cross or check (tick) mark†, and there are resources specifically to configure the color of the indicator in the on and off state‡.

A CheckBox or a RadioButton can often present a set of choices to the user more effectively than a List widget, a PopupMenu, or a row of PushButtons. In fact, these configurations are so common that Motif provides convenience routines for creating them: `XmCreateRadioButton()` and `XmCreateSimpleCheckBox()`. RadioBoxes and CheckBoxes are really specialized instances of the RowColumn manager widget that contain ToggleButton children.

Applications that use ToggleButtons must include the header file `<Xm/ToggleB.h>`. ToggleButtons may be created using the following code fragment:

```
Widget      toggle      =      XtVaCreateWidget      ("name",  
xmToggleButtonWidgetClass,parent, resource-value-list, NULL);
```

The value `XmONE_OF_MANY` resource can result in either a round or diamond shape, depending upon the XmDisplay object `XmNenableToggleVisual` resource. If this is `True`, then the result is round, otherwise a diamond. The diamond shape is consistent with a Motif 1.2 appearance.

When grouping ToggleButtons together in a single manager widget, the Motif toolkit expects you to use a RowColumn widget. The RowColumn widget has several resources intrinsic to its class that control the behavior of ToggleButton children. Setting the RowColumn resource `XmNradioBehavior` to `True` automatically changes the `XmNindicatorType` resource of every ToggleButton managed by the RowColumn to `XmONE_OF_MANY`, which provides the exclusive RadioButton behavior. Setting `XmNradioBehavior` to `False` sets the `XmNindicatorType` to `XmN_OF_MANY` and gives the CheckBox behavior. If you want to use ToggleButtons in a manager widget other than a RowColumn, you need to set the `XmNindicatorType` resource for each ToggleButton individually, as well as manage the state of each button.

MENUS

Menus provide the user with a set of choices in an application without complicating its normal visual appearance. These convenient mini-toolboxes are essential for the user who, like an auto mechanic that is busy working under the car, needs quick and convenient access to her tools without having to look or move away from her work. The *Motif Style Guide* provides for three different types of menus: PulldownMenus, PopupMenus, and OptionMenus. Despite the differences between the three types of menus, they all provide simple and convenient access to application functionality.



Fig. 1.4 ToggleButton Widget

Menu Types


```
Widget pushb_w = XmCreatePushButton (parent, "name", resource-
value-array, resource-value-count);
```

```
Widget pushb_g = XmCreatePushButtonGadget (parent, "name",
resource-value-array, resource-value-count);
```

The major callback routine associated with the `PushButton` widget is the `XmNactivateCallback`. The functions associated with this resource are called whenever the user activates the `PushButton` by pressing the left mouse button over it or by pressing the `SPACEBAR` when the widget has the keyboard focus. The other callback routines associated with the `PushButton` are the `XmNarmCallback` and the `XmNdisarmCallback`. Each function in an arm callback list is called whenever the user presses the left mouse button when the pointer is over the `PushButton`. When the `PushButton` is armed, the top and bottom shadows are inverted and the background of the button changes to the arm color. The arm callback does not indicate that the button has been released. If the user releases the mouse button within the widget, then the activate callback list is invoked. The arm callback is always called before the activate callback, whether or not the activate callback is even called.

When the user releases the button, the disarm callback list is invoked. When the button is disarmed, its shadow colors and the background return to their normal state. Like the arm callback, the disarm callback does not guarantee that the activate callback has been invoked. If the user changes her mind before releasing the mouse button, she can move the mouse outside of the widget area and then release the button. In this case, only the arm and disarm callbacks are called. However, the most common case is that the user actually selects and activates the button, in which case the arm callback is called first, followed by the activate callback and then the disarm callback.

The activate callback function is by far the most useful of the `PushButton` callbacks. It is generally unnecessary to register arm and disarm callback functions, unless your application has a specific need to know when the button is pushed and released, even if it is not activated.

```
/* pushb.c 2-- demonstrate the pushbutton widget. Display one
** PushButton with a single callback routine. Print the name
** of the widget and the number of "multiple clicks". This
** value is maintained by the toolkit.
*/

#include <Xm/PushB.h>
main (int argc, char *argv[])
{
    XtAppContext app;
    Widget toplevel, button;
    void my_callback(Widget, XtPointer, XtPointer);
    XmString btn_text;
    Arg args[2];

    XtSetLanguageProc (NULL, NULL, NULL);
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0,
&argc, argv,
                                NULL, sessionShellWidgetClass, NULL);
```

² Anthony Fountain, et al. *Motif Programming Manual*. United Kingdom: O'Reilly & Associates, Inc., 2001. p.403.

```

    btn_text = XmStringCreateLocalized ("Push Here");
    XtSetArg (args[0], XmNlabelString, btn_text);
    button = XmCreatePushButton (toplevel, "button", args, 1);
    XmStringFree (btn_text);
    XtAddCallback (button, XmNarmCallback, my_callback, NULL);
    XtAddCallback (button, XmNactivateCallback, my_callback,
NULL);
    XtAddCallback (button, XmNdisarmCallback, my_callback,
NULL);
    XtManageChild (button);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

void my_callback (Widget w, XtPointer client_data, XtPointer
call_data)
{
    XmPushButtonCallbackStruct *cbs =
(XmPushButtonCallbackStruct *) call_data;
    if (cbs->reason == XmCR_ARM)
        printf ("%s: armed\n", XtName (w));
    else if (cbs->reason == XmCR_DISARM)
        printf ("%s: disarmed\n", XtName (w));
    else
        printf ("%s: pushed %d times\n", XtName (w), cbs-
>click_count);
}

```

The callback structure associated with the PushButton callback routines is `XmPushButtonCallbackStruct`, which is defined as follows:

```

typedef struct {
    int reason;
    XEvent *event;
    int click_count;
} XmPushButtonCallbackStruct;

```

The reason parameter is set to `XmCR_ACTIVATE`, `XmCR_ARM`, or `XmCR_DISARM` depending on the callback that invoked the callback routine. We use this value to decide what action to take in the callback routine. The event that caused the callback routine to be invoked is referenced by the `event` field. The value of the `click_count` field reflects how many times the PushButton has been clicked repeatedly. A repeated button click is one that occurs during a predefined time segment since the last button click. Repeated button clicks can only be done using the mouse. The time segment that determines whether a button click is repeated is defined by the resource `multiClickTime`. This resource is not defined in the widget class hierarchy but on a per-display basis; the value should be left to the user to specify independently from the application. You can get or set this value using the functions `XtGetMultiClickTime()` or `XtSetMultiClickTime()`.

TEXT and TEXTFIELD

These widgets can be used for a variety of purposes, from a simple data-entry field to a full-fledged text editor. The chapter describes the selection mechanisms provided by the widgets and how they can be used to communicate with other applications via the clipboard. The widgets also allow the programmer to control the format of the data that is entered by the user.

Despite all that can be done with menus, and buttons, there are times when the user can best interact with an application by typing at the keyboard. The Text widget (fig. 1.6) is usually the best choice for providing this style of interface, as it provides full-featured text editing capabilities. The Text widget can be used anywhere the user might be expected to type free-form text, such as in a compose window in a mail application. Unlike standard text editors, the Text widget supports the point-and-click model that people expect from GUI-based applications. The TextField widget (fig 1.7) provides a single-line data entry field with the same set of editing commands as the Text widget, but it requires less overhead. Text widgets can also be used in output-only mode to display more textual information than is practical with a label or a button.

Even though the text widgets allow for complex interaction, they still provide simple mechanisms for program control. The widgets have resources that access the text, as well as control their behavior. They also provide callback routines that allow an application to intervene on actions that add text, delete text, or move the insertion cursor. The widgets support keyboard management methods that control the editing style, and line-wrapping. There are also convenience routines that enable quick and simple access to the clipboard.

The text widgets do have their limitations. For example, they do not support multiple colors or fonts, so a single widget can only use one color and one font. There is no support for text formatting such as paragraph specifications, automatic line numbering, or indentation.

Implementation:

In order to understand the complexities of the Text and TextField widgets, it is important to know about some of the basic resources and functions that they provide. This following describes the fundamentals of working with text widgets, including how to create the widgets, how to work with the textual data, and how to control simple aspects of appearance and behavior. Applications that wish to use the Text widget need to include the file `<Xm/Text.h>`. TextField widgets require the file `<Xm/TextF.h>`. One can create a Text widget using the following methods:

```
Widget      text_w      =      XtVaCreateWidget      ("name",
xmTextWidgetClass, parent, resource-value-list, NULL);
Widget text_w = XmCreateText (parent, "name", resource-
value-array, resource-value-count);
```

To create a TextField widget instead, either specify the class as `xmTextFieldWidgetClass` in the `XtVaCreateWidget()` call, or use the Motif convenience routine `XmCreateTextField()`.

Textual Data:

The `XmNvalue` resource of the Text and TextField widgets provides the most basic means of access to the internal text storage for the widgets. Unlike the other widgets in the Motif toolkit that use text, the text widgets do not use compound strings for their values. Instead, the value is specified as a regular C string.

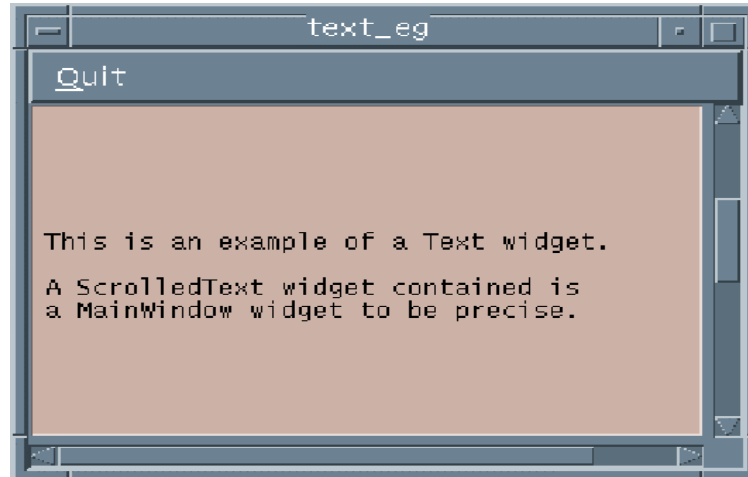


Fig. 1.6 Text Widget

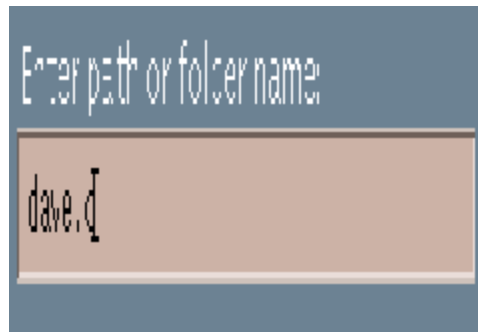


Fig. 1.7 TextField Widget

Aside from the above mentioned primitive widgets, there are still many other widgets available in Motif library.

MANAGER WIDGETS

After creating the widgets needed for the application design, we need *manager widgets* to manage other widgets. These control the size and location (geometry) and input focus policy for one or more widget children. The relationship between managers and the widgets that they manage is commonly referred to as the parent-child model. The manager acts as the parent and the other widgets are its children. Since manager widgets can also be children of other managers, this model produces the widget hierarchy, which is a framework for how widgets are laid out visually on the screen and how resources are specified in the resource database.

Since a manager is a widget that usually contains children, either primitives or other managers, it has to take note of several responsibilities. One responsibility of a manager is to position and shape its children so that the configuration of the children is appropriate for the manager's specialized purpose. Another responsibility is to determine whether a gadget child should process an input event and, if so, to dispatch the event to that child.

The **XmManager** Motif widget class is the superclass for all managers. **XmManager** is a subclass of **Core**. Like **XmPrimitive**, **XmManager** has resources to control colors or pixmaps used for

the foreground, shadows, and highlighting rectangle. Most managers do not have shadows or highlighting rectangles, but gadget children inherit the related resources. Managers also have resources that control keyboard traversal, and they provide callbacks for processing user requests for help. In addition, they have translations and actions for dispatching input events to gadget children, usually to the child that is the current focus of keyboard events.

Because the Manager widget class is a metaclass for a number of functional subclasses, the Manager widget class is never instantiated; the functionality it provides is inherited by each of its subclasses.

FRAME CLASS

A Frame (fig. 1.8) is a simple manager that surrounds a single child with a shadow and a margin. Its purpose is to provide a visible, three-dimensional border for objects such as RowColumns or Labels that do not provide a border. It can also be used to enhance the style of the border for a widget that already has a border. The Frame widget may have two children: a work area child and a label child. The Frame sizes itself just big enough to contain its children. A Frame can also have another child that appears as a title for the Frame.

To create :

```
widget = XmCreateFrame (parent, name, resource-value-array,
resource-value-count);
or
widget = XtCreateWidget (name, xmFrameWidgetClass, parent,
resource-value-array, resource-value-count);
```

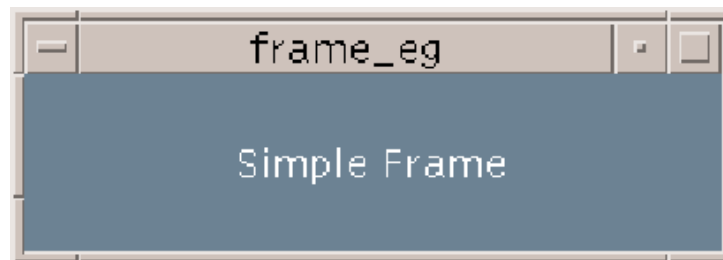


Fig. 1.8 Frame Widget

SCALE CLASS

A Scale (fig. 1.10) is a manager that functions as a control. It displays a value within a range and optionally allows the user to supply a new value by moving the slider. Its appearance and behavior are much like those of a ScrollBar without arrows. It also has a title and can display the current value next to the slider. If the application adds other children to a Scale, the Scale positions them evenly along the rectangular area that represents the range of values, and these children then act as tick marks or value labels. In Motif 1.2, the only sensible children that you could add to a Scale were Label widgets that represent tick marks, and these would have to be laid out by the programmer. However, in Motif 2.0, the function `XmScaleSetTicks()` was introduced which automatically places marks at calculated positions along the Scale edge.



Fig. 1.10 Scale Widget

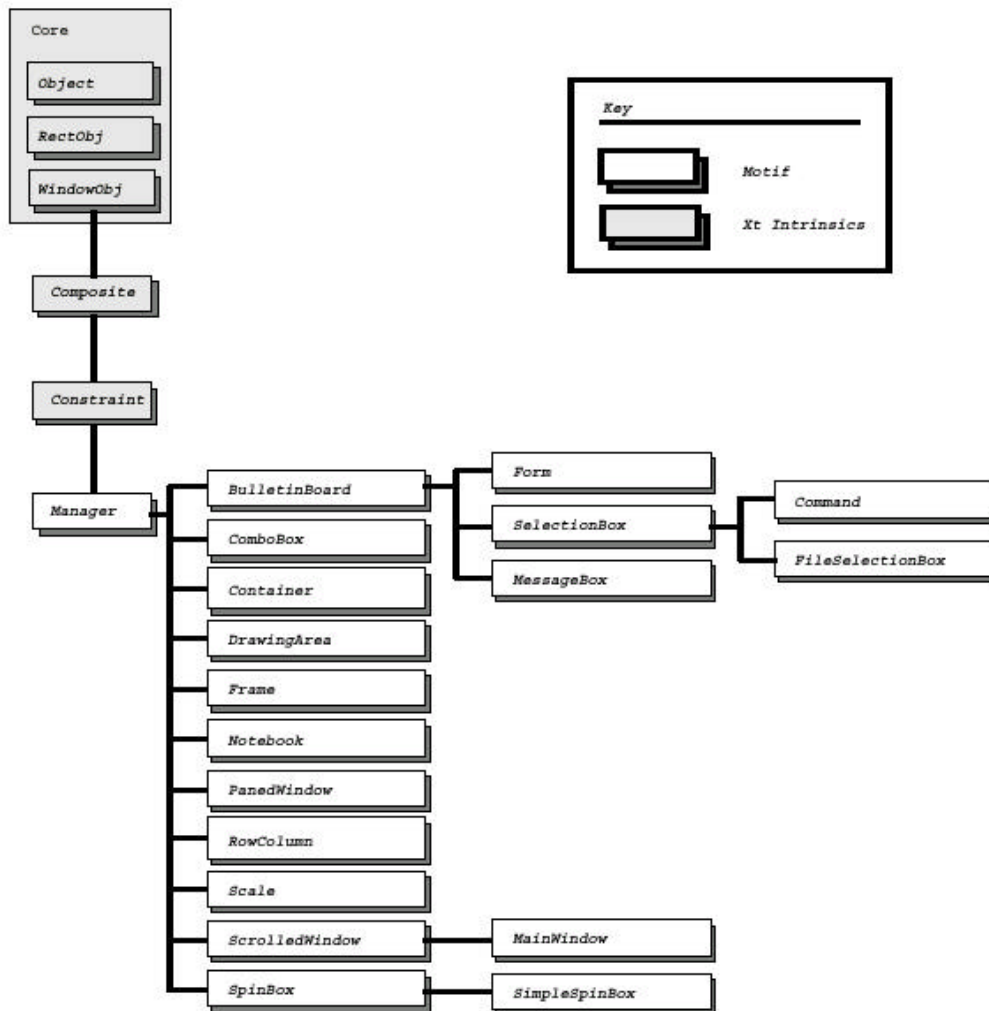


Fig. 1.9 Class hierarchy of a manager widget class

PANEDWINDOW

A PanedWindow arranges its children vertically from top to bottom or horizontally (Horizontal layout was introduced in Motif 2.0.), and forces them all to have the same width. In a vertical orientation, the widget takes its width from the widest widget in its list of children. When horizontally oriented, the PanedWindow

takes its height from the height of the tallest child. Each child is a *pane* of the window. Between each pair of panes, `PanedWindow` inserts an optional `Separator` and a control called a *sash*. By manipulating a sash with the mouse or keyboard, the user can increase or decrease the height of the pane above. `PanedWindow` has resources to control the margins, the spacing between panes, and the appearance of the sashes. Each pane of `aPanedWindow` has resources specifying a maximum and minimum height and whether or not either the pane itself or the `PanedWindow` should be allowed to resize the pane without user intervention.

SCROLLEDWINDOW CLASS

A `ScrolledWindow` manages a viewport and `ScrollBars` to implement a window onto a virtual scroll. The user can move the viewport to display different portions of the underlying scroll by using the `ScrollBars` or keyboard scrolling commands. `ScrolledWindow` is capable of performing scrolling operations automatically. In this mode, the application creates the widget that represents the scroll as a child of the

`ScrolledWindow`. The `ScrolledWindow` then creates a clipping window to act as the viewport, creates and manages the `ScrollBars`, and moves the viewport with respect to the scroll when the user issues a scrolling command.

`ScrolledWindow` can also allow the application to perform scrolling operations. In this mode, the application must create and manage the `ScrollBars` and must change the contents of the viewport in response to the user's scrolling commands. `List` and `Text` widgets are often used as virtual scrolls. Motif has convenience routines to create `List` and `Text` widgets inside `ScrolledWindows`, and the resulting `ScrolledList` and `ScrolledText` widgets perform scrolling operations without intervention by the application.

ROWCOLUMN CLASS

The `RowColumn` widget is perhaps the most widely used and robust of all of the manager widgets. The `RowColumn` widget lays out its children in rows and columns. Resources control the number of rows or columns and the packing of widgets into those rows and columns. The Motif toolkit uses the `RowColumn` internally to implement many objects that are not implemented as individual widgets, `RowColumn` implements both menus and nonmenu `WorkAreas`. Menus are widgets that allow the user to make choices among actions or states. Motif offers four basic kinds of menu:

- A `MenuBar` usually appears in the application's `MainWindow` and sometimes in other components. It most often consists of a row of `CascadeButtons` that, when activated, cause `PullDownMenus` to appear.
- A `PopupMenu` contains a set of choices that apply to a component of the application. The menu is not visible until the user takes an action that posts it. It can contain buttons that take action directly or `CascadeButtons` that cause `PullDownMenus` to appear.
- A `PullDownMenu` is associated with a `CascadeButton` in a `MenuBar`, a `PopupMenu`, or another `PullDownMenu`. The menu is not visible until the user posts it by activating the associated `CascadeButton`. Like a `PopupMenu`, a `PullDownMenu` can contain buttons that take action directly or `CascadeButtons` that cause other `PullDownMenus` to appear.
- An `OptionsMenu` allows the user to choose among one set of choices, usually mutually exclusive attributes or states. It consists of a label, a `CascadeButtonGadget` whose label shows the currently selected option, and a `PullDownMenu` containing buttons that represent the set of options.

One use for a nonmenu `RowColumn WorkArea` is to contain a set of `ToggleButton`s constituting a `RadioBox` or a `CheckBox`. When the user selects a `ToggleButton`, its state changes from on to off or

from off to on. Another use is to lay out an arbitrary set of widgets in a row, column, or two-dimensional formation.

BULLETINBOARD, FORM, MESSAGEBOX, SELECTIONBOX

Dialogs are container widgets that provide a means of communicating between the user and the application. A dialog widget usually asks a question or presents some information to the user. In some cases, the application is suspended until the user provides a response.

The usual superclass for a dialog widget is **XmBulletinBoard**. The dialog widget can be either a **BulletinBoard** itself or one of its more specialized subclasses. The **BulletinBoard** is the most basic of the manager widgets. The geometry management is, as the class name implies, like a bulletin board. A child is pinned up on the **BulletinBoard** in a particular location and remains there until it moves itself or someone else moves it. The **BulletinBoard** widget does not impose any layout policy on its children, but it does support keyboard traversal. The **BulletinBoard** is a superclass for more sophisticated and useful managers. The **BulletinBoard** is also designed to be used as the container for dialog boxes, so it has translation tables and callback routines for this purpose. The predefined Motif dialogs use the **BulletinBoard** widget class to handle all of their input mechanisms; each dialog widget class handles its own geometry management. The **BulletinBoard** is a container with no automatically created children; it supplies general behavior needed by most dialogs. Its subclasses provide child widgets and specific behavior tailored to particular types of dialogs:

- A **SelectionBox** is a **BulletinBoard** subclass that allows the user to select a choice from a list. It usually contains a **List**, an editable text field displaying the choice, and three or four buttons for accepting or canceling the choice and seeking help.
- A **FileSelectionBox** is a specialized **SelectionBox** for choosing a file from a directory. It contains two text fields, one containing a file search pattern and the other containing the selected filename; two lists, one displaying filenames and the other displaying subdirectories; and a set of buttons.
- A **Command** is a specialized **SelectionBox** for entering a command. Its main components are a text field for editing the command and a list representing the command history.
- A **MessageBox** is a **BulletinBoard** subclass for displaying messages to the user. It usually contains a message symbol, a message label, and up to three buttons. Motif provides distinct symbols for several kinds of messages: errors, warnings, information, questions, and notifications that the application is busy.
- A **TemplateDialog** is a specialized **MessageBox** that allows the application to build a custom dialog with additional children, such as a **MenuBar** and added buttons.
- A **Form** is a **BulletinBoard** subclass that performs constraint-based geometry management. The children of a **Form** have resources that represent attachments to other children or to the **Form**, offsets from the attachments, and relative positions within the **Form**. The **Form** calculates the positions and sizes of its children based partly on these constraints. This layout function makes **Form** useful outside dialogs as well.

DRAWINGAREA CLASS

A **DrawingArea** is a manager suited for use as a canvas containing graphical objects. An application must interact with a **DrawingArea** at a somewhat lower level than with other Motif widgets, but a **DrawingArea** provides the application with more fine-grained information about events. **DrawingArea** has callbacks to notify the application when the widget is exposed or resized and when it receives keyboard or mouse input.

An application generally must use Xlib routines to draw into the **DrawingArea**, and the application is responsible for updating the contents when necessary. The flexibility of a **DrawingArea**

makes it a useful widget for implementing both graphical and text features not provided by other Motif widgets.

Although the DrawingArea widget is subclassed from the Manager widget class, it is not generally used in the way that conventional managers are used. The widget does not do any drawing itself, and it doesn't define any keyboard or mouse behavior, although it does provide callbacks for user input. It is basically a free-form widget that can be used for application-specific purposes. The widget provides callback resources to handle keyboard, mouse, exposure, and resize events. While the DrawingArea widget can have children, it does not manage them in any defined way.

COMBOBOX CLASS

A ComboBox widget combines the capabilities of a TextField widget and a List widget. It allows users to enter information via TextField and also provides a list of possible choices via List to complete the text entry field. The application provides an array of compound strings that will fill this list and can also set the number of items that are visible in the list. If there are more items in the list than are viewable (as defined by the value of the **XmNvisibleItemCount** resource), a vertical scrollbar appears that allows the user to scroll through the list. The list can be displayed at all times, or it can be dropped down by the user by clicking on the down arrow in a drop-down-style ComboBox.

The TextField field in the ComboBox can be editable or non-editable. If the TextField field is editable, the user can type directly in the text field to enter a selection. If it is not editable, typing text may invoke a matching algorithm that will attempt to make a selection from the list using the characters typed by the user. In either case, list items can be selected using the keyboard and the mouse. When an item is selected, the item is displayed in reverse colors in the list and is displayed in the TextField field of the ComboBox.

SPIN BOX CLASS

A SpinBox is a manager that functions as a control by allowing the user to input data by selecting from, and rotating through, a set of values. It creates a pair of arrows that can be used to spin through a set of choices. The choices, which are usually related but mutually exclusive, are displayed consecutively one at a time in a single text field. Choices can be a range of numeric values or an ordered list of compound strings. The arrow buttons allow the user to advance or back up through the choices until the desired choice is displayed.

Text widget children are added to the Spin-Box, whereupon the range or set of values associated with each text is specified through constraint resources. The SpinBox automatically adds extra ArrowButtons which are used for rotating through the values of the text widget child which currently has the input focus. The programmer however has to supply the Text widgets underneath the SpinBox. For convenience, the SimpleSpinBox subclass is provided which encapsulates the most frequent use of this type of arrangement: it comes with a single built-in Text child.

CONTAINER CLASS

The Container class is a complex constraint widget which can lay out IconGadget children in three styles: in a tree arrangement, with a tabular data style, and in a free floating format based upon the x, y specifications for each child. The Container class allows for a more object-oriented approach to the front end of an application than the older MainWindow, in that the IconGadget children can pictorially represent application objects of some kind with the Container providing the layout and selection mechanisms.

NOTEBOOK CLASS

The Notebook class lays out its children as though they are pages in a book. That is, only one child is currently visible at any given time, and they all occupy a single area on the screen; the user can choose from the available pages either by selecting from Tabs which can be associated with a child, or by activating the Page Scroller, which is typically a SpinBox. To complete the analogy, resources are provided to control the general book-like characteristics of the Notebook in terms of its binding and overlapping page appearance. The Notebook is a constraint widget: you add children, and then specify the role which each child is to perform. Typically, a Form or other manager is added to represent some page, and optionally Push-Buttons can be added and associated with a page in order to represent Tab inserts along the edges of the Notebook pages.

EVENT HANDLING FOR WIDGETS

In one sense, the essence of X programming is the handling of asynchronous events. Events can occur in any order, in any window, as the user moves the pointer, switches between the mouse and the keyboard, moves and resizes windows, and invokes functions available through the user interface components.

X handles events by dispatching them to the appropriate application and to the separate windows that make up each application. Xlib provides many low-level functions for handling events. In special cases, one may need to dip down to this level to handle events. However, X simplifies event handling by having widgets handle many events for you, without any application interaction.

The functionality of a widget also encompasses its behavior in response to user events. This type of functionality is typically handled by action routines. Each widget defines a table of events, called a **translation table**, to which it responds. The translation table maps each event, or sequence of events, to one or more actions.

The translation table contains a list of event translations on the left side, with a set of action functions on the right side. When an event specified on the left occurs, the action routine on the right is invoked.

EVENT SPECIFICATION

In the Xt syntax, events are specified using symbols that are tied fairly closely to pure X hardware events. Keypress events are indicated by the symbol **keysyms**, which are hardware-independent symbols that represent individual keystroke.

Motif provides a further level of indirection in the form of **virtual keysyms**, which describe key events in a completely device-independent manner. For example, **osfActivate**, indicates that the user invoked an action that Motif considers to be an activating action. An activating action typically corresponds to the RETURN key being pressed or the left mouse button being clicked. Similarly, **osfHelp** corresponds to a user request for help, such as the HELP or F1 key being pressed.

Virtual keysyms can also be invoked by physical events, but Motif toolkit goes one step further and defines them in the form of virtual bindings. A system administrator, a user, or an application can specify virtual bindings. One common use of virtual bindings is to reconfigure the operation of the BACKSPACE and DELETE keys. A user can configure his own bindings by specifying the new virtual keysym bindings in a **.motifbind** file in his home directory. The advantage of using virtual bindings is that the interface remains consistent and nothing in the toolkit or the application needs to change.

Virtual keysyms bindings can also be set in a resource file, using the **XmNdefaultVirtualBindings** resource. The resource can be specified for all applications or on a per application basis. The only difference between the syntax for the resource specification and for the **.motifbind** file is that the resource specification must have a newline character (\n) between each entry.

CALLBACKS

Translations and actions allow a widget class to define associations between events and widget functions. A complex widget, such as the Motif Text Widget, is almost an application in itself, since its actions provide a complete set of editing functions. But beyond a certain point, a widget is helpless unless control is passed from the widget to the application. A widget that expects to call application functions defines one or more callback resources, which are the hooks on which application can hang its functions.

It is no accident that the callback resource names bear a resemblance to the names of widget action routines. In addition to highlighting the widget, the action routines call any application functions associated with the callbacks of the same name. There is no reason why a callback has to be called by an action; a widget could install a low-level event handler to perform the same task. However, this convention is followed by most widgets.

The widget's translation table registers the widget's interest in a particular type of event. When Xt receives an event that happened in the widget's window, it tests the event against the translation table. If there is no match, the event is thrown away. If there is a match, the event is passed to the widget and an action routine is invoked. The action routine may perform a function internal to the widget. Depending on the design of the widget, the action routine may then pass control to an application callback function. If the function is associated with a callback resource, it checks to see if a callback function has been registered for the resource, and if so, it dispatches the callback. There are several ways to connect an application function to a callback resource. The most common is to call XtAddCallback().

DEFINING TRANSLATIONS IN THE CLASS RECORD

All Motif widgets descend from Core. Therefore, all Motif widgets define keyboard translations through the following two fields of the Core class record:

1. The **tm_table** field, which holds the name of the string. This string contains information that maps event to action routine names.
2. The **actions** field, which holds the name of a string. This string contains information that maps the action routine names in the translation string to the action methods defined by your widget.

In addition to the two fields of CoreClassPart, both PrimitiveClassPart and ManagerClassPart provide an additional field named **translations**. You can set the translations field to one of the following:

1. NULL, meaning that the translations described by the tm_table of the CoreClassPart are the only translations defined in the class record.
2. XtInheritTranslations, meaning that you are inheriting the translations filed of your superclass.
3. A translations string of your own devising.

MOUSE BINDINGS

The **Motif Style Guide** defines a model for mouse button operations. By following this model, the widget's behavior will be consistent with other Motif widgets.

Motif Mouse Keysyms

Physical Key	Virtual Buttons	Purpose
Button 1	Select	Selects and Activates
Button 2	Transfer	Transfers data, including primary paste and drag operations
Button 3	Menu	Activates Popup Menus

VIRTUAL KEYSYMS

Purpose of Common Motif virtual Keysyms

Virtual Keysyms	Purpose
-----------------	---------

osfActivate	Activates a default action
osfAddMode	Toggles the selection mode between Normal and Add mode
osfBackSpace	Deletes the previous character
osfBeginLine	Moves the cursor to the beginning of the line
osfCancel	Cancels the current operation
osfClear	Clears the current selection
osfCopy	Copies to the clipboard
osfCut	Cuts to the clipboard
osfDelete	Deletes data
osfDeselectAll	Deselects the current selection
osfDown	Moves the cursor down
osfEndLine	Moves the cursor to the end of the line
osfHelp	Calls the function specified by the XmNhelpCallback resource
osfInsert	Toggles between Replace and Insert mode (if specified without modifiers)
osfLeft	Moves the cursor to the left
osfMenu	Activates the Popup Menu
osfMenuBar	Traverses to the MenuBar
osfPageDown	Moves down one page
osfPageLeft	Moves left one page
osfPageRight	Moves right one page
osfPageUp	Moves up one page
osfPaste	Pastes from the clipboard
osfPrimaryPast	Pastes the primary selection
osfRestore	Restores a previous setting
osfRight	Moves the cursor to the right
osfSelect	Establishes the current selection
osfSelectAll	Selects an entire block of data
osfSwitchDirection	Toggles the string layout direction
osfUndo	Undoes the most recent action
osfUp	Moves the cursor up

KEYBOARD TRAVERSAL

Most Motif users move a mouse in order to move the cursor from one widget to another. However, the **Motif Style Guide** insists that Motif applications be usable even when a mouse is not available. For that reason, most Motif applications allow users to traverse between widgets by pressing the tab key or by pressing one of the arrow keys.

In writing a Motif widget, very little is needed in order to implement keyboard traversal. Code in the superclass XmPrimitive and XmManager handle keyboard traversal for the vast majority of widgets. In order to tap this code, the widget need only inherit the traversal translations of the appropriate superclass. To do this, simply specify XtInheritTranslations for the translations field of the PrimitiveClassPart or ManagerClassPart.

Widget writers may also wish to override the default values of two superclass resources, XmNtraversalOn and XmNnavigationType.

If the XmNtraversalOn resource is set to *True* (as it is by default), then traversal is activated for this widget. However, output-only widgets like *ExmString* should not have keyboard traversal activated. Therefore, widgets like *ExmString* override the default value of XmNtraversalOn, and set it to *False* instead.

The `XmNnavigationType` resource determines whether the widget is a tab group. The `XmPrimitive` widget establishes a default value of `XmNONE` for this resource. By contrast, the `XmManager` widget sets the default to `XmTabGroup`. Widget writers may want to set different defaults.

With all the created widgets, we need to explicitly call certain methods to display or realize them. This is achieved via the `XtRealizeWidget()` function. After which, the program has to enter the main event handling loop. The function `XtAppMainLoop(app)` does this. After this call, Xt has control over the program and it is this that dispatches events that invoke callbacks etc.

SAMPLE CODE

```
/* Written by Dan Heller and Paula Ferguson.
** Copyright 1994, O'Reilly & Associates, Inc.
**
** The X Consortium, and any party obtaining a copy of these files
from
** the X Consortium, directly or indirectly, is granted, free of
charge, a
** full and unrestricted irrevocable, world-wide, paid up, royalty-
free,
** nonexclusive right and license to deal in this software and
** documentation files (the "Software"), including without limitation
the
** rights to use, copy, modify, merge, publish, distribute,
sublicense,
** and/or sell copies of the Software, and to permit persons who
receive
** copies from any such party to do so. This license includes without
** limitation a license to do the foregoing actions under any patents
of
** the party supplying this software to the X Consortium.
**
** Modified by A.J.Fountain, IST
** for Motif 2.1, X11R6, ANSI.
*/

/* xmemo.c -- a memo calendar program that creates a calendar on the
** left and a list of months on the right. Selecting a month changes
** the calendar. Selecting a day causes that date to become activated
** and a popup window is displayed that contains a text widget. This
** widget is presumably used to keep memos for that day. You can pop
** up and down the window by continuing to select the date on that
month.
*/
#include <stdio.h>
#include <X11/Xos.h>
#include <Xm/List.h>
#include <Xm/Frame.h>
#include <Xm/LabelG.h>
#include <Xm/PushB.h>
#include <Xm/RowColumn.h>
```

```

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

int    year;
void   date_dialog(Widget, XtPointer, XtPointer);
void   set_month(Widget, XtPointer, XtPointer);
Widget list_w, month_label;

typedef struct _month {
    char    *name;
    Widget  form, dates[6][7];
} Month;

Month months[] = { /* only initialize "known" data */
    { "January" }, { "February" }, { "March" }, { "April" }, { "May" },
    { "June" }, { "July" }, { "August" }, { "September" }, { "October" },
    { "November" }, { "December" }
};

/* These only take effect if the app-defaults file is not found */
String fallback_resources[] = {
    "*bold.fontName: -*-courier-bold-r-*--18-*",
    "*bold.fontType: FONT_IS_FONT",
    "*medium.fontName: -*-courier-medium-r-*--18-*",
    "*medium.fontType: FONT_IS_FONT",
    "*XmPushButton*.renderTable: bold",
    "*XmLabelGadget*.renderTable: medium",
    NULL
};

main (int argc, char *argv[])
{
    Widget      toplevel, frame, rowcol, rowcol2, label;
    XtAppContext app;
    int         month;
    Arg         args[8];
    int         n;

    XtSetLanguageProc (NULL, NULL, NULL);

    toplevel = XtVaOpenApplication (&app, "XMemo", NULL, 0, &argc,
    argv,
    sessionShellWidgetClass, NULL);

    /* The form is the general layout manager for the application.
    ** It will contain two widgets (the calendary and the list of
    months).
    ** These widgets are laid out horizontally.
    */
    n = 0;
    XtSetArg (args[n], XmNorientation, XmHORIZONTAL); n++;
    rowcol = XmCreateRowColumn (toplevel, "rowcol", args, n) ;

```

```

/* Place a frame around the calendar... */
frame = XmCreateFrame (rowcol, "frame1", NULL, 0) ;
/* the calendar is placed inside of a RowColumn widget */
rowcol2 = XmCreateRowColumn (frame, "rowcol2", NULL, 0) ;
/* the month label changes dynamically as each month is selected
*/
month_label = XmCreateLabelGadget (rowcol2, "month_label", NULL,
0);
XtManageChild (month_label);
label = XmCreateLabelGadget (rowcol2, " Su Mo Tu We Th Fr Sa",
NULL,0);
XtManageChild (label);

/* Create a ScrolledText that contains the months. You probably
won't
** see the ScrollBar unless the list is resized so that not all of
** the month names are visible.
*/
{
    XmString strs[XtNumber (months)];
    for (month = 0; month < XtNumber (months); month++)
        strs[month] = XmStringCreateLocalized
(months[month].name);
    list_w = XmCreateScrolledList (rowcol, "list", NULL, 0);
    XtVaSetValues (list_w, XmNitems,
        strs,
        XmNitemCount,
        XtNumber (months),
        NULL);
    for (month = 0; month < XtNumber (months); month++)
        XmStringFree (strs[month]);
    XtAddCallback (list_w, XmNbrowseSelectionCallback, set_month,
NULL);
    XtManageChild (list_w);
}

/* Determine the year we're dealing with and establish today's
month */
if (argc > 1)
    year = atoi (argv[1]);
else {
    long time(long *), t = time ((long *) 0);
    struct tm *today = localtime (&t);
    year = 1900 + today->tm_year;
    month = today->tm_mon + 1;
}
XmListSelectPos (list_w, month, True);

XtManageChild (rowcol2);
XtManageChild (frame);
XtManageChild (rowcol);

```

```

    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

/* set_month() -- callback routine for when a month is selected.
** Each month is a separate, self-contained widget that contains the
** dates as PushButton widgets. New months do not overwrite old ones,
** so the old month must be "unmanaged" before the new month is
managed.
** If the month has not yet been created, then figure out the dates
and
** which days of the week they fall on using clever math
computations...
*/
void set_month(Widget w, XtPointer client_data, XtPointer call_data)
{
    XmListCallbackStruct *list_cbs = (XmListCallbackStruct *)
call_data;
    char text[BUFSIZ];
    register char *p;
    int i, j, m, tot, day;
    static int month = -1;
    XmString xms;
    Arg args[8];
    int n;

    if (list_cbs->item_position == month + 1)
        return; /* same month, don't bother redrawing */

    if (month >= 0 && months[month].form)
        XtUnmanageChild (months[month].form); /* unmanage last month
*/
    month = list_cbs->item_position - 1; /* set new month */
    sprintf (text, "%s %d", months[month].name, year);
    xms = XmStringGenerate ((XtPointer) text, NULL, XmCHARSET_TEXT,
NULL);
    XtVaSetValues (month_label, XmNlabelString, xms, NULL);
    XmStringFree(xms);
    if (months[month].form) {
        /* it's already been created -- just manage and return */
        XtManageChild (months[month].form);
        return;
    }

    /* Create the month Form widget and dates PushButton widgets */
    n = 0;
    XtSetArg (args[n], XmNorientation, XmHORIZONTAL); n++;
    XtSetArg (args[n], XmNnumColumns, 6); n++;
    XtSetArg (args[n], XmNpacking, XmPACK_COLUMN); n++;
    months[month].form = XmCreateRowColumn (XtParent(month_label),
"month_form", args, n);

```

```

        /* calculate the dates of the month using science */
        /* day_number() takes day-of-month (1-31), returns day-of-week (0-
6) */
        m = day_number (year, month + 1, 1);
        tot = days_in_month (year, month + 1);

        /* We are creating a whole bunch of PushButtons, but not all of
        * them have dates associated with them.  The buttons that have
        * dates get the number sprintf'ed into it.  All others get two
blanks.
        */
        for (day = i = 0; i < 6; i++) {
            for (j = 0; j < 7; j++, m += (j > m && --tot > 0)) {
                char *name;
                if (j != m || tot < 1)
                    name = " ";
                else {
                    sprintf(text, "%2d", ++day);
                    name = text;
                }

                n = 0;
                /* this is where we will hold the dialog later. */
                XtSetArg (args[n], XmNuserData, NULL); n++;
                XtSetArg (args[n], XmNsensitive, (j % 7 == m && tot > 0));
n++;

                XtSetArg (args[n], XmNshadowThickness, 0); n++;
                months[month].dates[i][j] = XmCreatePushButton
(months[month].form, name, args, n);
                XtManageChild (months[month].dates[i][j]);
                XtAddCallback
                    (months[month].dates[i][j],
XmNactivateCallback, date_dialog, (XtPointer) day);
            }
            m = 0;
        }
        XtManageChild (months[month].form);

        /* The RowColumn widget creates equally sized boxes for each child
        ** it manages.  If one child is bigger than the rest, all children
        ** are that big.  If we create all the PushButtons with a 0 shadow
        ** thickness, as soon as one PushButton is selected and its
thickness
        ** is set to 2, the entire RowColumn resizes itself.  To
compensate
        ** for the problem, we need to set the shadow thickness of at
least
        ** one of the buttons to 2, so that the entire RowColumn is
        ** initialized to the right size.  But this will cause the button
to
        ** have a visible border and make it appear preselected, so, we
have

```

```

    ** to make it appear invisible.  If it is invisible then it cannot
be
    ** selected, but it just so happens that the last 5 days in
    ** the month will never have selectable dates, so we can use any
one
    ** of those.  To make the button invisible, we need to unmap the
    ** widget.  We can't simply unmanage it or the parent won't
consider
    ** its size, which defeats the whole purpose.  We can't create the
    ** widget and then unmap it because it has not been realized, so
it
    ** does not have a window yet.  We don't want to realize and
manage
    ** the entire application just to realize this one widget, so we
    ** set XmNmappedWhenManaged to False along with the shadow
thickness
    ** being set to 2.  Now the RowColumn is the right size.
*/

```

```

    XtVaSetValues (months[month].dates[5][6],XmNshadowThickness, 2,
XmNmappedWhenManaged, False,NULL);
}

```

```

/* date_dialog() -- when a date is selected, this function is called.
** Create a dialog (oplevel shell) that contains a multiline text
** widget for memos about this date.
*/

```

```

void date_dialog(Widget w, XtPointer client_data, XtPointer call_data)
{

```

```

    int date = (int) client_data;
    Widget dialog;
    XWindowAttributes xwa;

```

```

    /* the dialog is stored in the PushButton's XmNuserData */

```

```

    XtVaGetValues (w, XmNuserData, &dialog, NULL);

```

```

    if (!dialog) {
        /* it doesn't exist yet, create it. */
        char buf[32];
        Arg args[5];
        int n, n_pos, *list;

```

```

        /* get the month that was selected -- we just need it for its
name */

```

```

        if (!XmListGetSelectedPos (list_w, &list, &n_pos))
            return;

```

```

        sprintf (buf, "%s %d %d", months[list[0]-1].name, date, year);
        XtFree ((char *) list);

```

```

        dialog = XtVaCreatePopupShell ("popup",
            topLevelShellWidgetClass, XtParent (w),
            XmNtitle, buf,
            XmNallowShellResize, True,
            XmNdeleteResponse, XmUNMAP,

```

```

        NULL);
    n = 0;
    XtSetArg (args[n], XmNrows,      10); n++;
    XtSetArg (args[n], XmNcolumns,   40); n++;
    XtSetArg (args[n], XmNeditMode,  XmMULTI_LINE_EDIT); n++;
    XtManageChild (XmCreateScrolledText (dialog, "text", args,
n));
    /* set the shadow thickness to 2 so user knows there is a memo
    ** attached to this date.
    */
    XtVaSetValues (w,
        XmNuserData, dialog,
        XmNshadowThickness, 2,
        NULL);
}
/* See if the dialog is realized and is visible.  If so, pop it
down */
if (XtIsRealized (dialog) && XGetWindowAttributes
    (XtDisplay (dialog), XtWindow (dialog), &xwa) &&
    xwa.map_state == IsViewable)
    XtPopdown (dialog);
else
    XtPopup (dialog, XtGrabNone);
}

/* the rest of the file is junk to support finding the current date.
*/

static int mtbl[] = { 0,31,59,90,120,151,181,212,243,273,304,334,365
};

int days_in_month(int year, int month)
{
    int days;

    days = mtbl[month] - mtbl[month - 1];
    if (month == 2 && year % 4 == 0 && (year % 100 != 0 || year % 400
== 0))
        days++;
    return days;
}

int day_number(int year, int month, int day)
{
    /* Lots of foolishness with casts for Xenix-286 16-bit ints */

    long days_ctr;      /* 16-bit ints overflowed Sept 12, 1989 */

    year -= 1900;
    days_ctr = ((long)year * 365L) + ((year + 3) / 4);
    days_ctr += mtbl[month - 1] + day + 6;
    if (month > 2 && (year % 4 == 0))

```

```
    days_ctr++;
    return (int) (days_ctr % 7L);
}
```

Conclusion

Motif came to being as a response to the use of graphical user interfaces and at the same time make up for the drawbacks of GUI. Most basic window programming are quite large because of the necessity to write or call upon many functions to control the windowing system. Motif now becomes an efficient manner of reducing such problems by packaging common GUI entities together as widgets. Also, Motif has a single data structure for all its widgets, allowing it to be used in any platform without needing much changes and causing inconveniences on the developers/designers.

Appendix A

CONCEPTS

Motif

Essentially, Motif is more of a specification than an implementation. This means that Motif GUIs can still be created even without using Motif toolkits. The specifications focused mainly on the design of the objects that make up a user interface.

The Motif specification is broken down into two basic parts:³

- The output model describes what the objects on the screen look like. This model includes the shapes of buttons, the use of three-dimensional effects, the use of cursors and bitmaps, and the positioning of windows and subwindows. Although some recommendations are given concerning the use of fonts and other visual features of the desktops, Motif is flexible in most of these recommendations.
- The input model specifies how the user interacts with the elements on the screen. The key point of the specification is that consistency should be maintained across all applications. Similar user-interface elements should look and act similarly regardless of the application that contains them.

Gadgets

There may be occasions in Motif programming when we want to have the properties of a primitive widget but do not wish to worry about the management of the window properties of the widget. Motif has to manage each created widget's window and associated resources. If we have several widgets within our interface this could cause complications for our application.

To attempt to alleviate many of these problems Motif provides *Gadgets*. Gadgets are basically windowless widgets and, therefore, require less resources than a widget. Control of the gadget is the responsibility of the parent of the gadget. Gadgets don't handle their own events and is instead handled by the manager widget containing them.

We use gadgets instead of widgets because more widgets mean more X windows. The lesser number of X windows we have, the more efficient our application becomes. But then again, windows doesn't really eat up that much resources and X implementations have all ready improved which makes using widgets just as fine.

Not all widgets have corresponding gadgets. The following gadgets are available: **ArrowButton**, **Label** and **Separator**. They behave in a similar manner to their corresponding widgets.

³ Anthony Fountain, et al. *Motif Programming Manual*. United Kingdom: O'Reilly & Associates, Inc., 2001. p.5.

Widget Classes

The organisation of a large system of GUI components in Motif can be quite complex. In order to aid the design and understanding of Motif various widget classes have been constructed. Each class can be categorised by a broad functionality, at a variety of levels. Motif defines a hierarchy of widget classes (Fig A) and a widget will *inherit* properties from a higher class in the widget hierarchy. Some levels of the hierarchy have strong relationships with Xt Ininsics since widgets are actually created in this toolkit.

Shell Widgets

All widgets are contained in a shell widget. This is usually the top level widget. The primary function of a shell widget is as an interface to the window manager.

The **application** shell is normally the top level for an application and is created by `XtVaAppInitialize()` or related functions.

There are two other shells:

- The **Override shell** is used for pop-up menus that must be at the top-level. To achieve this, the window manager must usually be bypassed. Consequently, the override shell is not often used by Motif programs.
- The **Transient** shell is used for dialogs. However Motif repackages this shell so that dialogs become a subclass widget.

Constraint Widgets

Constraint widgets are concerned with the positioning and alignment of widgets contained within them. *XmManager* is a (Motif) subclass of the constraint widget specifying general manager facilities concerned with, for example, callbacks and highlight colour.

Construction widgets

Motif defines two basic classes of widgets that provide the basic building blocks of any GUI: **primitive** and **manager**. The majority of the remainder of this text is devoted to these widget classes. Within each of these basic classes, several different sub-classes of widget are defined.

The broad function of each class is as follows:

- **Primitive** widgets are designed to work as a single entity. They provide the building blocks with which we assemble our GUI. The `PushButton` is an example of a primitive widget class.
- **Manager** widgets are designed to be *containers* and may have primitive and manager widgets placed under their control. The main function of manager widgets is to help control the design of a GUI. Manager widgets control how we organise a GUI by prescribing standard, or uniform, layouts or providing widgets that let us place widgets in an ordered fashion.

Widget Resources

Each widget has a number of resources. These control many features of the widget such as the foreground and background colours, size *etc.*. A particular widget will have specialised resources such as callback resources which define how the widget responds to an event *etc.*

Every widget is documented in the Motif Reference Manual which gives a complete list of the resources that a particular widget employs. When discussing individual widgets we will only consider the important resources that define the main characteristics of the widget concerned. The following Chapter addresses how widget resources can be set and altered for a given application.

The levels of the hierarchy and related widget resources are:

- **Core**
 - This *superclass* gives background, size and position resources that are common to all widgets.
- **XmPrimitive**
 - The superclass of all primitive widgets defines resources related areas such as foreground and highlight colour.
- **Composite**
 - The superclass of containers for widgets has two sub-classes:
- **Shell**
 - widgets have resources related to interfacing with the window manager.
- **Constraint**
 - widgets have resources that are concerned with the positioning and alignment of widgets contained within. *XmManager* is a subclass of constraint and specifies general manager widget resources such as callbacks and highlight colour.
- **Widget level**
 - Resources particular to a specific widget.

Resources

Each Motif widget is defined as a C data structure. These resources or the widget's data attributes are defined by these data structures. In motif, there is this concept of *resources*. These resources are the look and properties of the widget, which is set by the developer, depending on how he wants the widget to behave and look like. This can be done by using the `XtSetArg` method, after which, the `arg` array is passed as an argument in the method creating the widget. Another option is setting the resources is to enter each desired property as method argument, when creating the widget, immediately.

```
button = XtVaCreateWidget ("pushme",
xmPushButtonWidgetClass,toplevel,
XmNlabelString, label, XmNwidth, 200, XmNheight, 50, NULL);
```

or

```
Arg args[2];
int n = 0;
XtSetArg(args[n], XmNlabelString, label); n++;
label = XmCreateLabel (toplevel, "label", args, n);
XtManageChild (label);
```

Difference between invoking `XtCreateWidget` and `XmCreateWidgetName`

`XtCreateWidget` is a generic callback resource and requires the programmer to indicate the widget class that is required for the creation of the widget, according to what widget is desired to be created. `XmCreateWidgetName`, on the other hand, is specific as the widget class that is equivalent to the widget name is indicated beforehand.

Compiling a Motif Program

To compile a Motif program, we have to link the Motif, Xt and Xlib libraries. This is how to compile a Motif program:

```
<compiler> <filename.c> -o <exe> -lXm -lXt -lX11
```

where

-lXm denotes the access of the motif library,
-lXt denotes the X toolkit, and
-lX11 denotes the access of the X11 library

Take for example if we want to compile `push.c`: `gcc push.c -o push -lXm -lXt -lX11`

A sample makefile for motif:

```
MOTIFHOME = /usr/X11R6/  
CFLAGS = -g -I$(MOTIFHOME)/include -I$(OPENWINHOME)/include  
LIBS = -R$(MOTIFHOME)/lib -R$(OPENWINHOME)/lib -L$(MOTIFHOME)/lib  
-L$(OPENWINHOME)/lib -lXm -lXt -lX11  
PROG = helloOBSJ = hello.c$(PROG): $(OBSJ)  
$(CC) $(CFLAGS) $(OBSJ) -o $(PROG) $(LIBS)
```

Running a Motif Program

To run: `<exe>`, in this case, `push` is typed.

Sample Hello world program:

```
/* hello.c -- initialize the toolkit using an application context and  
a  
** toplevel shell widget, then create a pushbutton that says Hello  
using  
** the varargs interface.  
*/  
#include <Xm/PushButton.h>  
  
main (int argc, char *argv[])  
{  
    Widget          toplevel, button;  
    XtAppContext    app;  
    void            button_pushed(Widget, XtPointer, XtPointer);  
    XmString        label;  
    Arg             args[2];  
    XtSetLanguageProc (NULL, NULL, NULL);
```

```
    toplevel = XtVaOpenApplication (&app, "Hello", NULL, 0, &argc,
argv, NULL,
                                sessionShellWidgetClass, NULL);

    label = XmStringCreateLocalized ("Push here to say hello");
    XtSetArg(args[0], XmNlabelString, label);
    button = XmCreatePushButton (toplevel, "pushme", args, 1);
    XmStringFree (label);

    XtAddCallback (button, XmNactivateCallback, button_pushed, NULL);
    XtManageChild (button);

    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

void button_pushed (Widget widget, XtPointer client_data, XtPointer
call_data)
{
    printf ("Hello World!\n");
}
```

Appendix B

X/MOTIF BUILDERS AND VENDORS

BX PRO

ICS: Integrated Computer Solutions
www.ics.com

LXB - Linux X11/Motif GUI Builder

bruce parkin, university of minnesota
<http://www.tc.umn.edu/~parki005/lxb/lxb.html>

X-Designer

Imperial Software Technology
<http://www.ist.co.uk/xd/>

ViewKit

ICS: Integrated Computer Solutions
<http://www.viewkit.com/>

Builder Accessory

SGI: Silicon Graphics Inc.
<http://www.sgi.com>

Builder Xcessory

Scientific Computers Ltd.
<http://www.scl.com/products/ics/builders/>
www.scl.com/products/ics/builders

BIBLIOGRAPHY:

Fountain, Antony; Huxtable, Jeremy; Ferguson, Paula; and Heller, Dan. *The Definitive Guides to the X Windows System, Motif Programming Manual*. United Kingdom: O'Reilly & Associates, Inc., Anthony Fountain and Jeremy Huxtable, 2001.

http://www.cs.cf.ac.uk/Dave/X_lecture/X_book_caller/X_book_caller.html#ch:motif2.0

<http://www.opengroup.org/tech/desktop/motif/motif.data.sheet.htm>

<http://www.hiraeth.com/alan/tutorials/xmotif/xmotif.html>

The Open Group, *Desktop Product Documentation: Motif 2.1—Programmer's Guide*, (UK: The Open Group, 1997), pp. 31-37.