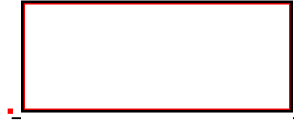


Slackware Linux Unleashed



Introduction

Part I Introduction

1 Introduction to Linux

2 Types of Linux

3 Installing Linux

4 Booting Linux

5 Special Installations

Part II Using Linux

6 Getting Started

7 Basic Commands

8 File System

9 GNU Project Utilities

10 `bash`

11 `pdksh`

12 `tcsh`

13 Shell Programming

14 Communications Tools

15 Using the Linux Documentation

Part III Editing, Typesetting, and More

16 Text Editors

17 `groff`

18 `gqn` and `gtbl`

19 `TeX`

20 Printing

Part IV GUI

21 Installing XFree86

22 Configuring XFree86

23 Using Motif

24 OPEN LOOK and OpenWindows

25 Ghostscript

Part V Linux for Programmers

26 gawk

27 Programming in C

28 Programming in C++

29 Perl

30 Tcl and Tk

31 Other Compilers

32 ObjectBuilder

33 Interviews

34 Motif Programming

35 XView Programming

36 SmallTalk/X

37 Mathematics on Linux

Part VI Linux for System Administrators

38 System Administration Basics

39 Devices

40 Processes

41 Users

42 SCSI Devices

43 Networking

44 UUCP

45 Mail and News

46 Network Security

Part VII Setting Up an Internet Site

47 Setting Up an Internet Site

48 FTP and Anonymous FTP Sites

49 Configuring a WAIS Site

50 Setting Up a Gopher Service

51 Setting Up WWW Services

52 CGI Scripts

53 HTML Programming Basics

54 Java and JavaScript

Part VIII Advanced Programming Topics

55 Source Code Control

56 Working with the Kernel

57 Device Drivers

58 The Pseudo File System

59 Network Programming

60 Server Support for PEX

61 Using Browsers

Part IX Applications

62 DOSemu

63 Wine

64 HylaFAX

65 Games

66 Databases

67 Cactus/Lone Star Utilities

68 Useful Personal Tools in Linux

69 Graphics Tools in Linux

Part X Appendixes

A FTP Sites and Newsgroups

B Commercial Vendors for Linux

C The Linux Documentation Project

D The GNU General Public License

E Copyright Information

F What's on the CD-ROM

- [- 33 -](#)
 - [Interviews](#)
 - [What Is the Interviews System?](#)
 - [How to Get and Install Interviews](#)
 - [TIP](#)
 - [Sample Applications](#)
 - [FIGURE 33.1.](#)
 - [FIGURE 33.2.](#)
 - [Listing 33.1. The](#)
 - [binary files for Interviews.](#)
 - [TIP](#)
 - [idraw](#)
 - [FIGURE 33.3.](#)
 - [TIP](#)
 - [idraws Pull-Down Menus](#)
 - [Changes to Xdefaults](#)
 - [ibuild](#)
 - [FIGURE 33.4.](#)
 - [TIP](#)
 - [TIP](#)
 - [TIP](#)
 - [NOTE](#)
 - [NOTE](#)
 - [NOTE](#)
 - [CAUTION](#)
 - [ibuilds Pull-Down Menus](#)
 - [Generating Source Files](#)
 - [NOTE](#)
 - [Subclasses](#)
 - [Code Generation](#)
 - [Additional Resources in Xdefaults](#)
 - [Building an Application with ibuild](#)
 - [FIGURE 33.5.](#)
 - [FIGURE 33.6.](#)
 - [FIGURE 33.7.](#)
 - [NOTE](#)
 - [FIGURE 33.8.](#)

- [NOTE](#)
- [CAUTION](#)
- [CAUTION](#)
 - [Adding Interactors](#)
 - [FIGURE 33.9.](#)
 - [FIGURE 33.10.](#)
 - [FIGURE 33.11.](#)
- [NOTE](#)
- [Listing 33.2. The Sample](#)
- [1.h file.](#)
- [Listing 33.3.](#)
- [The Sample1.c file.](#)
- [Listing 33.4.](#)
- [Adding code to the callback function.](#)
- [NOTE](#)
- [Summary](#)

- 33 -

Interviews

by Kamran Husain

IN THIS CHAPTER

- What Is the Interviews System?
- How to Get and Install Interviews
- Sample Applications
- idraw
- Changes to Xdefaults

- ibuild
- Code Generation
- Building an Application with ibuild

In this chapter, you will learn about the Interviews C++ class library development system. You will learn about the tools available for Interviews and how to use them to create front-end GUI applications.

Interviews offers a rich set of C++ functionality and tools for building applications. This chapter can't possibly do it justice in the space provided. By reading to the end of the chapter, though, you will get a feel for creating truly object-oriented applications using Interviews.

To get the most out of this chapter, you need some knowledge of how to run the C++ compiler and how to program in object-oriented languages. If you have not already done so, you should install the X window package with at least Motif window manager (mwm) or the OPEN LOOK manager (olwm).

What Is the Interviews System?

The Interviews subsystem is a windowing system for X Window. Interviews is copyrighted by The Board of Trustees of the Leland Stanford Junior University. Interviews was developed by Mark Linton's group at Stanford.

What makes Interviews an interesting environment is that it provides an object-oriented approach to building user interfaces. All components within Interviews, such as windows, menus, scrollbars, and so on, are objects with inherited behavior. The name Interviews is derived from the package presenting an interactive view of some data. For example, a front end to a database provides an interactive view to the data in the database.

How to Get and Install Interviews

The Interviews kit is distributed in two packages: iv1 and iv2. The first package, iv1, contains the binary and source files for the libraries. The second package includes the man pages and a PostScript reference manual for the Interviews system.

The Interviews package is included on the CD that comes with this book.

TIP: I recommend that you print the PostScript version of the Interviews manual if you plan on developing any applications in Interviews. This manual is in the file `/usr/doc/interviews/ refman.PS`. You need a PostScript printer to print this document, or you can view this manual via `ghostscript`.

Sample Applications

The Interviews package comes with several applications found in the `/usr/interviews/bin` directory. Applications to give you an idea of some of Interviews' potential are shown in the `dclock` and `idemo` applications, shown in Figures 33.1 and 33.2.



FIGURE 33.1. *The Interviews `dclock` application.*

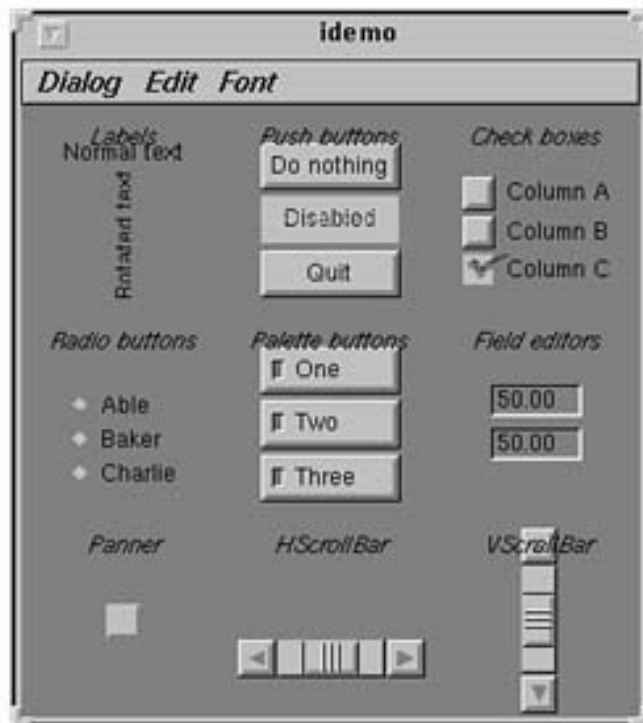


FIGURE 33.2. *A demo application with Interviews.*

The list of other ready-to-run applications under Interviews is shown in Listing 33.1.

We will cover some of these applications in a bit more detail in this chapter. The rest you can try out on your own.

Listing 33.1. The binary files for Interviews.

```
$ ls -al /usr/interviews/bin
```

```
total 1187
```

```
drwxr-xr-x  2 root  root    1024 Nov  5 17:40 .
```

drwxr-xr-x	6	root	root	1024	Nov	5	17:40	..
-rwxr-xr-x	1	root	root	9220	Apr	17	1994	alert
-rwxr-xr-x	1	root	root	82	Apr	17	1994	cpu
-rwxr-xr-x	1	root	root	17412	Apr	17	1994	dclock
-rwxr-xr-x	1	root	root	259076	Apr	17	1994	doc
-rwxr-xr-x	1	root	root	1616	Apr	17	1994	ibmkmf
-rwxr-xr-x	1	root	root	689156	Apr	17	1994	ibuild
-rwxr-xr-x	1	root	root	37892	Apr	17	1994	iclass
-rwxr-xr-x	1	root	root	17412	Apr	17	1994	idemo
-rwxr-xr-x	1	root	root	115716	Apr	17	1994	idraw
-rwxr-xr-x	1	root	root	9220	Apr	17	1994	ifc
-rwxr-xr-x	1	root	root	642	Apr	17	1994	ivmkmf

```

-rwxr-xr-x    1 root    root          13316 Apr 17  1994 logo
-rwxr-xr-x    1 root    root          17412 Apr 17  1994 mailbox
-rwxr-xr-x    1 root    root          1736  Apr 17  1994 remind

```

TIP: You can always try the applications in `/usr/interviews/bin` from an `xterm` by running the command as a background process. For example, to run the `ibuild` application, type the command `ibuild &`. If you do not run `ibuild` as a background process, you can press `Ctrl-z` to put it in the background of a bash shell.

idraw

Let's start with an application in the Interviews package. Learning this application will familiarize you with Interviews and working with its development tools. (See Figure 33.3.)

The `idraw` application lets you draw rectangles, polygons, ellipses, and other shapes interactively. Drawings are stored in files that can be printed on a PostScript printer. You can open an existing drawing by typing a filename on the command line when starting up `idraw`.

You must engage a tool before you can use it. You engage a tool by clicking on its icon, or by typing the character that is below and to the right of its icon. The icon of the drawing tool that's engaged appears in inverted colors. When it is engaged, you use the tool by clicking the left mouse button in the drawing area.

The Select, Move, Scale, Stretch, Rotate, and Alter tools manipulate existing graphics. Magnify makes a part of the view expand to fill the entire view.

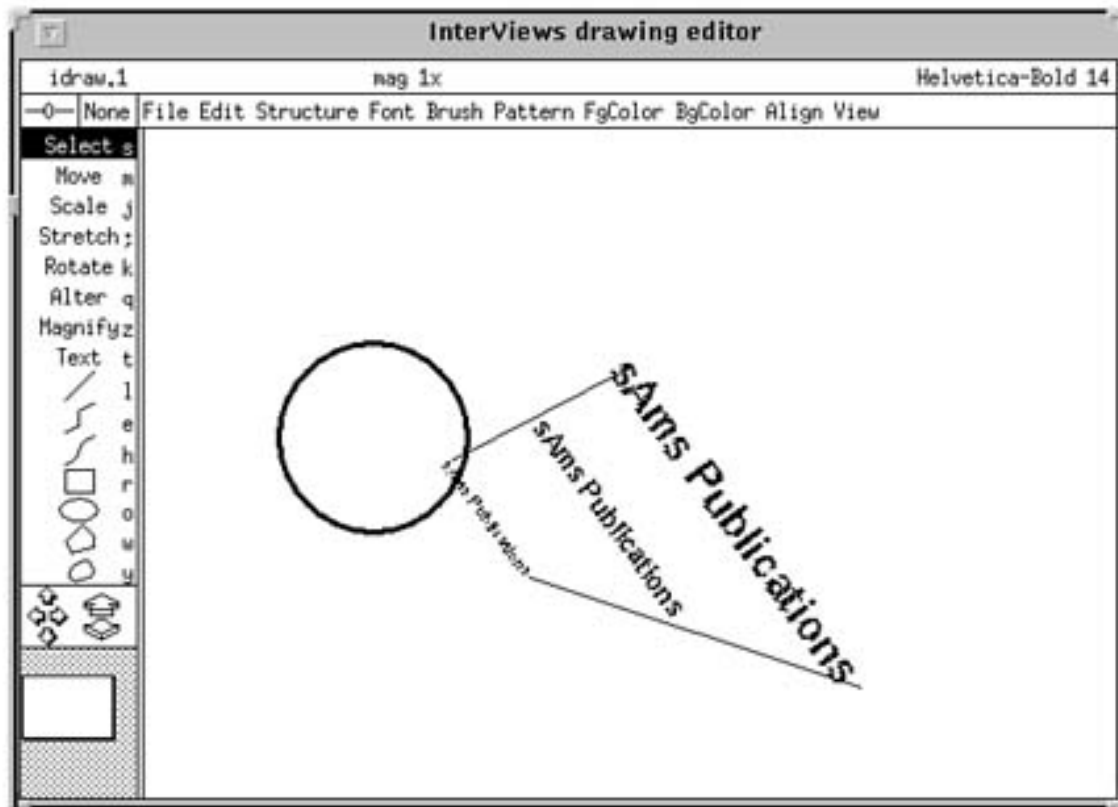


FIGURE 33.3. *The idraw application's main screen.*

Text, Line, Multiline, Open Spline, Ellipse, Rectangle, Polygon, and Closed Spline create new graphics. Each tool works as follows:

- Select is used to select a graphic, unselecting all others.
- A graphic is selected if its handles are visible. Handles are small inverse-video squares which either surround the graphic or demarcate its important points (such as the endpoints of a line).
- If you hold down the Shift key and select an item, it selects the unselected graphic (or unselects the selected graphic) you clicked on, but does not affect the status of other selections.
- Clicking anywhere other than on a graphic unselects everything. You can also drag a rubberband rectangle around a group of graphics to select all of them simultaneously. The right mouse button invokes Select while the mouse is in the drawing area.

- Move lets you move graphics from one spot to another.

TIP: The middle mouse button invokes Move while the mouse is in the drawing area.

- Scale is used to resize graphics about their centers.
- Stretch lets you stretch graphics vertically or horizontally, while tying down the opposite edge.
- Rotate enables you to rotate graphics about their centers, according to the angle between two radii: the one defined by the original clicking point and the one defined by the current dragging point.
- Alter is used to alter a graphic's structure. See the rest of the descriptions of tools to see how Alter affects each tool individually.
- Magnify is used to magnify a portion of the drawing specified by sweeping out a rectangular area. The `idraw` application magnifies the area to occupy the entire screen, if possible.
- Text inserts textual data. Create some text. Left-click to position the first line of text, and then type as much text as you want. The input can be edited using `emacs`-style keystrokes. You can leave text editing mode by typing `ESC` or simply clicking somewhere else. The Alter tool lets you edit the text in an existing text graphic.
- Line creates a line. The Shift key constrains the line to lie on either the vertical or the horizontal axis. You may left-click with the Alter tool on either endpoint of a line to move the endpoint to a new location.
- Multiline lets you create a set of connected lines. The Shift key constrains each segment to lie on either the vertical or the horizontal axis. Each left-click starts a new segment (adds a vertex). Each right-click removes the last vertex added. The middle button finalizes the multiline. The Alter tool lets you move, add, and remove vertices from an existing multiline.
- Open Spline creates an open B-spline. The Shift key constrains each control point to lie on either the vertical or the horizontal axis with the preceding point. Each left-

click adds a control point. Each right-click removes the last control point added. The middle button finalizes the spline. The Alter tool lets you move, add, and remove control points from an existing open spline.

- Ellipse enables you to create an ellipse. The Shift key constrains the ellipse to the shape of a circle. The Alter tool does not affect ellipses.
- Rectangle enables you to create a rectangle. The Shift key constrains the rectangle to the shape of a square. The Alter tool lets you move the rectangle's corners independently to form a four-sided polygon.
- Polygon enables you to create a polygon. The Shift key constrains each side to lie on either the vertical or the horizontal axis. Each left-click starts a new segment (adds a vertex). Each right-click removes the last vertex added. The middle button finalizes the polygon.
- The Alter tool lets you move, add, and remove vertices from an existing polygon.
- Closed Spline enables you to create a closed B-spline in the same manner that you create an open B-spline.

idraws Pull-Down Menus

The pull-down menus File, Edit, Structure, Font, Brush, Pattern, FgColor, BgColor, Align, and View, located above the drawing area, contain commands for editing the drawing and controlling `idraw`'s execution. The File menu contains the following commands to operate on files:

New	Destroys the current drawing and replaces it with an unnamed blank drawing.
Revert	Rereads the current drawing, destroying any unsaved changes.
Open...	Specifies an existing drawing.
Save As	Saves the current drawing in a file with a specific name.
Save	Saves the current drawing in the file from which it came.
Print...	Sends a PostScript version of the drawing to a printer or a file.
Import Graphic...	Can import images from files in the following formats: TIFF; PostScript generated by <code>pgmtops</code> , <code>ppmtops</code> , and <code>idraw</code> ; X bitmap format; and Unidraw format.

Quit Quits `idraw`.

The Edit menu contains the following commands for editing graphics:

Undo	Successive Undo commands undo previous editing operations.
Redo	Successive Redo commands redo subsequent editing operations up to the first operation undone by Undo. Undone operations that have not been redone are lost as soon as a new operation is performed.
Cut, Copy, Paste	Removes the selected graphics from the drawing and places them in a temporary storage area. Copy keeps the original on the drawing area. Paste puts the contents of the storage area, if any, into the location selected by the mouse.
Duplicate	Duplicates the selected graphics and adds the copies to the drawing.
Delete	Destroys all selected graphics.
Select All	Selects every graphic in the drawing.
Flip Horizontal, Flip Vertical	Flips the selected graphics into their mirror images along the horizontal or vertical axis.
90 degrees Clockwise and CounterCW	Rotate the selected graphics 90 degrees clockwise or counterclockwise.
Precise Move..., Precise Scale..., Precise Rotate...	These buttons let you move, scale, or rotate graphics by exact amounts that you type in a dialog box. You can specify movements in pixels, points, centimeters, or inches. Scalings are specified in terms of magnification factors in the horizontal and vertical dimensions. Rotations are always in degrees.

The Structure menu contains the following commands to modify the structure of the drawing, which is the order in which graphics are drawn:

Group	Collects the selected graphics in a newly created picture. A picture is simply a graphic that contains other graphics. Group enables you to build hierarchies of graphics.
Ungroup	Dissolves any selected pictures.
Bring To Front	Brings the selected graphics to the front of the drawing so that they are drawn on top of (after) other graphics.
Send To Back	Sends the selected graphics to the back of the drawing so that they are drawn behind (before) other graphics.

The Font menu contains a set of fonts in which to display text. When you set the current font from the menu, you also set the font of all selected graphics to that font. A font indicator in the upper-right corner displays the current font.

The Brush menu contains a set of brushes with which to draw lines. When you set the current brush from the menu, you also set the brush of all selected graphics to that brush. The nonexistent brush draws invisible lines and non-outlined graphics.

The arrowhead brushes add arrowheads to one or both ends of lines, multilines, and open splines. A brush indicator in the upper-left corner displays the current brush.

The Pattern menu contains a set of patterns with which to fill graphics, but not text. Text always appears solid, but you can use a different color than black to get a halftoned shade. When you set the current pattern from the menu, you also set the pattern of all the selected graphics to that pattern. The nonexistent pattern draws unfilled graphics, while the other patterns draw graphics filled with a bitmap or a halftoned shade.

The FgColor and BgColor menus contain a set of colors with which to draw graphics and text. When you set the current foreground or background color from the FgColor or BgColor menu, you also set the foreground or background color of all the selected graphics. The ON bits in the bitmaps for dashed lines and fill patterns appear in the foreground color; the OFF bits appear in the background color.

A black-and-white printer prints a halftoned shade of gray for any color other than black or white. The brush, pattern, and font indicators all reflect the current colors.

The Align menu contains commands to align graphics with other graphics.

The first graphic selected stays fixed, while the other graphics move in the order that they were selected according to the type of alignment chosen. The last Align command, Align to Grid, aligns a key point on each selected graphic to the nearest point on `idraw`'s grid.

The View menu contains the following commands:

New View	Creates a duplicate <code>idraw</code> window containing a second view of the current drawing. The second view may be panned, zoomed, and edited independently of the first. Any number of additional views may be made in this manner. Changes made to a drawing through one view appear synchronously in all other views of the same drawing. You may also view another drawing in any <code>idraw</code> window via the Open command.
Close View	Closes the current <code>idraw</code> window. Closing the last <code>idraw</code> window is equivalent to issuing a Quit command.

Normal Size	Sets the magnification to unity so the drawing appears at actual size. A What You See Is What You Get (WYSIWYG) display.
Reduce to Fit	Reduces the magnification until the drawing fits entirely within the view.
Center Page	Modifies the view over the center of a 8.5- by 11-inch page as it's printed. Other page sizes are not supported to date.
Orientation	Toggles the drawing's orientation. If the editor was showing a portrait view of the drawing, it now shows a landscape view of the drawing, and vice versa.
Grid on/off	Toggles idraw's grid on or off. When the grid is on, idraw draws a grid of equally spaced points behind the drawing.
Grid Spacing	Enables you to change the grid spacing by specifying one or two values in the units desired (pixels, points, centimeters, or inches). If two values are given (separated by a space), the first specifies the horizontal spacing, and the second specifies the vertical spacing. One value specifies equal horizontal and vertical spacing.
Gravity on/off Toggle	Toggles gravity on or off. Gravity constrains tool operation to the grid, whether or not the grid is visible.

Changes to Xdefaults

You can customize the number of changes that can be undone and the font, brush, pattern, or color menus by setting resources in your Xdefaults database. Each string of the form `idraw.resource:definition` sets a resource. For example, to customize any of the paint menus, set a resource given by the concatenation of the menu's name and the entry's number (such as `idraw.pattern8`) for each entry that you want to override. All menus use the number 1 for the first entry.

You must set resources only for the entries that you want to override, not for all of them. If you want to add entries to the menus, simply set resources for them. However, don't skip any numbers after the end of the menu, because the menu ends at the first undefined resource. To shorten a menu instead of extending it, specify a blank string as the resource for the entry following the last item on the menu. The `idraw` application understands the resources listed in Table 33.1. Table 33.1. The `idraw` resources.

<code>history</code>	Sets the maximum number of changes that can be undone (20 by default).
----------------------	--

<code>initialfont</code>	Specifies the font that is active on startup. Supply a number that identifies the font by its position in the Font menu, starting from 1 for the first entry.
<code>font</code>	Defines a custom font to use for an entry in the Font menu. Give three strings, separated by whitespace. The first string defines the font's name; the second string defines the corresponding print font; and the third string defines the print size. For example, <code>idraw.font3:8x13bold Courier-Bold 13</code> defines the third font entry.
<code>initialbrush</code>	Specifies the brush that is active on startup. Give a number that identifies the brush by its position in the Brush menu, starting from 1 for the first entry. Define a custom brush to use for an entry in the Brush menu. The definition requires two numbers: a 16-bit hexadecimal number to define the brush's line style (each 1 bit draws a dash and each 0 bit produces a gap) and a decimal integer to define the brush's width in pixels. For example, <code>idraw.brush2:ffff 1</code> defines a single pixel-wide solid line. If the definition specifies only the string <code>none</code> , it defines the nonexistent brush.
<code>initialpattern</code>	Specifies the pattern that is active on startup. Give a number that identifies the pattern by its position in the Pattern menu, starting from 1 for the first entry.
<code>pattern</code>	Defines a custom pattern to use for an entry in the Pattern menu. You can specify the pattern from a 16x16 bitmap, an 8x8 bitmap, a 4x4 bitmap, a grayscale number, or the string <code>none</code> . You specify the 16x16 bitmap with sixteen 16-bit hexadecimal numbers, the 8x8 bitmap with eight 8-bit hexadecimal numbers, the 4x4 bitmap with a single 16-bit hexadecimal number, and the grayscale number with a single floating-point number. The floating-point number must contain a period to distinguish itself from the single hexadecimal number, and it must lie between 0.0 and 1.0, with 0.0 corresponding to a solid pattern and 1.0 to a clear pattern. On the printer, the bitmap patterns appear as bitmaps, the grayscale patterns appear as halftoned shades, and the <code>none</code> patterns never obscure any underlying graphics. For example, <code>idraw.pattern8:8421</code> defines a diagonally hatched pattern.

- `initialfgcolor` Specify the foreground color that is active on startup. Give a number that identifies the color by its position in the FgColor menu, starting from 1 for the first entry, `fgcolor`. Define a custom color to use for an entry in the FgColor menu. Give a string defining the name of the color, and (optionally) three decimal numbers between 0 and 65,535 following the name to define the red, green, and blue components of the color's intensity. The intensities override the name; that is, `idraw` looks up the name in a window system database of common colors only if you omit the intensities.
- `initialbgcolor` Specifies the background color that is active on startup. Give a number that identifies the color by its position in the BgColor menu, starting from 1 for the first entry.
- `bgcolor` Defines a custom color to use for the entry in the BgColor menu. The same rules apply to background colors as to foreground colors.

ibuild

The `ibuild` package is an editor that enables you to graphically create a user interface for an Interviews application. The `ibuild` application can save your work as an external file, or as actual C++ code that, when compiled, generates the same user interface that you would create with `ibuild`. You restore a previously saved interface by typing `ibuild` and the interface filename on the command line. (See Figure 33.4.)

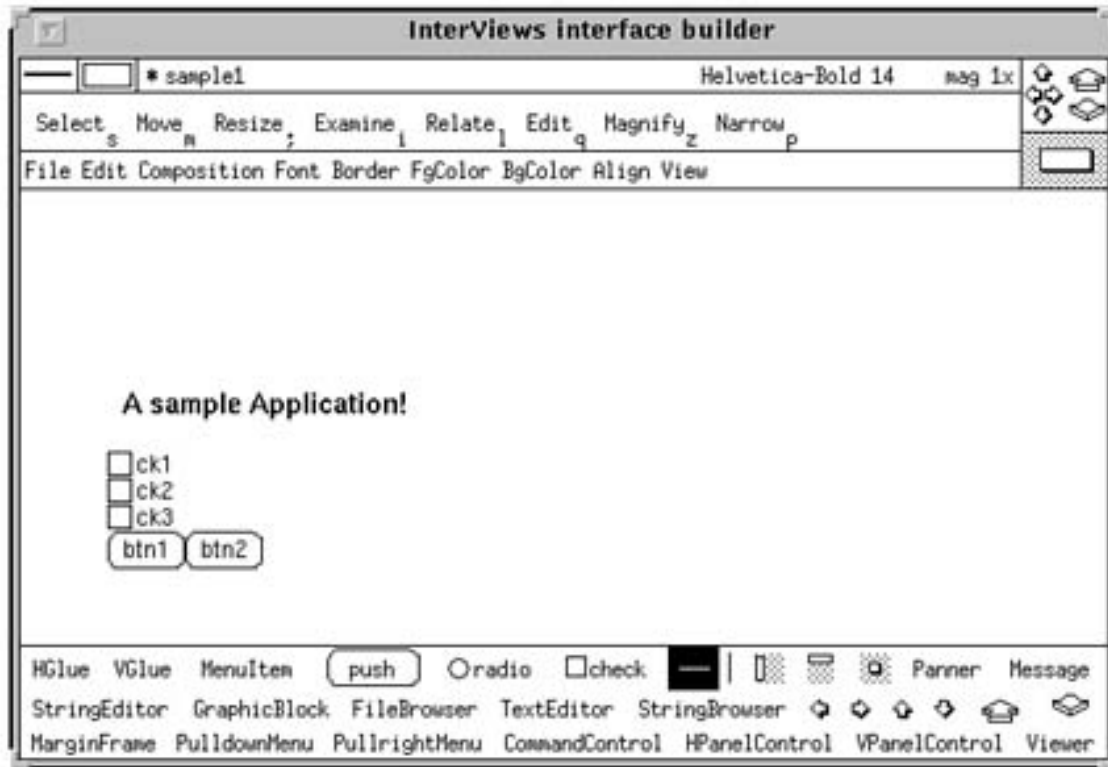


FIGURE 33.4. *The `ibuild` application.*

TIP: The `idraw` and `ibuild` applications are very similar in terms of the semantics for editing and other viewing commands. If you learn one of these two tools, you will feel at home with the other.

The `ibuild` application uses the `TOOLDIR` environment variable for its tools and other parameters. If `TOOLDIR` is not defined, `ibuild` looks for its parameters in the current working directory.

Let's examine the layout of the menus.

The first row of an `ibuild` editor displays information about its brush (border width), foreground and background color, and so on in a fashion similar to that in `idraw`. The next two rows show tools that initiate direct manipulation and pull-down menus that contain commands. A panner on the top-right corner lets you pan and zoom the workspace.

The middle portion of the editor shows the workspace for composing and assembling user

interface components.

The bottom of the editor contains rows of objects in the form of names and iconic drawings that represent familiar Interviews abstractions. Clicking on one tool, dragging, and releasing it in the workspace enables a user to place an instance of the corresponding prototype at the desired location.

Direct manipulation tools lie horizontally along `ibuild`'s second row. You must engage a tool before you can use it. You engage a tool by clicking on its icon, or by typing the character below and to the right of its icon. The icon of the tool that's engaged appears in inverted colors. When it is engaged, you use the tool by clicking the left mouse button in the workspace.

The Select, Move, Resize, Examine, Relate, and Edit tools manipulate text-based user interface components. Magnify makes a part of the view expand to fill the entire view. Narrow enables a user to navigate the structure of a composed interface in the same editor.

The `ibuild` application provides the following tools:

- Select
- Move
- Resize
- Examine
- Info
- Relate
- Edit
- Magnify
- Narrow
- Create

- Tools
- Execute
- Quit

The Select tool is used for selecting an object and unselecting all others. An object is selected if its handles are visible. Handles are small inverse-video squares that surround the object. If you hold down the Shift key, Select extends the selection. It selects the unselected component (or unselects the selected component) you clicked on, but does not unselect other selections. Clicking anywhere other than on a component unselects everything. You may also drag a rubberband rectangle around a group of components to select all of them simultaneously.

TIP: The right mouse button invokes the Select button while the mouse is in the workspace.

- The move button is used for moving objects from one spot to another.

TIP: The middle mouse button invokes the Move button while the mouse is in the workspace.

The Resize tool can change the size of any selected objects. When user interface components from the bottom panel are initially created in the workspace, or when selected components are composed with composition interface objects in the Composition pull-down menu, they are displayed with their natural sizes. Resize simulates the resizing of the application window during running of the application by letting users drag and sweep the selected user interface component.

The Examine tool is used to look at attributes associated with the selected user interface components. User interface components in `ibuild` contain attributes that assist users in designing the layout of the interface, or in customizing the generated code. Typically, the pop-up menu of the Examine tool shows Info and Props entries as options.

Selecting Info causes a component-specific dialog box to pop up. Some of the common attributes shown in the dialog box include the following: Base Class Name, which describes the actual class name in the Interviews library to which the selected component corresponds;

and Class Name, which describes the user-specified class name of the component.

If the value of Class Name is the same as that of the Base Class Name, the library class is used for instantiating the component during code generation, or a set of subclass files are generated to assist the customization of the new class, as described earlier. Member Name describes the member name of the selected component as a member variable of the closest enclosing MonoScene object. If the member name is exported, the MonoScene object can access and manipulate this specific member instance. Canvas Dimensions shows how much workspace (corresponding to actual screen space) in pixels is actually allocated to the selected component. Natural Size, Shrinkability, and Stretchability describe how much workspace the selected component wants to have.

The dialog box associated with the Props entry shows the selected component's instance name and user-defined properties. Property specification takes the form attribute:value.

In order to assist users in locating certain elements with certain member names or instance names, `ibuild` enables a user to probe these attributes by Shift-left-clicking on these elements when they are composed.

The Info entry has a pull-right menu that has class name, member name pairs in it; the pull-right menu associated with the Props entry has class name, instance name pairs.

Examining a GraphicBlock component causes a third Graphics entry to display in the pop-up menu. Selecting this option causes graphics in the GraphicBlock to be transferred to `idraw` for further refinement.

Graphics can be transferred back to `ibuild` by simply saving and quitting `idraw`. Attributes associated with graphics in `ibuild` that are not known to `idraw`, such as member name, are not lost in this process. This option is supported only for backward compatibility. A much better technique is to use the Narrow tool to narrow into GraphicBlocks, which changes `ibuild` into `idraw` in the same window.

NOTE: All machine-generated names are guaranteed to be unique in one session of `ibuild`. This property is lost with user-defined names, and when files generated by different sessions of `ibuild` are compiled together.

The Relate tool provides a direct manipulation interface to semantically connect compatible components. Typically, the end result of the Relate operation is the sharing of attributes

between the related components. For example, relating a scroller to a file browser lets the scroller take the file browser's member name as its scrolling target. Relating two push buttons causes the push buttons to mutually exclude each other when receiving mouse clicks. In this case, the second (destination) push button shares the `ButtonState` defined by the first (source) push button. `Relate` is also very useful for semantically connecting `Unidraw` objects. For instance, an `Editor` needs to be related to all enclosed `Unidraw` objects such as `CommandControls`, `PanelControls`, and `Viewers`. In this case, the simplest way to establish the relations is to relate the `Editor` with the object directly below it in the instance hierarchy, which causes the relations to establish recursively.

The `Edit` tool manipulates text-based components. Engaging the `Edit` tool enables users to perform editing on `MenuItem`, `PushButton`, `RadioButton`, `CheckBox`, `Message`, `stringEditor`, `PulldownMenu`, `PullrightMenu`, `CommandControl`, `HPanelControl`, and `VPanelControl` components.

The `Magnify` tool magnifies a portion of the interface specified by sweeping out a rectangular area. The `ibuild` application magnifies the area to occupy the entire screen, if possible.

The `Narrow` tool enables a user to navigate the structure of composed interfaces in the same editor. Initially, `ibuild` starts at global scope. Engaging the `Narrow` tool and selecting a component causes the hierarchy of the interface to be displayed in a pop-up menu, with the highlighted entry showing the current scope. Selecting a different entry causes the editor to switch to the scope of the selected component. For instance, if a user chooses to narrow into an `HBox`, cutting or pasting components in the workspace now affects only the `HBox`.

NOTE: Leaf-level components don't define new scopes. Narrowing into graphics-related components such as `GraphicBlocks`, `CommandControls`, `PanelControls`, and `Viewers` transforms `ibuild` into `idraw` in the same window, in addition to switching the scope level. This feature is extremely useful because specifying structured graphics objects, class names, and member names can all be done within `ibuild`. Firing up `idraw` using the `Examine` tool is still supported for backward compatibility purposes.

The `Create Tool` creates a prototypical tool of the current interface, and installs it in the bottom tools palette. Users can choose between having the filename, a default icon of the existing interface, or a customized bitmap as the new tool's icon. The newly created tool can then be treated as a library tool for instantiation. This allows domain-specific abstractions to be built and used across sessions of `ibuild`.

The Tools command enables a user to install tools to, or remove tools from, the bottom tools panel. Users can choose to install (or remove) multiple tools from the left (right) stringbrowser by selecting multiple entries and clicking the << Install (or Remove >>) button.

The Execute tool lets the user choose an executable from a file-chooser, and executes the selected file without leaving `ibuild`.

The Quit command exits the current session of `ibuild`.

NOTE: The Edit menu for `ibuild` contains commands that are very similar to those in `idraw`. Refer to the `idraw` section for more details.

CAUTION: Changes made with the Examine tool and Relate tool can be undone.

The Show Glue tool shows HGlue components with horizontal strips and VGlue components with vertical strips. Showing Glue components with strips makes the structure associated with the interface more apparent.

The Hidden Glue tool shows HGlue and VGlue components with their background color. This is useful when the actual appearance of the interface is desired.

The Natural Size tool shows the selected interface components in their natural form, similar to the way they are displayed initially when the generated interfaces are executed.

ibuilds Pull-Down Menus

The pull-down menus File, Edit Composition, Border, FgColor, BgColor, Align, and View lie across the third row. They contain commands that are executed by pulling down the menu and releasing the mouse button on the command, or by typing the character associated with the command.

The File menu contains the following commands to operate on files:

- The New command destroys the current interface and replaces it with an unnamed

blank interface.

- The Revert command rereads the current interface, destroying any unsaved changes.
- The Open command specifies an existing interface file to edit through a FileChooser, which lets you browse the file system easily.
- The Save As command saves the current interface in a file with a specific name.
- The Save command saves the current interface in the file from which it came.

Generating Source Files

The Generate button generates code for the interfaces in `ibuild`. A sequence of dialog boxes pops up to let users check off files that they don't want to be overwritten. The generated files include sets of subclass files and support files. Subclass files have class name suffixes, and support files have filename postfixes. For instance, the filename of a session of `ibuild` is `dialogBox`, and the top-level `MonoScene` object that drives the interface is `DialogBox`.

Support files generated include the following:

- `dialogBox-props`
- `dialogBoxmake`
- `dialogBox-imake`
- `dialogBox-main.c`
- `dialogBox-imake`
- `dialogBox-make`

For each `MonoScene` component in `ibuild`, a set of subclass files is generated. In this case, they include the following:

- `DialogBox.c`

- `DialogBox.h`
- `DialogBox-core.c`
- `DialogBox-core.h`

When you regenerate the code for an application, you are asked in a dialog box whether you want to regenerate these files. As a general rule, all the default files that `ibuild` recommends by checking in their dialog box should be overwritten.

The `dialogBox-props` file contains Interviews properties for customization. Rewrite this file whenever you regenerate.

The `dialogBox-main.c` file contains a prototypical main routine to drive the interface. Rewrite this file whenever you regenerate.

The `DialogBox-core.c` file encapsulates all information about the appearance of the interface. The protected member variables are declared in this file.

The `DialogBox-core.h` file describes what objects are exported to the subclass objects (such as `DialogBox`) for manipulation. This file is, and should be, overwritten whenever you regenerate a new application using `ibuild`.

The `DialogBox.h` and `DialogBox.c` files enable users to implement application-specific interfaces. When you are regenerating code from an `ibuild` session, you are asked whether you want to overwrite these files. If you already have application-dependent code in these files, you don't want to overwrite these files. By default, `ibuild` doesn't overwrite these files.

NOTE: These two files are under the user's control; the core files (`DialogBox-core.[ch]`) are under `ibuild`'s control.

The Composition menu contains commands to modify the structure of the interface. See Table 33.2 for a list of these commands.

Table 33.2. Composition commands.

<i>Command</i>	<i>Action</i>
Dissolve	Dissolves the selected components by deleting the top-level parents and exposing the children of the selected components. Leaf-level components cannot be dissolved. Compose the selected components with the corresponding composition object. The order of selection decides the order in which they are composed.
Hbox	Tiles the selected components left to right in abutting fashion.
Vbox	Tiles the selected components top to bottom in abutting fashion.
Deck	Stacks the selected components on top of each other, with the last selected component on the top.
Frame	Puts a frame around each of the selected components.
ShadowFrame	Puts a shadow frame around each of the selected components.
ViewPort	Puts a viewport around each of the selected components.
MenuBar	Tiles the selected components row by row as in an HBox (horizontal box). It also implements the sweeping effect when MenuItems, pull-down menus, or pull-right menus are the selected components.
Shaper	Redefines the resizing behavior of the selected components. Shaper is an <code>ibuild</code> -generated class, which is useful for overriding the default resizing behavior of library components.

The `Reorder` command reorders the components inside a composition according to the current selection order. For instance, narrowing into an `HBox`, reselecting its components, and executing `Narrow` defines ordering of the components in the `HBox` corresponding to the new selection order.

Similarly, the `Raise` command brings the selected components to the front of the interface so that they are drawn on top of (after) the other components in the interface. For example, if a selected component is raised in a `Deck`, it appears on top of all the components.

The `Lower` command sends the selected components to the back of the interface so that they are drawn behind (before) the other components in the interface.

The `Font` menu contains a set of fonts with which to print text-based components. The default value is the current font from the menu. You also set all the selected components' fonts to that font. A font indicator in the upper-right corner displays the current font.

The `Border` menu contains a set of brushes that are used to set border widths of `Border` and `Frame` components. A border indicator in the upper-left corner displays the current border.

The FgColor and BgColor menus contain a set of colors with which to draw components and text. When you set the current foreground or background color from the FgColor or BgColor menu, you also set all the selected components' foreground or background colors.

The Align menu contains commands to set the alignment of the message on Message-based components. Examples of these components include pull-down and pull-right menus, MenuItem components, and Message components. The effects of alignment are more apparent when the selected components are resized.

Subclasses

Subclasses enable you to introduce user-defined objects for customizing your interfaces. These are abstraction mechanisms that break down complicated user interfaces into more manageable and reusable subcomponents, which are more amenable to user customization. Without subclass objects, the generated code consists of static functions that assemble the library user interface elements into complete interfaces.

The subclasses that you can work with are as listed here :

- MonoScene Subclass
- Dialog Subclass
- Editor Subclass

Member names associated with the components are therefore useless.

With subclass objects, children of a subclass instance become the instance's interior definition. If the interior components are exported, they become member variables of the subclass instance.

Selecting MonoScene Subclass causes each selected component to be enclosed in a MonoScene subclass object.

Selecting Dialog Subclass is similar to selecting MonoScene Subclass, except that it provides features of the Dialog class in the library as well.

Editor Subclass is useful when the end application is a domain-specific editor. Objects kept by an editor, such as the KeyMap, Selection, Component, and ControlState, can all be

subclassed to define domain-specific behavior.

Code Generation

Conceptually, components created in the workspace are instances of the actual Interviews and Unidraw counterparts. All separate components (uncomposed and composed components without common parents) generate separate windows after code generation. For example, if the user-specified filename associated with an `ibuild` session is `hello`, the files `hello-imake`, `hello-make`, `hello-props`, and `hello-main.c` are generated.

- `hello-imake` and `hello-make` are an `imakefile` and a `makefile` generated by `ibuild` using `ibmkmf`, respectively.
- `hello-props` contains attribute:value pairs required by Interviews to associate properties to user-interface objects on a per instance basis.
- `hello-main.c` contains a generated `main()` routine to instantiate the interfaces. If no `MonoScene` objects exist in `ibuild`, static functions are generated in `hello-main.c` to assemble the interface components. `MonoScene` (or `MonoScene` subclass) objects are user-defined abstractions to decompose complex user interfaces into simpler, higher-level elements, which are described later.

All user interface components can be subclassed by simply renaming their Class Name attribute to be different from the Base Class Name, using the Examine tool and selecting the Info entry.

A set of four files is generated for each subclassed component in `ibuild`. If the Class Name of a subclassed component is `Displayler`, and the Base Class Name is `stringEditor`, the following files are created: `Displayler.h`, `Displayler.c`, `Displayler-core.h`, and `Displayler-core.c` as the output of the Generate command.

For the previous set of files, `Displaylercore` is a subclass of `stringEditor`, and `Displayler` is a subclass of `Displaylercore`. The `-core.*` postfixed files are core files that are under `ibuild`'s control, and you should not modify them. These core files contain enhanced widget definitions to accommodate library deficiencies and provide a more convenient user model.

`Displayler.h` and `Displayler.c` are subclass files that are provided for customization,

which typically involves redefinition of some virtual functions defined by the base class.

The `ibuild` application also allows the creation of new user interface abstractions by providing `MonoScene` subclass, `Dialog` subclass, and `Editor` subclass composition mechanisms. Typically, the topmost composition of a completed user interface component is a `MonoScene` subclass object. All enclosed components of a subclass object can be thought of as its members. For instance, if a `Dialog` subclass instance called `Informer` is used to wrap (abstract) the previous interface, `Informer.h`, `Informer.c`, `Informer-core.h`, and `Informer-core.c` are generated. In this case, `Informercore` is a subclass of `Dialog`, and `Informer` is a subclass of `Informercore`. Like `Displayer`, `Informer-core.h` and `Informer-core.c` are under `ibuild`'s control; `Informer.h` and `Informer.c` are for user customization. In addition to providing enhanced widget definitions, `Informer-core.c` also contains definitions of its appearance by instantiating its member components. `Informer-core.h` provides an interface to `Informer.h` where the exported members can be selected by using the `Examine` tool, as explained later.

Additional Resources in Xdefaults

You must set resources only for the entries that you want to override, not for all of them. If you want to add entries to the menus, simply set resources for them. However, don't skip any numbers after the end of the menu, because the menu ends at the first undefined resource. To shorten a menu instead of extending it, specify a blank string as the resource for the entry following the last item in the menu.

The `ibuild` application understands the resources listed in Table 33.3, in addition to those specified in `idraw`.

Table 33.3. Additional resources for `ibuild`.

<i>Resource</i>	<i>Action</i>
<code>initialborder</code>	Specifies the border that is active on startup. Give a number that identifies the border by its position in the <code>Border</code> menu starting from 1 for the first entry. Border specification is similar to brush specification in <code>idraw</code> , except only solid brushes should be used because they are used to set border widths of <code>Border</code> and <code>Frame</code> components.

border i Defines a custom border to use for the *i*th entry in the Border menu. Unlike normal brush specification, the first 16-bit hexadecimal number should always be 0xffff to indicate solid brush. The second hexadecimal number should give the desired border width in pixels. Border specification affects only certain interface objects, as described earlier

Building an Application with ibuild

Let's start with a simple application, shown in Figure 33.5. You will build an interface with three check buttons, two push buttons, and a label. The label was created with the Message tool.



FIGURE 33.5. *The ibuild sample application.*

Collect the check buttons by selecting them and grouping them with the Composition/VBox selection. For the push buttons, use the HBox selection to group them horizontally.

All components of the application were glued together, using the VBox item, to create one

big application. Because this is one screen, attach the Monoscreen composition property to all objects in this application in order to allow all portions of the application screen to be filled with blanks, which are not explicitly covered by an object.

Now, save the application with the File/Save As button. See Figure 33.6 for this dialog box. After saving this application, use the File/Generate option to generate all the source, makefiles, and property files for this application. Now you can build the application as shown in Figure 33.7.



FIGURE 33.6. *Saving the sample application.*

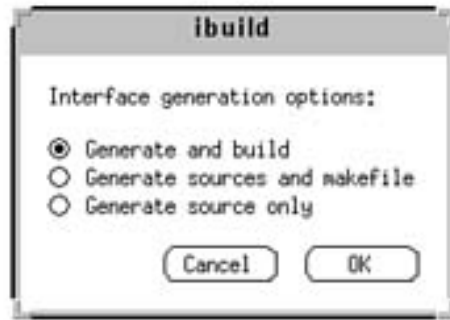


FIGURE 33.7. *Generating files for the sample application.*

The following files are created in your current working directory after you are done:

`iview/sample1`

`iview/sample1-imake`

`iview/sample1-main.c`

`iview/sample1-main.o`

`iview/sample1-make`

`iview/sample1-props`

NOTE: If you do not have the `TOOLDIR` variable defined, you also see a lot of files with the extension `Tool`. Do not delete them unless you point `TOOLDIR` to the `/usr/Interviews Tools` directory. The `Tools` files provide some interesting insights into how the interface tools work for Interviews.

The version of `ibuild` on this distribution creates a makefile that fails around line 182, when you try to create the application with a `make` command. The error is that two makefile variables, `SRCS` and `OBJS`, create assignments that do not have a trailing `\` for each source file to create. Edit the makefile manually to add these trailing backslashes, or concatenate all the names into one line.

So, the lines

`SRCS =`

`_MonoScene_5.$(CCSUFFIX)`

`_MonoScene_5-core.$(CCSUFFIX)`

`sample1-main.$(CCSUFFIX)`

`OBJS =`

`_MonoScene_5.o`

```
_MonoScene_5-core.o
```

```
sample1-main.o
```

become the following:

```
SRCS =    _MonoScene_5.$(CCSUFFIX) \

        _MonoScene_5-core.$(CCSUFFIX) \

        sample1-main.$(CCSUFFIX)

OBJS = _MonoScene_5.o \

        _MonoScene_5-core.o \

        sample1-main.o
```

Figure 33.8 shows the makefile for this application generated by the build.



```

# -----
# DO NOT EDIT -- generated by ibmkuf
SPECIAL_IMAKEFLAGS =
    -f sample1-imake -s sample1-make -DUseInstalled -DTurnOptimizingOn=0
CCSUFFIX = c
SRCS =
    sample1-main,$(CCSUFFIX)
OBJS =
    sample1-main.o
AOUT = sample1.exe

DEPLIBUNIDRAW =
DEPLIBIV =
DEPLIBXEXT =
DEPLIBX11 =
DEPLIBM =

LIBDIRPATH = -L$(LIBDIR)
:182

```

FIGURE 33.8. *A sample makefile for this application.*

NOTE: Do not forget to edit the makefile to create the correct arguments for the SRCS and OBJS macros.

Now, you can create the application `sample1-exe` with the following command:

```
$ make -f sample1-make sample1
```

This make command spews a lot of messages from the GNU g++ compiler. You should not see any error messages if you have edited your `sample1-make` file as described earlier. After you have created the application, you can execute this application with this command:

```
$ sample1
```

The result of this application is shown in Figure 33.5.

CAUTION: If you do assign the Monoscreen composition to the final VBox or HBox of this application, your application will not fill in any blank areas on the main screen. The result is a semitransparent window of whatever happens to be on the screen at the time and the rest of your application's objects.

CAUTION: Do not forget to group all of your objects together, or each object will have its own independent window.

Adding Interactors

The last order of business for your sample application is to add code to take actions when a button is pressed. This step enables you to attach your own code to your newly created application. A GUI by itself really does not serve any purpose, does it? The code that takes action is called an Interactor.

Add another button to the application you just created, and attach an Interactor to it. Click with the left button on this button, and hold it down. You see two menu items pop up on the mouse cursor. While holding the left button down, move the cursor to the right of this menu to get another list of items, including one titled PushButton. Move the cursor to this item and go left. You are presented with a dialog box (shown in Figure 33.9) showing all the attributes of this push button.

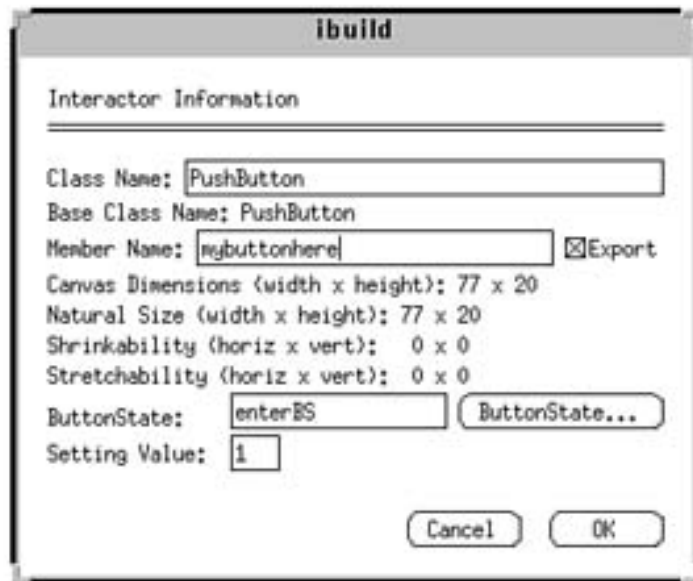


FIGURE 33.9. *The PushButton attributes.*

Name this push button by typing `mybuttonhere` in the Member Name box, and check the Export button.

Next, specify the `ButtonState` instance to use by typing `enterBS` in the `ButtonState` dialog. Now click on the `ButtonState` button to create this instance. See Figure 33.10 for this dialog box.

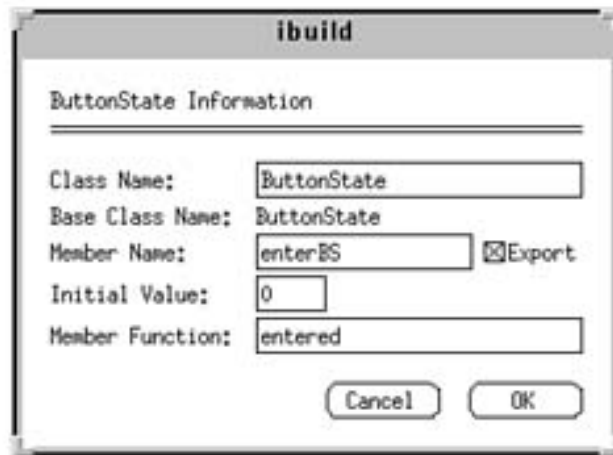


FIGURE 33.10. *Creating the `ButtonState` instance.*

The finished application is shown in Figure 33.11.

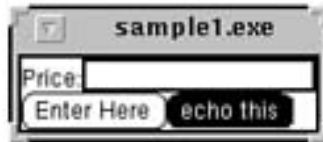


FIGURE 33.11. *The finished application.*

Make sure that the `ButtonState` is exported, and make entered the name of the function that is called when this button is pressed. Click OK to accept this.

Now, set the setting value of this push button to 1. Click OK to accept it.

Press the left mouse button on the string utility, and then press the left mouse button again on the Info button to get the info for `stringEditor`. As you did with the push button, create an exported member function called `stringEditor`. Click OK to accept and dismiss this dialog.

Use the Relate tool to connect the push button to the `stringEditor`.

Click on the push button, and hold the left mouse button down to select the `PushButton` item from the pop-up menu that appears. Let go of the mouse button. A line is drawn from this button to your cursor.

Now, move the cursor to the `stringEditor`, and press the left mouse button again. This

brings up another pop-up menu from which you select `stringEditor` as your target.

The `stringEditor` is now the target, and the push button is the source. To see whether they are related, use the Examine tool to ensure that the `ButtonState` name of `stringEditor` is now `enterBS`.

NOTE: The order in which you select the relationship is important. The target (`stringEditor`) receives the `ButtonState` from the source (`pushButton`).

The final step is to create the classes that wrap the core around another class. Remember that we created a `MonoScreen` class as the topmost class for this application. Using the Examine tool, get the info on the `Mono_*` class for this application. When you get the dialog box for this application, change the Class Name to something else, like `Sample1`. Click OK to save.

Now, generate the code for this application. You see the following four files listed:

`iview/Sample1-core.c`

`iview/Sample1-core.h`

`iview/Sample1.c`

`iview/Sample1.h`

Do not edit the `*-core` files. Edit the `Sample1.c` and `Sample1.h` files to create your own functions. The `Sample1.h` file looks like the one shown in Listing 33.2. The `Sample1.c` file is shown in Listing 33.3.

Listing 33.2. The Sample 1.h file.

```
#ifndef Sample1_h
```

```
#define Sample1_h

#include "Sample1-core.h"

class Sample1 : public Sample1_core {

public:

    Sample1(const char*);

    virtual void entered();

};

#endif
```

Listing 33.3. The Sample1.c file.

```
#include <InterViews/streditor.h>

#include <InterViews/button.h>

#include "Sample1.h"

#include <IV-2_6/_enter.h>

Sample1::Sample1(const char* name) : Sample1_core(name) {}

void Sample1::entered() {

    /* unimplemented */

}
```

Modify the `Sample1.c` file to look like Listing 33.4.

Listing 33.4. Adding code to the callback function.

```
#include <InterViews/streditor.h>

#include <InterViews/button.h>

#include "Sample1.h"

#include <IV-2_6/_enter.h>

#include <stream.h>

Sample1::Sample1(const char* name) : Sample1_core(name) {}

void Sample1::entered() {
```

```
/* Get user input */

int value;

enterBS->GetValue(value);

if (value != 0) {

    cout << stringEditor->Text() << "!\n";

    cout.flush();

    enterBS->SetValue(0);

}

}
```

Create the application with the following command:

```
$ make -f sample1-make sample1.exe
```

After a long time (depending on the speed of your PC), you will have a `sample1.exe`

application in your directory. Run this program. As you press the center button, you see the `cout` command execute and print to the standard output.

NOTE: Here's a note about creating Interviews applications from scratch using C++ classes only. Generally, you use the `ibuild` program to create all your applications with the Interviews libraries. However, there may be times when you want to create small, quick applications using just the base Interviews classes. This step may prove not to be worth the hassle after you are done, because an `ibuild` application may actually get you quicker results. I have found it easier to just use `ibuild` than to try to create applications from scratch. Some examples of such from the bottom up applications are available via ftp from the site `leland.stanford.edu`.

Summary

This has been a very short chapter on a very complicated topic: creating Graphical User Interface front ends using C++ and Interviews. The code required for Interviews is based on reusable components. The files for this package are located in the `/usr/interviews` directory.

You can use `idraw` to create drawings of icons and learn about the interface tools available to you. Learning to use this tool will help you work with `ibuild`.

The `ibuild` application is an Interviews tool that enables you to interactively create C++ applications. `ibuild` also generates the C++ code for you to run the `g++` compiler, or to actually create the makefiles and related source code. The makefiles include a minor bug around line 182 where lines are not connected correctly with a backslash.

The tools available in `ibuild` can be grouped together using the `HBox`, `VBox` and other tools. You can import other functionality for creating Dialogs, File selection boxes, and `MonoScenes`. By wrapping the application main classes in your own classes, you can add your own C++ (or C) code to any newly created application. You have to be careful which files you edit, because most of the core files are re-created when you modify the interface and regenerate the code. The core subclasses that you create enable you to create more functionality for the core class without having to modify the core GUI classes.

- [- 34 -](#)

- [Motif Programming](#)

- [Writing Motif Applications](#)
- [Naming Conventions](#)
- [TIP](#)
- [Writing Your First Motif Application](#)
- [NOTE](#)
- [Listing](#)
- [34.1. A sample Motif application.](#)
- [TIP](#)
- [Compiling This Application](#)
- [The Widget Hierarchy](#)
 - [Core](#)
 - [XmPrimitive](#)
 - [XmManager](#)
- [The Label Widget](#)
- [Listing](#)
- [34.2. How to use a Label Widget.](#)
- [Strings in Motif: Compound Strings](#)
- [TIP](#)
- [The XmPushButton Widget Class](#)
- [TIP](#)
- [TIP](#)
- [The XmToggleButton Widget Class](#)
- [Listing](#)
- [34.3. Using ToggleButton in Motif.](#)
- [Convenience Functions](#)
- [Listing](#)
- [34.4. Sample convenience function for creating a label on a form.](#)
- [Listing](#)
- [34.5. Creating a label.](#)
- [The List Widget](#)
- [Listing](#)
- [34.6. Using List Widgets.](#)
- [XmScrollBar](#)
- [Listing](#)
- [34.7. Using the Scale Widget.](#)
- [Text Widgets](#)
- [XmBulletinBoard Widgets](#)
- [XmRowColumn Widgets](#)
- [Listing](#)
- [34.8. Using RowColumn Widgets.](#)
- [XmForm Widgets](#)
- [TIP](#)
- [TIP](#)
- [Designing Layouts](#)

- [Listing](#)
- [34.9. Setting up a simple hierarchy.](#)
- [Menus](#)
 - [Pop-Up Menus](#)
- [Listing](#)
- [34.10. Setting up pop-up menus.](#)
 - [The Menu Bar](#)
- [Listing](#)
- [34.11. Creating a menu bar.](#)
- [Listing](#)
- [34.12. Creating menu bars with pull-down menu items.](#)
 - [The Options Menu](#)
 - [Accelerators and Mnemonics](#)
- [TIP](#)
- [Dialog Boxes](#)
- [Listing](#)
- [34.13. Code fragment to confirm quit command.](#)
 - [Modes of a Dialog Box](#)
- [Events](#)
 - [Expose Events](#)
 - [Pointer Events](#)
 - [Keyboard Events](#)
 - [Window Crossing Events](#)
 - [Event Masks](#)
- [Listing](#)
- [34.14. Tracking a pointer.](#)
- [Managing the Queue](#)
- [TIP](#)
 - [Work Procedures](#)
 - [Using Timeouts](#)
- [Listing](#)
- [34.15. Setting up cyclic timers.](#)
 - [Handling Other Sources](#)
- [NOTE](#)
- [The Graphics Context](#)
- [Drawing Lines, Points, Arcs, and Polygons](#)
 - [Drawing a Line](#)
- [Listing](#)
- [34.16. Drawing lines and points.](#)
 - [Drawing a Point](#)
 - [Drawing an Arc](#)
- [Using Fonts and FontLists](#)
- [The X Color Model](#)
- [Listing](#)
- [34.17. Convenience function for getting colors.](#)
- [Pixmap, Bitmap, and Image](#)

- [Summary](#)

- 34 -

Motif Programming

by Kamran Husain

IN THIS CHAPTER

- Writing Motif Applications
- Designing Layouts
- Menus
- Dialog Boxes
- Managing the Queue
- The Graphics Context
- Drawing Lines, Points, Arcs, and Polygons
- Using Fonts and FontLists
- Pixmaps, Bitmaps, and Images

This chapter will cover the following topics:

- The basics of writing Motif applications for Linux
- Special naming conventions in Motif and X
- Writing and compiling your first Motif application
- Revisiting the Widget hierarchy
- Using labels and strings in Motif
- Using various common Widgets
- Designing layout
- Using menus

- Dialog boxes
- Event handling and other sources of input
- Colors in X
- Drawing lines, points, arcs, and polygons

A question you might be asking is "Why include a topic on a development system that you have to pay for, when just about everything for Linux is free?" Well, if you want to develop any applications for the Common Desktop Environment (CDE), you should know how to program Motif applications. Linux is a mature enough system to enable you this luxury of building portable applications. (Plus, the \$150 or so you pay for the Motif license will well pay for itself if you can do the work at home on your Linux system rather than commuting!)

Writing Motif Applications

This chapter introduces you to writing Motif applications. The information here will not be limited to writing applications for Linux alone, because the concepts in this chapter can be applied to other UNIX systems as well.

In programming Motif applications, you have to get used to programming in an event-driven environment. A typical C application runs from start to finish at its own pace. When it needs information, the application looks for this information from a source such as a file or the keyboard and (almost always) gets the information as soon as it asks for it. If the information is not available when the application asks for it, the application either waits for it or displays an error message. Also, the order of the incoming data is important for such applications; pieces of data that are out of sequence may cause the application to behave strangely.

In the case of event-driven programming, an application must wait for events on an input queue. The queue orders all incoming events in the order they are received. The first message to come in from one end of a queue is the first one to leave the queue. (Such queues are often called FIFOs, for First In, First Out.) An event can be anything from a mouse click to a timeout notification.

Because events can come in at any time, and in no predefined order, they are referred to as asynchronous events. That is, the order and time of arrival of each event is not deterministic. The application must wait for an event to occur and then proceed based on that event. Thus the term event-driven programming.

In the case of the X Window system, each X Window application has one input queue for all of its incoming events. The application must wait for events on this input queue. Similarly, a server waits for an event from a client and then responds based on the type of event received. This event handling and other aspects of X programming are handled by a toolkit called `XtIntrinsics`, or `Xt` for short.

In `Xt`, an application will typically run in a loop forever. This loop is called an event loop. An application enters the loop by calling a function `XtAppMainLoop()`. While in this event loop, an application will always wait for an event. When the application receives an event, the application handles the event itself or almost always "dispatches" the event to a window or `Widget`.

A `Widget` registers functions that it wants called when a type of event is received. This function is called a callback function. In most cases, a callback function is independent of the entire application. For example, some `Widgets` will redraw themselves when a pointer button is clicked in their display area. In this case, they would register a redraw callback function on a button click.

Xt also supports actions, which enable applications to register a function with Xt. An action is called when one or more sequences of specific event types are received. For example, pressing Ctrl-X would call the `exit` function. The mapping of the action to an event is handled via a translation table within Xt. Functions that handle specific events are referred to as event handlers.

Look at Figure 34.1 to see how the toolkit exists with Motif. As you can see from the connections in the figure, an application can get to the core Xlib functions through three means: via Motif, via the Xt library, or directly. This flexible hierarchy gives you many options when developing Motif applications because you are at liberty to take full advantage of all functions in all three libraries.

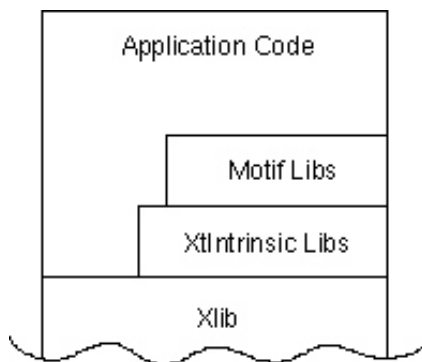


Figure 34.1. *The toolkit hierarchy for X, Xt, and Motif.*

Naming Conventions

By default, most Xlib functions begin with the letter X, but you should not always rely on this being true for all functions. Several macros and functions in the X Window system do not begin with X. For example, the names `BlackColor` and `WhiteColor` are not macros. In general, though, if a name in Xlib begins with X, it's probably a function. If a name begins with a capital letter (A through Z), it's a macro.

With Xt, the naming conventions get better, but only slightly. In Xt, macros are not differentiated from functions in any way.

TIP: Do not rely on the name of a toolkit function to give you information about whether it's a macro. Read the manual to be absolutely sure.

In Motif, almost all declarations begin with Xm. Therefore, XmC refers to a class, XmR refers to a resource, XmN refers to a name, and XtN refers to Xt resources used by Motif. Declarations ending with the words `WidgetClass` define the base class for a type of `Widget`. A few conventions to remember about parameters for most Xlib functions are

- Width is always to the left of height
- X is always to the left of y
- Source is always to the left of destination
- Display usually is the first parameter

With practice, you will be able to identify the type of parameters to pass and which toolkit a function belongs to, and be able to make some educated guesses as to what parameters an unknown function might expect.

Writing Your First Motif Application

Let's look at the basic format for a Motif application, shown in Listing 34.1. (I added line numbers for your benefit.) I will discuss this application in detail. You will build other Motif applications based on the structure in this particular application as you progress through this chapter.

NOTE: The line numbers are for reference only.

Listing 34.1. A sample Motif application.

```
1      /*

2      ** This is a typical Motif application with one button that

3      ** exits it.

4      */

5      #include <X11/Intrinsic.h>

6      #include <Xm/Xm.h>

7      #include <Xm/Form.h>

8      #include <Xm/PushB.h>

9      void bye(Widget w, XtPointer clientdata, XtPointer calldata);
```

```
10     int main(int  argc, char **argv)

11     {

12         Widget top;

13         XtAppContext app;

14         Widget aForm;

15         Widget aButton;

16         Arg    args[5];

17         /**

18         *** Initialize the toolkit.

19         **/

20         top = XtAppInitialize(&app, "KBH", NULL, 0, (Cardinal *)&argc,

21                               argv, NULL, args, 0);

22         /**
```



```
23      *** Create a Form on this top level Widget. This is a nice Widget

24      *** to place other Widgets on top of.

25      **/

26      aForm = XtVaCreateManagedWidget("Form1",

27          xmFormWidgetClass, top,

28          XmNheight,90,

29          XmNwidth,200,

30          NULL);

31      /**

32      *** Add a button on the form you just created. Note how this Button

33      *** Widget is connected to the form that it resides on. Only

34      *** left, right, and bottom edges are attached to the form. The

35      *** top edge of the button is not connected to the form.

36      **/
```

```
37     aButton = XtVaCreateManagedWidget("Push to Exit",

38         xmPushButtonWidgetClass, aForm,

39         XmNheight, 20,

40         XmNleftAttachment, XmATTACH_FORM,

41         XmNrightAttachment, XmATTACH_FORM,

42         XmNbottomAttachment, XmATTACH_FORM,

43         NULL);

44     /**

45     *** Call the function "bye" when the PushButton receives

46     *** an activate message; i.e. when the pointer is moved to

47     *** the button and Button1 is pressed and released.

48     **/

49     XtAddCallback( aButton, XmNactivateCallback,

50         ^bye, (XtPointer) NULL);
```

```
51      XtRealizeWidget(top);

52      XtAppMainLoop(app);

53      return(0);

54  }


55      void  bye(Widget w, XtPointer clientdata, XtPointer calldata)

56      {

57          exit(0);

58      }
```

The listing shows an application in which a button attaches itself to the bottom of a form. See Figure 34.2.



Figure 34.2. *The output of Listing 34.1 (L34_1 . c).*

No matter how you resize the window, the button will always be on the bottom. The application does the following things in the order listed:

1. Initializes the toolkit to get a shell Widget.
2. Makes a Form Widget.
3. Manages all Widgets as they are created.
4. Makes the Button Widget on top of the Form Widget.
5. Attaches a callback function to the button.
6. Realizes the Widget (that is, makes the hierarchy visible).
7. Goes into its event loop.

Let's look at the application in more detail. The `include` files in the beginning of the listing are required for most applications. Notably, the two files shown in lines 5 and 6 are required for just about any Motif application you'll ever write.

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

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

These two lines declare the definitions for `XtIntrinsics` and `Motif`, respectively. In some systems, you may not require the first inclusion, but it's harmless to put it in there because multiple inclusions of `Intrinsic.h` are permitted. In addition, each Motif Widget requires its own header file. In Listing 34.1, the Widgets `Form` and `PushButton` are included via statements in lines 7 and 8:

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

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

The variables in the program are declared in lines 12 through 16:

```
Widget top;
```

```
XtAppContext app;
```

```
Widget aForm;
```

```
Widget aButton;
```

```
Arg args[5];
```

The `top`, `aForm`, and `aButton` represent Widgets. Even though their functions are different, they can all be referred to as Widgets.

The `XtAppContext` type is an opaque type, which means that a Motif programmer does not have to be concerned about how the type is set up. Widgets are opaque types as well; only the items that are required for the programmer are visible.

The first executable line of the program calls the `XtAppInitialize()` function (in line 20) to initialize the Xt toolkit and create an application shell and context for the rest of the application. This value is returned to the Widget `top` (for top-level shell). This Widget will provide the interface between the window manager and the rest of the Widgets in this application.

The application then creates a Form Widget on this top-level Widget. A Form Widget is used to place other Widgets on top of itself. It is a Manager Widget because it "manages" other Widgets.

There are two steps required for displaying a Widget: First you have to manage it (with `XtVaCreateManagedWidget()`) and then you have to realize it (with `RealizeWidget()`).

Managing a Widget enables it to be visible. If a Widget is unmanaged, it will never be visible. By managing a Widget, the program gives the viewing control over to the windowing system so it can display the Widget. If the parent Widget is unmanaged, any child Widgets remain invisible, even if they are managed.

Realizing a Widget actually creates all the subwindows under an application and displays them. Normally, only the top-level Widget is realized after all the Widgets are managed. This call will realize all the children of this Widget.

Note that realizing a Widget takes time. A typical program will manage all the Widgets except the topmost one. This way the application will only have to realize the topmost Widget when the entire tree has to display only the topmost parent. You have to realize a Widget at least once, but you can manage and "unmanage" Widgets as you want to display or hide them.

You can always create and manage a Widget to call `XtCreate` and `XtManageChild` in two separate calls. However, the samples in this chapter will use a single call to create and manage a Widget: `XtVaCreateManagedWidget`.

Note the parameters in this call to create the Form Widget shown in lines 26 through 30:

```
aForm = XtVaCreateManagedWidget( "Form1",  
  
    xmFormWidgetClass, top,  
  
    XmNheight, 90,  
  
    XmNwidth, 200,  
  
    NULL );
```

The first parameter is the name of the new Widget. The second parameter describes the class of the Widget being created. Recall that this is simply the Widget name sandwiched between `xm` and `WidgetClass`. So, in this case, it is `xmFormWidgetClass`. Note the lowercase `x` for the class pointer. This class pointer is declared in the header files included at the beginning of the file, `Form.h`.

TIP: As another example, the class pointer for a Label would be called `xmLabelWidgetClass` and would require the `Label.h` file. Motif programmers have to be especially wary of the case of all variables.

The next argument is the parent Widget of this new Widget. In this case, `top` is the parent of `Form1`. The `top` Widget is returned from the call to `XtAppInitialize`.

The remaining arguments specify the parameters of this Widget. In this case you are setting the width and height of

the Widget. This list is terminated by a NULL parameter.

After the form is created, a button is placed on top of it. A Form Widget facilitates placement of other Widgets on top of it. In this application you will cause the button to "attach" itself to the bottom of the form. The following three lines (40-42) attach themselves to the form:

```
XmNleftAttachment, XmATTACH_FORM,
```

```
XmNrightAttachment, XmATTACH_FORM,
```

```
XmNbottomAttachment, XmATTACH_FORM,
```

The class of this button is included in the `PushButton.h` file and is called `xmPushButtonWidgetClass`. The name of this Widget is also the string that is displayed on the face of the button. Note that the parent of this button is the `aForm` Widget. The hierarchy is as follows:

```
top -> is the parent of aForm -> is the parent of -> aButton.
```

The next step is to add a callback function when the button is pressed. This is done with the call to `XtAddCallback`, as shown in the following:

```
XtAddCallback( aButton, XmNactivateCallback, bye, (XtPointer) NULL);
```

Here are the parameters for this call:

- `aButton` is the `PushButton` Widget.
- `XmNactivateCallback` is the action that will trigger a call to this function.
- `bye` is the name of the function that will be called when the action is triggered. (You should declare this function before making this function call, or you will get a compiler error.)

- `NULL` is a pointer. This pointer could point to some structure meaningful to function `bye`.

This will register the callback function `bye` for the Widget. Now the topmost Widget, `top`, is realized. This causes all managed Widgets below `top` to be realized. The application then goes into a loop that processes all incoming events.

The `bye` function of this program simply exits the application.

Compiling This Application

Now comes the tough part of compiling this application into a working application. You will use the `gcc` compiler that comes with Linux for this purpose.

First, check the location of the libraries in your system. Check the `/usr/lib/X11` directory for the following libraries: `libXm.a`, `libXt.a`, and `libX11.a`. If possible, use the shared library versions of these libraries with `.so` extensions followed by some numbers. The advantage of using shared libraries is a smaller Motif application; a typical application like the one you've been working on can be up to 1MB in size because of Motif's overhead.

The disadvantage of shared libraries is that your end user may not have the correct version of the library in his path. This does annoy some end users, especially if no fast method of acquiring this resource is available. Also, shipping a shared library with your application may cause you to pay some licensing fees to the original library vendor. From a programmer's perspective, shared libraries are sometimes impossible to use with your debugger. Of course, if your debugger supports them, use it. Check your compiler documentation. In most cases, if you intend to use shared libraries, use the static versions to do your debugging and testing, and then compile the shared version. Always check your vendor's licensing agreement for details on how to ship shared libraries.

The application can be compiled with this command:

```
gcc 134_1.c -o list1 -lXm -lXt -lX11
```

The program can be run from a command line if you create a script file:

```
$ cat mk
```

```
gcc $1.c -o $1 -lXm -lXt -lX11
```

and pass it just the filename without the extension. The best way is to create a `makefile`, but this script file will work with the examples in this text. So, to compile `134_1.c`, you would use the script as follows:

```
$ mk 134_1
```


You should see the output shown in Figure 34.2 on your screen. Your application is the one with 134_1 in its frame.

The Widget Hierarchy

The Motif Widget set is a hierarchy of Widget types. (See Figure 34.3.) Any resources provided by a Widget are inherited by all its derived classes. Consider the three most important base classes: Core, XmPrimitive, and XmManager.

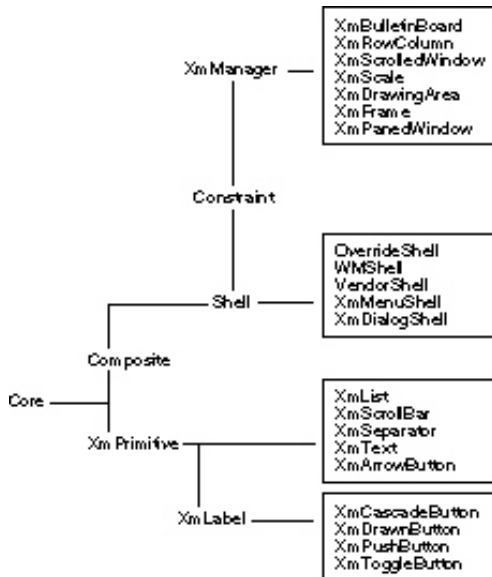


Figure 34.3. *The partial Motif hierarchy.*

Core

The Core Widget class provides the basis for all classes. It provides at least the following resources for all Widgets: XmNx, XmNy: A Widget's position on the display.

XmNheight, XmNwidth: A Widget's size.

XmNborderWidth: Set to 1 by default.

XmNsensitive: A Boolean resource that specifies whether this Widget can receive input.

XmNcolorMap: The default color map.

XmNbackground: The background color.

Check the Motif Programmer's reference manual for a complete listing.

XmPrimitive

The XmPrimitive Widget class inherits all the resources from Core and adds more functionality.

XmNforeground: The foreground color.

XmNhighlightOnEnter: Changes color when the pointer is within a displayed area of Widget.

XmNhighlightThickness: If **XmNhighlightOnEnter** is **TRUE**, changes the border to this thickness.

XmNhighlightColor: The color a **Widget** changes to when highlighted.

XmNshadowThickness: This is the thickness to show the pseudo-three-dimensional look for which **Motif** is famous. The default value is 2.

XmNtopShadowColor and **XmNbottomShadowColor:** Sets the color for top and bottom lines around a **Widget**.

XmNuserData: A pointer available for the programmer's use.

The **XmPrimitive** **Widget** also provides the **XmNdestroyCallback** resource. This can be set to a function that would do clean-up when a **Widget** is destroyed. In **Motif 1.2.x** or later, the **XmPrimitive** class also provides a **XmNhelpCallback** that is called when the **F1** key is pressed in the **Widget**'s window. This is to allow specific help information for a button, for example.

XmManager

The **XmManager** class provides support for all **Motif** **Widgets** that contain other **Widgets**. This is never used directly in an application, and it works in a manner similar to the **XmPrimitive** class. It provides the following resources:

XmNforeground: The color of the pixels in the foreground.

XmNshadowThickness: For the three-dimensional effect.

XmNtopShadowColor: For the three-dimensional effect. This is automatically defaulted to a color derived from the background color. This color is used on the left and top borders of a **Widget**.

XmNbottomShadowColor: For the three-dimensional effect. This is automatically defaulted to a color derived from the background color. This color is used on the right and bottom borders of a **Widget**.

XmNuserData: For storing user data. Could be used as a **void** pointer.

The Label Widget

The **Label** **Widget** is used to display strings or pixmaps. Include the **Xm/Label.h** file in your source file before using this **Widget**. Some of the resources for this **Widget** include these: **XmNalignment:** Determines the alignment of the text in this **Widget**. The acceptable values are **XmALIGNMENT_END**, **XmALIGNMENT_CENTER**, and **XmALIGNMENT_BEGIN** for right, center, and left justification, respectively.

XmNrecomputeSize: A **Boolean** resource. If set to **TRUE**, the **Widget** will resize should the size of the string or pixmap change dynamically. This is the default. If set to **FALSE**, the **Widget** will not attempt to resize itself.

XmNlabelType: The default value of this type is **XmSTRING** to show strings. However, it can also be set to **XmPIXMAP** when displaying a pixmap specified in the **XmNpixmap** resource.

XmNlabelPixmap: This is used to specify which pixmap to use when the **XmNlabelType** is set to **XmPIXMAP**.

XmNlabelString: This is used to specify which **XmString** compound string to use for the label. This defaults to the name of the label. See the section, "Strings in **Motif**: Compound Strings."

To get acquainted with left and right justification on a label, see Listing 34.2. The listing also shows how the

resources can be set to change Widget parameters programmatically and via the .Xresource files.

Listing 34.2. How to use a Label Widget.

```
/*

** This application shows how to use a Label Widget.

*/

#include <X11/Intrinsic.h>

#include <Xm/Xm.h>

#include <Xm/Form.h>

#include <Xm/PushB.h>

#include <Xm/Label.h>    /** <---- for the label **/


void  bye(Widget w, XtPointer clientdata, XtPointer calldata);


int main(int  argc, char **argv)
```

```
{

Widget top;

XtAppContext app;

Widget aForm;

Widget aLabel;          /** <---- for the label ***/

Widget aButton;

Arg    args[5];

/**

*** Initialize the toolkit.

**/

top = XtAppInitialize(&app, "KBH", NULL, 0, (Cardinal *)&argc,

                     argv, NULL, args, 0);

/**
```

```
*** Create a Form on this top-level Widget. This is a nice Widget

*** to place other Widgets on top of.

**/

aForm = XtVaCreateManagedWidget("Form1",

    xmFormWidgetClass, top,

    XmNheight,90,

    XmNwidth,200,

    NULL);

/**

*** Add a button on the form you just created. Note how this Button

*** Widget is connected to the form that it resides on. Only

*** left, right, and bottom edges are attached to the form. The

*** top edge of the button is not connected to the form.

**/
```

```
aButton = XtVaCreateManagedWidget("Push to Exit",

    xmPushButtonWidgetClass, aForm,

    XmNheight, 20,

    XmNleftAttachment, XmATTACH_FORM,

    XmNrightAttachment, XmATTACH_FORM,

    XmNbottomAttachment, XmATTACH_FORM,

    NULL);

/**

*** Now let's create the label for us.

*** The alignment is set to right-justify the label.

*** Note how the label is attached to the parent form.

**/

aLabel = XtVaCreateManagedWidget("This is a right justified Label",
```

```
    xmLabelWidgetClass, aForm,

    XmNalignment, XmALIGNMENT_END,

    XmNleftAttachment, XmATTACH_FORM,

    XmNrightAttachment, XmATTACH_FORM,

    XmNtopAttachment, XmATTACH_FORM,

    NULL);

/**

*** Call the function "bye" when the PushButton receives

*** an activate message; i.e. when the pointer is moved to

*** the button and Button1 is pressed and released.

**/

XtAddCallback( aButton, XmNactivateCallback, bye, (XtPointer) NULL);

XtRealizeWidget(top);

XtAppMainLoop(app);
```

```
return(0);
```

```
}
```

```
void bye(Widget w, XtPointer clientdata, XtPointer calldata)
```

```
{
```

```
exit(0);
```

```
}
```

Output is shown in Figure 34.4.



Figure 34.4. *Using the Label Widget.*

Avoid using the `\n` in the label name. If you have to create a multistring Widget, use the `XmStringCreate` calls to create a compound string (see the next section). Another way to set the string is by specifying it in the resource file and then merging the resources.

The listing shows the label to be right-justified. You could easily center the string horizontally by not specifying the alignment at all and letting it default to the center value. Alternatively, try setting the alignment parameter to `XmALIGNMENT_BEGINNING` for a left-justified label.

Strings in Motif: Compound Strings

A compound string is Motif's way of representing a string. In a typical C program, a NULL-terminated string is enough to specify a string. In Motif, a string is also defined by the character set it uses. Strings in Motif are referred to as compound strings and are kept in opaque data structures called `XmString`.

In order to get a compound string from a regular C string, use this function call:

```
XmString XmStringCreate( char *text, char *tag);
```

This function will return an equivalent compound string given a pointer to a NULL-terminated C string and a tag. The tag specifies which fontlist to use and is defaulted to `XmFONTLIST_DEFAULT_TAG`.

New lines in C strings have to be handled by special separators in Motif. So to create a string and preserve the newlines, use the call

```
XmString XmStringCreateLtoR( char *text, char *tag);
```

The compound strings have to be created and destroyed just like any other object. They persist long after the function call that created them returns. Therefore, it's a good idea to free all locally used `XmStrings` in a function before returning, or else all references to the strings will be lost. The definition of a call to free `XmString` resources is

```
XmStringFree( XmString s);
```

You can run similar operations on strings as you would in a C program, except that these string operations are called by different names. Use the function

```
Boolean XmStringByteCompare( XmString s1, XmString s2);
```

for a strict byte-for-byte comparison, and for just the text comparison, use

```
Boolean XmStringCompare( XmString s1, XmString s2);
```

To check if a string is empty, use

```
Boolean XmStringEmpty( XmString s1);
```

To string two strings together, use

```
XmString XmStringConcat( XmString s1, XmString s2);
```

`XmString Concat()` creates a new string by concatenating `s2` to `s1` and returns it. This returned string has to be freed just like `s1` and `s2`.

If you want to use `sprintf`, use it on a temporary buffer and then create a new string. For example:

```
char str[32];
```

```
XmString xms;
```

```
.....
```

```
sprintf(str, " pi = %lf, Area = %lf", PI, TWOPI*r);
```

```
xms = XmStringCreateLtoR( str, XmFONTLIST_DEFAULT_TAG);
```

```
.....
```

```
n = 0;
```

```
XtSetArg(arg[n],XmNlabelString,xms); n++;
```

```
XtSetValues(someLabel, arg, n);
```

```
XmStringFree(xms);
```

If a string value becomes corrupted without your performing any direct actions on it, check to see whether the Widget is not making a copy for its use of the passed `XmString`. Sometimes a Widget might be keeping only a pointer to the `XmString`. If that string were freed, the Widget might wind up pointing to bad data.

One good way to check is to set an `XmString` resource, then use the `XtGetValues` function to get the same resource from the Widget. If the values of the `XmStrings` are the same, the Widget is not making a copy for itself. If they are not the same, it is safe to free the original because the Widget is making a local copy. The default course of action is to assume that a Widget makes a copy of such resources for itself.

A similar test could be used to determine whether a Widget returns a copy of its resource or a pointer to it. Use the same listing shown two paragraphs ago, but this time use a `getValue` to get the same resource twice. Then do the comparison to see whether the address for the original string matches the address of the returned value from `getValue()`: If the values match, the Widget keeps a pointer. If the values do not match, the Widget keeps an internal copy.

```
/**

*** This is a sample partial listing of how to check if the

*** data returned on an XtGetValues and an XtSetValues

*** call is a copy or a reference.

***/

#include "Xm/Text.h"

..

Widget w;
```

```
XmString x1, x2, x3;

x3 = XmStringCreateLtoR("test", XmFONTLIST_DEFAULT_TAG);

XmTextSetString(w,x3);

...

x1 = XmTextGetString(w);

x2 = XmTextGetString(w);


XtWarning(" Checking SetValues");

if (x1 != x3)

    XtWarning("Widget keeps a copy ! Free original!");

else

    XtWarning("Widget does not keep a copy! Do NOT free original");
```

```

XtWarning(" Checking GetValues");

if (x1 == x2)

    XtWarning("Widget returns a copy! Do NOT free");

else

    XtWarning("Widget does not return a copy! You should free it ");

```

The `XtWarning()` message is especially useful for debugging the progress of programs. The message is relayed to the `stderr` of the invoking application. If this is an `xterm`, you will see an error message on that terminal window. If no `stderr` is available for the invoke mechanism, the message is lost.

TIP: The `XtSetArg` macro is defined as

```

#define XtSetArg(arg,n,d) \
((void)((arg).name = (n).(arg).value = (XtArgVal)(d)))

```

Do not use `XtSetArg (arg [n++] ...` because this will increment `n` twice.

The XmPushButton Widget Class

The `XmPushButton` is perhaps the most heavily used Widget in Motif. Listings 34.1 and 34.2 show the basic usage for `PushButton`. When a button is pressed in the `PushButton` area, the button goes into an armed state (a state between not pressed and going to pressed). The color of the button changes to reflect this state, and you can set this color by using `XmNarmColor`. This color is shown when the `XmNfillOnArm` resource is set to `TRUE`.

TIP: If the `armColor` for a `PushButton` does not seem to be working, try setting the `XmNfillOnArm` resource to `TRUE`.

The callback functions for a `PushButton` are the following: `XmNarmCallback`: Called when a `PushButton` is armed.

`XmNactivateCallback`: Called when a button is released in the Widget area while the Widget is armed. This is not invoked if the pointer is outside the Widget when the button is released.

`XmNdisarmCallback`: Called when a button is released with the pointer outside the Widget area while the Widget is armed.

TIP: If a callback has more than one function registered for a Widget, all of the functions will be called, but not necessarily in the order they were registered. Also, do not rely on the same order being preserved on other systems. If you want more than one function performed during a callback, sandwich them in one function call.

In Listing 34.2, you saw how a callback function was added to a `PushButton` with the `XtAddCallback` function. The same method can be used to call other functions for other actions such as the `XmNdisarmCallback`.

The `XmToggleButton` Widget Class

This class is a subclass of the `XmLabel` Widget class. You can have two types of buttons: one of many or n of many. When using one of many, the user can make only one selection from many items. (See Figure 34.5.) When using n of many, the user can select many options. (See Figure 34.6.) Note the way the buttons are drawn, with one of many shown as diamonds and n of many shown as boxes.



Figure 34.5. *Using one of many toggle buttons.*



Figure 34.6. *Using n of many toggle buttons.*

The resources for this Widget include the following: `XmNindicatorType`: Determines the style. Can be set to `XmN_OF_MANY` or `XmONE_OF_MANY` (the default).

`XmNspacing`: The number of pixels between the button and its label.

`XmNfillOnSelect`: The color of the button changes to reflect a set when the `XmNfillOnArm` resource is set to `TRUE`.

`XmNfillColor`: The color to show when set.

`XmNset`: A Boolean resource indicating whether the button is set or not. If this resource is set from a program, the button will automatically reflect the change.

It's easier to use the convenience functions `XmToggleButtonGetState(Widget w)` to get the Boolean state for a Widget, and `XmToggleButtonSetState(Widget w, Boolean b)` to set the value for a `ToggleButton` Widget.

Similar to the `PushButton` class, the `ToggleButton` class has three callbacks: `XmNarmCallback`: Called when the `ToggleButton` is armed.

`XmNvalueChangedCallback`: Called when a button is released in the Widgets area while the Widget is armed. This is not invoked if the pointer is outside the Widget when the button is released.

`XmNdisarmCallback`: Called when a button is released with the pointer outside the Widget area while the Widget was armed.

For the callbacks, the data passed into the Callback function is a structure of type:

```
typedef struct {

    int    reason;

    Xevent    *event;

    int    set;

} XmToggleButtonCallbackStruct;
```

The reason for the callback is one of the following events: XmCR_ARM, XmCR_DISARM, or XmCR_ACTIVATE. The event is a pointer to XEvent , which caused this callback. The set value is 0 if the item is not set, or nonzero if set. Look at Listing 34.3, which shows the use of ToggleButton. The ToggleButtons are arranged in one column via the RowColumn Widget, discussed later in this chapter.

Listing 34.3. Using ToggleButton in Motif.

```
/*

** This is a typical Motif application that demonstrates the use of

** a ToggleButton Widget(s) stored on a RowColumn Widget.

*/

#include <X11/Intrinsic.h>

#include <Xm/Xm.h>

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



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

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

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

```
void  bye(Widget w, XtPointer clientdata, XtPointer calldata);
```

```
#define MAX_BTNS 4
```

```
int main(int  argc, char **argv)
```

```
{
```

```
Widget top;
```

```
XtAppContext app;
```

```
Widget aForm;
```

```
Widget aButton;
```

```
Widget aRowCol;
```

```
char str[32];
```

```
Widget aToggle[MAX_BTNS];
```

```
Arg args[5];
```

```
int i;
```

```
/**
```

```
*** Initialize the toolkit.
```

```
**/
```

```
top = XtAppInitialize(&app, "KBH", NULL, 0, (Cardinal *)&argc,
```

```
argv, NULL, args, 0);
```

```
/**
```

```
*** Create a Form on this top-level Widget. This is a nice Widget
```

```
*** to place other Widgets on top of.
```

```
** /
```

```
aForm = XtVaCreateManagedWidget("Form1",
```

```
    xmFormWidgetClass, top,
```

```
    XmNheight,150,
```

```
    XmNwidth,100,
```

```
    NULL);
```

```
/**
```

```
*** Add a button on the form you just created. Note how this Button
```

```
*** Widget is connected to the form that it resides on. Only
```

```
*** left, right, and bottom edges are attached to the form. The
```

```
*** top edge of the button is not connected to the form.
```

```
** /
```

```
aButton = XtVaCreateManagedWidget("Push to Exit",
```

```

xmPushButtonWidgetClass, aForm,

XmNheight, 20,

XmNleftAttachment, XmATTACH_FORM,

XmNrightAttachment, XmATTACH_FORM,

XmNbottomAttachment, XmATTACH_FORM,

NULL);

```

```

#define DO_RADIO

```

```

/**

```

```

*** A quick intro to the hierarchy in Motif.

```

```

*** Let's create a RowColumn Widget to place all ToggleButtons.

```

```

*** Note how the RowColumn button attaches itself to

```

```

*** the top of the button.

```

```

**/

```

```

aRowCol = XtVaCreateManagedWidget("rowcol",

```

```
        xmRowColumnWidgetClass, aForm,

#ifdef DO_RADIO

        XmNradioBehavior, TRUE,

        XmNradioAlwaysOne, TRUE,

#endif

        XmNleftAttachment, XmATTACH_FORM,

        XmNrightAttachment, XmATTACH_FORM,

        XmNtopAttachment, XmATTACH_FORM,

        XmNbottomAttachment, XmATTACH_WIDGET,

        NULL) ;

/**

*** Make ToggleButtons on this form called RowCol. Attach them all to the

*** RowColumn Widget on top of the form.

***

*** Note the radioBehavior setting
```

```
    **/

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

    {

        sprintf(str,"Button %d",i);

        aToggle[i] = XtVaCreateManagedWidget(str,

            xmToggleButtonWidgetClass, aRowCol,

            XmNradioBehavior, TRUE,

            NULL);

    }

XmToggleButtonSetState(aToggle[0],TRUE, FALSE);

/**

*** Call the function "bye" when the PushButton receives

*** an activate message; i.e. when the pointer is moved to
```

```

*** the button and Button1 is pressed and released.

**/

XtAddCallback( aButton, XmNactivateCallback, bye, (XtPointer) NULL);

XtRealizeWidget(top);

XtAppMainLoop(app);

return(0);

}

void  bye(Widget w, XtPointer clientdata, XtPointer calldata)

{

exit(0);

}

```

Refer to Figure 34.6, shown previously, for the output of this listing with the `#define DO_RADIO` line commented out. By defining the `DO_RADIO` label, you can make this into a Radio Button application. That is, only one of the buttons can be selected at one time. Refer to Figure 34.5, shown previously, for the radio behavior of these buttons.

Convenience Functions

Usually the time to set resources for a Widget is at the Widget's creation. This is done either with the `XtVaCreateManagedWidget` call or with the `XmCreateYYY` call, where YYY is the name of the Widget you are creating.

This test uses the variable argument call to create and manage Widgets. I do it simply because it's easier for me to see what resources I am setting and to create them all in one function call. You might have a personal preference to do it in two steps. Either way is fine. Keep in mind, though, that if you do use the `XmCreateYYY` call, you have to set the resource settings in a list of resource sets. Listing 34.4 is an example of creating a Label Widget. This is a function that creates a Label Widget on a Widget given the string x.

Listing 34.4. Sample convenience function for creating a label on a form.

```
/**

*** This is a sample convenience function to create a label

*** Widget on a parent. The string must be a compound string.

**/

Widget makeLabel( Widget onThis, XmString x)

{

Widget    lbl;

Cardinal n;

Arg    arg[5];

n = 0;

XtSetArg(arg[n], XmNalignment, XmALIGNMENT_BEGIN); n++;
```



```

XtSetArg(arg[n], XmNlabelString, x); n++;

lbl = XmCreateLabel("A Label", onThis, arg, n);

XtManageChild(lbl);

return(lbl);

}

```

Or you could use the variable argument lists in creating this label, as shown in Listing 34.5.

Listing 34.5. Creating a label.

```

/**

*** Another way of making a label on a parent.

**/

Widget makeLabel( Widget onThis, XmString x)

{

Widget    lbl;

lbl = XmCreateManagedWidget("A Label",

                               xmLabelWidgetClass, onThis,

```

```

        XmNalignment, XmALIGNMENT_BEGIN,

        XmNlabelString, x,

        NULL);

return(lbl);

}

```

In either case, it's your judgment call as to which one to use. The variable list method of creating is a bit easier to read and maintain. But what about setting values after a Widget has been created? This would be done via a call to `XtSetValue` with a list and count of resource settings. For example, to change the alignment and text of a label, you would use `XtSetValues`:

```

n = 0;

XtSetArg(arg[n], XmNalignment, XmALIGNMENT_BEGIN); n++;

XtSetArg(arg[n], XmNlabelString, x); n++;

XtSetValues(lbl,arg,n);

```

Similarly, to get the values for a Widget, you would use `XtGetValues`:

```

Cardinal n; /* usually an integer or short... use Cardinal to be safe */

int align;

```

```

XmString x;

...

n = 0;

XtSetArg(arg[n], XmNalignment, &align); n++;

XtSetArg(arg[n], XmNlabelString, &x); n++;

XtGetValues(lbl, arg, n);

```

In the case of other Widgets, let's use the Text Widget. This setting scheme is hard to read, quite clumsy, and prone to typos. For example, when you get a string for a Text Widget, should you use `x` or address of `x`?

For this reason, Motif provides convenience functions. For example, in the `ToggleButton` Widget class, rather than using the combination of `XtSetValue` and `XtSetArg` calls to get the state, you would use one call, `XmToggleButtonGetState(Widget w)`, to get the state. These functions are valuable code savers when writing complex applications. In fact, you should write similar convenience functions whenever you can't find one that suits your needs.

The List Widget

This displays a list of items from which the user selects. The list is created from a list of compound strings. Users can select either one or many items from this list. The resources for this Widget include the following: `XmNItemCount`: Determines the number of items in the list.

`XmNItems`: An array of compound strings. Each entry corresponds to an item in the list. Note that a List Widget makes a copy for all items in its list when using `XtSetValues`; however, it returns a pointer to its internal structure when returning values to a `XtGetValues` call. Therefore, do not free this pointer from `XtGetValues`.

`XmNselectedItemCount`: The number of items currently selected.

`XmNselectedItems`: The list of selected items.

`XmNvisibleItemCount`: The number of items to display at one time.

`XmNselectionPolicy`: Used to set single or multiple selection capability. If set to `XmSINGLE_SELECT`, the user will be able to only select one item. Each selection will invoke the `XmNsingleSelectionCallback`. Selecting one item will deselect a previously selected item. If set to `XmEXTENDED_SELECT`, the user will be able to

select a block of contiguous items in a list. Selecting a new item or more will deselect another previously selected item and will invoke the `XmNmultipleSelection` callback. If set to `XmMULTIPLE_SELECT`, the user will be able to select multiple items in any order. Selecting one item will not deselect another previously selected item but will invoke the `XmNmultipleSelection` callback. If set to `XmBROWSE_SELECT`, the user can move the pointer (with the button pressed) across all the selections, but only one item will be selected. Unlike the `XmSINGLE_SELECT` setting, the user does not have to press and release the button on an item to select it. The `XmbrowseSelectionCallback` will be invoked when the button is finally released on the last item browsed.

It is easier to create the List Widget with a call to `XmCreateScrolledList()` because this will automatically create a scrolled window for you. However, this method may prove to be slow when compared to `XtSetValues()` calls. If you feel that speed is important, consider using `XtSetValues()`. You should create the list for the first time by using `XtSetValues`.

The following convenience functions will make working with List Widgets easier: `XmListAddItem(Widget w, XmString x, int pos)`

This will add the compound string `x` to the List Widget `w` at the one relative position `pos`. If `pos` is 0, the item is added to the back of the list. This function is very slow, so do not use it to create a newlist, because it rearranges the entire list before returning.

```
XmListAddItems(Widget w, XmString *x, int count, int pos);
```

This will add the array of compound strings `x` of size `count` to the List Widget `w` from the position `pos`. If `pos` is 0, the item is added to the back of the list. This function is slow, too, so do not use it to create a newlist.

```
XmDeleteAllItems(Widget w)
```

This will delete all the items in a list. It's better to write a convenience function to do

```
n = 0; XtSetArg(arg[n], XmNitems, NULL); n++; XtSetArg(arg[n], XmNitemCount, 0); n++; XtSetValues(mylist, arg, n);
```

```
XmDeleteItem(Widget w, XmString x)
```

Deletes the item `x` from the list. This is a slow function.

```
XmDeleteItems(Widget w, XmString *x, int count)
```

Deletes all the count items in `x` from the list. This is an even slower function. You might be better off installing a new list.

```
XmListSelectItem(Widget w, XmString x, Boolean Notify)
```

Programmatically selects `x` in the list. If `Notify` is `TRUE`, the appropriate callback function is also invoked.

```
XmListDeselectItem(Widget w, XmString x)
```

Programmatically deselects `x` in the list.

```
XmListPos(Widget w, XmString x)
```

Returns the position of `x` in the list. Returns 0 if not found.

Let's use the List Widget for a sample application. See Listing 34.6.

Listing 34.6. Using List Widgets.

```
/*

** This is a typical Motif application for using a list Widget

*/

#include <X11/Intrinsic.h>

#include <Xm/Xm.h>

#include <Xm/Form.h>

#include <Xm/PushB.h>

#include <Xm/List.h>      /*** for the list Widget ***/

#include <Xm/ScrolledW.h> /*** for the scrolled window Widget ***/

/*** Some items for the list ***/
```

```
#define NUMITEMS 8

char *groceries[NUMITEMS] = {

    "milk",

    "eggs",

    "bread",

    "pasta",

    "cd-rom",    /** I don't go out often!*/

    "bananas",

    "yogurt",

    "oranges",

    };

/* For the list Widget, we need compound strings */

XmString xarray[NUMITEMS];

#define USE_SCROLL
```

```
void  bye(Widget w, XtPointer clientdata, XtPointer calldata);
```

```
int  main(int  argc, char **argv)
```

```
{
```

```
Widget top;
```

```
XtAppContext app;
```

```
Widget aForm;
```

```
Widget aList;
```

```
Widget aButton;
```

```
Arg    args[15];
```

```
int    i;
```

```
/**
```

```
*** Initialize the toolkit.
```

```
    **/

    top = XtAppInitialize(&app, "KBH", NULL, 0, (Cardinal *)&argc,

        argv, NULL, args, 0);

/**

*** Create a Form on this top-level Widget. This is a nice Widget

*** to place other Widgets on top of.

**/

aForm = XtVaCreateManagedWidget("Form1",

    xmFormWidgetClass, top,

    XmNheight, 90,

    XmNwidth, 200,

    NULL);

/**
```


*** Add a button on the form you just created. Note how this Button

*** Widget is connected to the form that it resides on. Only

*** left, right, and bottom edges are attached to the form. The

*** top edge of the button is not connected to the form.

**/

aButton = XtVaCreateManagedWidget("Push to Exit",

 xmPushButtonWidgetClass, aForm,

 XmNheight, 20,

 XmNleftAttachment, XmATTACH_FORM,

 XmNrightAttachment, XmATTACH_FORM,

 XmNbottomAttachment, XmATTACH_FORM,

 NULL);

/**

*** Now create a list of items for this Widget.

**/

```
for (i=0; i < NUMITEMS; i++)

    xarray[i] = XmStringCreateLtoR(groceries[i],

                                   XmSTRING_DEFAULT_CHARSET);

#endif

/**

*** Then create the list Widget itself. Note this will not

*** put up a scroll bar for you.

**/

aList = XtVaCreateManagedWidget("Push to Exit",

    xmListWidgetClass, aForm,

    XmNitemCount, NUMITEMS,

    XmNitems, xarray,

    XmNvisibleItemCount, 4,

    XmNscrollBarDisplayPolicy, XmAS_NEEDED,
```

```
    XmNleftAttachment,XmATTACH_FORM,

    XmNrightAttachment,XmATTACH_FORM,

    XmNtopAttachment,XmATTACH_FORM,

    XmNbottomAttachment,XmATTACH_WIDGET,

    XmNbottomWidget,aButton,

    NULL);

#else

/**

*** Alternatively, use the scrolled window with the following code:

**/

i = 0;

XtSetArg(args[i], XmNitemCount, NUMITEMS); i++;

XtSetArg(args[i], XmNitems, xarray); i++;

XtSetArg(args[i], XmNvisibleItemCount, 4); i++;
```

```

XtSetArg(args[i], XmNscrollBarDisplayPolicy, XmAS_NEEDED); i++;

XtSetArg(args[i], XmNleftAttachment,XmATTACH_FORM); i++;

XtSetArg(args[i], XmNrightAttachment,XmATTACH_FORM); i++;

XtSetArg(args[i], XmNtopAttachment,XmATTACH_FORM); i++;

XtSetArg(args[i], XmNbottomAttachment,XmATTACH_WIDGET); i++;

XtSetArg(args[i], XmNbottomWidget,aButton); i++;

aList = XmCreateScrolledList(aForm,"groceryList",args,i);

XtManageChild(aList);

#endif

/**

*** Call the function "bye" when the PushButton receives

*** an activate message; i.e. when the pointer is moved to

*** the button and Button1 is pressed and released.

**/

```

```

XtAddCallback( aButton, XmNactivateCallback, bye, (XtPointer) NULL);

XtRealizeWidget(top);

XtAppMainLoop(app);

return(0);

}

void bye(Widget w, XtPointer clientdata, XtPointer calldata)

{

exit(0);

}

```

XmScrollBar

The ScrollBar Widget enables the user to select a value from a range. Its resources include the following:

XmNvalue: The value representing the location of the slider.

XmNminimum and **XmNmaximum:** The range of values for the slider.

XmNshowArrows: Boolean value if set shows arrows at either end.

XmNOrientation: Set to **XmHORIZONTAL** for a horizontal bar or **XmVERTICAL** (default) for a vertical bar.

XmNprocessingDirection: Set to either **XmMAX_ON_LEFT** or **XmMAX_ON_RIGHT** for **XmHORIZONTAL** orientation, or **XmMAX_ON_TOP** or **XmMAX_ON_BOTTOM** for **XmVERTICAL** orientation.

`XmNincrement`: The increment per move.

`XmNpageIncrement`: The increment if a button is pressed in the arrows or the box. This is defaulted to 10.

`XmNdecimalPoint`: Shows the decimal point from the right.

Note that all values in the ScrollBar Widget's values are given as integers. Look at the radio station selection example in Listing 34.7. A point to note in this listing is that the exit button for the application is offset on the left and right by 20 pixels. This is done via the `XmATTACH_FORM` value for each side (left or right) being offset by the value in `XmNleftOffset` and `XmNrightOffset`, respectively. See Listing 34.3 on how to attach items to a form.

Listing 34.7. Using the Scale Widget.

```
/*

** An application to show the radio station selection via a scale.

*/

#include <X11/Intrinsic.h>

#include <Xm/Xm.h>

#include <Xm/Form.h>

#include <Xm/PushB.h>

#include <Xm/Scale.h>

#define MAX_SCALE 1080
```

```
#define MIN_SCALE 800

void  bye(Widget w, XtPointer clientdata, XtPointer calldata);

void myfunction(Widget w, XtPointer dclient,  XmScaleCallbackStruct *p);


int main(int  argc, char **argv)

{

Widget top;

XtAppContext app;

Widget aForm;

Widget aScale;

Widget aButton;

XmString xstr; /* for the scale title */

Arg  args[5];

/**
```

```
*** Initialize the toolkit.

**/

top = XtAppInitialize(&app, "ScaleMe", NULL, 0, (Cardinal *)&argc,

                    argv, NULL, args, 0);

/**

*** Create a Form on this top-level Widget. This is a nice Widget

*** to place other Widgets on top of.

**/

aForm = XtVaCreateManagedWidget("Form1",

                                xmFormWidgetClass, top,

                                XmNheight, 90,

                                XmNwidth, 240,

                                NULL);
```



```
/**

*** Add a button on the form you just created. Note how this Button

*** Widget is connected to the form that it resides on. Only

*** left, right, and bottom edges are attached to the form. The

*** top edge of the button is not connected to the form.

**/

aButton = XtVaCreateManagedWidget("Push to Exit",

    xmPushButtonWidgetClass, aForm,

    XmNheight, 20,

    XmNleftAttachment, XmATTACH_FORM,

    XmNleftOffset, 20,

    XmNrightAttachment, XmATTACH_FORM,

    XmNrightOffset, 20,

    XmNbottomAttachment, XmATTACH_FORM,

    NULL);
```

```
/**

***  Create the radio FM selection scale here.

***  Note that since we have to work with integers,

***  we have to set the frequency scale to 10x actual

***  value and then set the decimal point explicitly

***  for the display. No provision is made for selecting

***  odd frequencies.

**/

xstr = XmStringCreateLtoR("Radio Stations", XmSTRING_DEFAULT_CHARSET);

aScale = XtVaCreateManagedWidget("sample it",

    xmScaleWidgetClass, aForm,

    XmNheight, 40,

    XmNminimum, 800,

    XmNvalue, 1011,
```

```
XmNmaximum, 1080,  
  
XmNdecimalPoints, 1,  
  
XmNtitleString, xstr,  
  
XmNshowValue, TRUE,  
  
XmNorientation, XmHORIZONTAL,  
  
XmNprocessingDirection, XmMAX_ON_RIGHT,  
  
XmNleftAttachment, XmATTACH_FORM,  
  
XmNrightAttachment, XmATTACH_FORM,  
  
XmNtopAttachment, XmATTACH_FORM,  
  
XmNbottomAttachment, XmATTACH_WIDGET,  
  
XmNbottomAttachment, aButton,  
  
NULL);  
  
XmStringFree(xstr);
```

```
/**

*** Call the function "bye" when the PushButton receives

*** an activate message; i.e. when the pointer is moved to

*** the button and Button1 is pressed and released.

**/

XtAddCallback( aButton, XmNactivateCallback, bye, (XtPointer) NULL);

XtAddCallback( aScale, XmNvalueChangedCallback, myfunction, (XtPointer) NULL);

XtRealizeWidget(top);

XtAppMainLoop(app);

return(0);

}

void bye(Widget w, XtPointer clientdata, XtPointer calldata)

{

exit(0);
```

```
}

void myfunction(Widget w, XtPointer dclient, XmScaleCallbackStruct *p)

{

int k;

k = p->value;

if ((k & 0x1) == 0) /** % 2 is zero ** check limits and increase **/

{

k++;

if (k >= MAX_SCALE) k = MIN_SCALE + 1;

if (k <= MIN_SCALE) k = MAX_SCALE - 1;

XmScaleSetValue(w,k); /** this will redisplay it too **/

}

}
```

In the case of FM selections, you would want the bar to show odd numbers. A good exercise for you would be to allow only odd numbers in the selection. Hint: Use `XmNvalueChangedCallback` as follows:

```
XtAddCallback(aScale, XmNvalueChangedCallback, myfunction);
```

The output is shown in Figure 34.7.

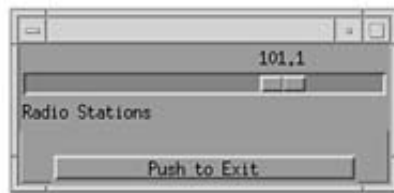


Figure 34.7. Using the Scale Widget.

The callback sends a pointer to the structure of type `XmScaleCallbackStruct` to a function called `myfunction`. A sample function, `myfunction()`, follows:

```
/**

*** Partial listing for not allowing even numbers for FM selection.

**/

#define MAX_SCALE 1080
```

```
#define MIN_SCALE 800

static void myfunction(Widget w, XtPointer dclient, XmScaleCallbackStruct *p)

{

int k;

k = p->value;

if ((k & 0x1) == 0)  /** % 2 is zero ** check limits and increase **/

{

k++;

if (k >= MAX_SCALE) k = MIN_SCALE + 1;

if (k <= MIN_SCALE) k = MAX_SCALE - 1;

XmScaleSetValue(w,k);  /** this will redisplay it too **/

}

}
```

Text Widgets

This Widget enables the user to type in text and provides full text-editing capabilities. This text could be multiline or single-line. If you are sure you want only single-line input from the user, you can specify the `TextField` Widget. This is simply a scaled-down version of the `Text` Widget. The resources for both are the same unless explicitly stated. They include the following: `XmNvalue`: A character string, just like in C. This is different from Motif 1.1 or older, in which this value used to be a compound string. If you have Motif 1.2 or later, this will be C string.

`XmNmarginHeight` and `XmNmarginWidth`: The number of pixels on either side of the Widget. The default is five pixels.

`XmNmaxLength`: Sets the limit on the number of characters in the `XmNvalue` resource.

`XmNcolumns`: The number of characters per line.

`XmNcursorPosition`: The number of characters at the cursor position from the beginning of the text file.

`XmNeditable`: Boolean value. If set to `TRUE`, enables the user to insert text.

The callbacks for this Widget are `XmNactivateCallback`: Called when the user presses the Enter key.

`XmNfocusCallback`: Called when the Widget receives focus from the pointer.

`XmNlosingFocusCallback`: Called when the Widget loses focus from the pointer.

This Widget has several convenience functions: `XmTextGetString(Widget w)` returns a C string (`char *`).

`XmTextSetString(Widget w, char *s)` sets a string for a Widget.

`XmTextSetEditable(Widget w, Boolean TRUEOrFALSE)` sets the Widget's editable string.

`XmTextInsert(Widget w, XmTextPosition pos, char *s)` sets the text at the position defined by `pos`. This `XmTextPosition` is an opaque item defining the index in the text array.

`XmTextShowPosition(Widget w, XmTextPosition p)` scrolls to show the rest of the string at position `p`.

`XmTextReplace(Widget w, XmTextPosition from, XmTextPosition to, char *s)` replaces the string starting from the location `from` inclusive to the position `to`, inclusive with the characters in string `s`.

`XmTextRemove(Widget w)` clears the text in a string.

`XmTextCopy(Widget w, Time t)` copies the currently selected text to the Motif clipboard. The `Time t` value is derived from the most recent `XEvent` (usually in a callback), which is used by the clipboard to take the most recent entry.

`XmTextCut(Widget w, Time t)` is similar to `XmTextCopy` but removes the selected text from the text's buffer.

`XmTextPaste(Widget w)` pastes the contents of the Motif clipboard onto the text area at the current cursor (insertion) position.

`XmTextClearSelection(Widget w, XmTextPosition p, XmTextPosition q, Time t)` selects the text from location `p` to location `q`.

In the following example, you could construct a sample editor application using the Text Widget. For the layout of the buttons, you would want to use Widgets of the `XmManager` class to manage the layout for you rather than having to do it in your own application. These Manager Widgets are `XmBulletinBoard`

`XmRowColumn`
`XmForm`

XmBulletinBoard Widgets

The `BulletinBoard` class enables the programmer to lay out Widgets on a `BulletinBoard` by specifying their `XmNx` and `XmNy` resources. These values are relative to the top-left corner of the `BulletinBoard` Widget. The `BulletinBoard` will not move its child Widgets around by itself. If a Widget resizes, it's the application's responsibility to resize and restructure its Widgets on the `BulletinBoard`.

The resources for the Widget are as follows: `XmNshadowType`: Specifies the type of shadow for this Widget. It can be set to `XmSHADOW_OUT` (the default), `XmSHADOW_ETCHED_IN`, `XmSHADOW_ETCHED_OUT`, or `XmSHADOW_IN`.

`XmNshadowThickness`: The number of pixels for the shadow. This is defaulted to 0 to not see a shadow.

`XmNallowOverlap`: Enables the children to be overlapped as they are laid on the Widget. This is a Boolean resource and is defaulted to `TRUE`.

`XmNresizePolicy`: Specifies the resize policy for managing itself. If set to `XmRESIZE_NONE`, it will not change its size. If set to `XmRESIZE_ANY`, the default, it will grow or shrink to attempt to accommodate all of its children automatically. If set to `XmRESIZE_GROW`, it will grow only, never shrink, automatically.

`XmNbuttonFontList`: Specifies the font for all `XmPushButton` children.

`XmNlabelFontList`: Specifies the default font for all Widgets derived from `XmLabel`.

`XmNtextFontList`: Specifies the default font for all `Text`, `TextField`, and `XmList` children.

It also provides the callback `XmNfocusCallback`, which is called when any children of the `BulletinBoard` receive focus.

XmRowColumn Widgets

The `XmRowColumn` Widget class orders its children in a row-and-column (or row major) fashion. This is used to set up menus, menu bars, and radio buttons. The resources provided by this Widget include the following:

`XmNorientation`: `XmHORIZONTAL` for a row major layout of its children; `XmVERTICAL` for a column major layout.

`XmNnumColumns`: Specifies the number of rows for a vertical Widget and the number of columns for a horizontal Widget.

XmNpacking: Determines how the children are packed. `XmPACK_TIGHT` enables the children to specify their own size. It fits children in a row (or column if `XmHORIZONTAL`) and then starts a new row if no space is available. `XmPACK_NONE` forces BulletinBoard-like behavior. `XmPACK_COLUMN` forces all children to be the size of the largest child Widget. This uses the `XmNnumColumns` resource and places all of its children in an organized manner.

XmNentryAlignment: Specifies which part of the children to use in its layout alignment. Its default is `XmALIGNMENT_CENTER`, but it can be set to `XmALIGNMENT_BEGINNING` for left or `XmALIGNMENT_END` for right side. This is on a per-column basis.

XmNverticalEntryAlignment: Specifies the alignment on a per-row basis. It can be assigned a value of `XmALIGNMENT_BASELINE_BOTTOM`, `XmALIGNMENT_BASELINE_TOP`, `XmALIGNMENT_CONTENTS_BOTTOM`, `XmALIGNMENT_CONTENTS_TOP`, or `XmALIGNMENT_CENTER`.

XmNentryBorder: The thickness of a border drawn around all children. Defaulted to 0.

XmNresizeWidth: A Boolean variable. If set to `TRUE`, will enable the RowColumn Widget to resize its width when necessary.

XmNresizeHeight: A Boolean variable. If set to `TRUE`, will enable the RowColumn Widget to resize its height when necessary.

XmNradioBehaviour: Works with `ToggleButton`s only. It enables only one `ToggleButton` of a group of buttons to be active at time. The default is `FALSE`.

XmNisHomogeneous: If set to `TRUE`, specifies that only children of the type of class in `XmNentryClass` can be children of this Widget. The default is `FALSE`.

XmNentryClass: Specifies the class of children allowed in this Widget if `XmNisHomogeneous` is `TRUE`.

A sample radio button application was shown in Listing 34.3. To see another example of the same listing but with two columns, see Listing 34.8.

Listing 34.8. Using RowColumn Widgets.

```
/*

** This is another Motif application that demonstrates the use of

** a ToggleButton Widget(s) stored on a multicolumn RowColumn Widget.

*/
```

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

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

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

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

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

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

```
void bye(Widget w, XtPointer clientdata, XtPointer calldata);
```

```
#define MAX_BTNS 8
```

```
#define NUM_COL 2
```

```
int main(int argc, char **argv)
```

```
{
```

```
Widget top;

XtAppContext app;

Widget aForm;

Widget aButton;

Widget aRowCol;

char  str[32];

Widget      aToggle[MAX_BTNS];

Arg         args[5];

int  i;


/**

*** Initialize the toolkit.

**/

top = XtAppInitialize(&app, "KBH", NULL, 0, (Cardinal *)&argc,

                     argv, NULL, args, 0);
```

```
/**

*** Create a Form on this top-level Widget. This is a nice Widget

*** to place other Widgets on top of.

**/

aForm = XtVaCreateManagedWidget("Form1",

    xmFormWidgetClass, top,

    XmNheight,150,

    XmNwidth,200,

    NULL);

/**

*** Add a button on the form you just created. Note how this Button

*** Widget is connected to the form that it resides on. Only
```

```
*** left, right, and bottom edges are attached to the form. The
```

```
*** top edge of the button is not connected to the form.
```

```
**/
```

```
aButton = XtVaCreateManagedWidget("Push to Exit",
```

```
    xmPushButtonWidgetClass, aForm,
```

```
    XmNheight, 20,
```

```
    XmNleftAttachment, XmATTACH_FORM,
```

```
    XmNrightAttachment, XmATTACH_FORM,
```

```
    XmNbottomAttachment, XmATTACH_FORM,
```

```
    NULL);
```

```
#define DO_RADIO
```

```
/**
```

```
*** A quick intro to the hierarchy in Motif.
```

```
*** Let's create a RowColumn Widget to place all Toggle
```

```
*** Buttons.

*** Note how the RowColumn button attaches itself to

*** the top of the button.

**/

aRowCol = XtVaCreateManagedWidget("rowcol",

    xmRowColumnWidgetClass, aForm,

#ifdef DO_RADIO

    XmNradioBehavior, TRUE,

    XmNradioAlwaysOne, TRUE,

#endif

    XmNnumColumns, NUM_COL,

    XmNleftAttachment, XmATTACH_FORM,

    XmNrightAttachment, XmATTACH_FORM,

    XmNtopAttachment, XmATTACH_FORM,

    XmNbottomAttachment, XmATTACH_WIDGET,
```

```
        NULL) ;

/**

*** Make ToggleButtons on this form called RowCol. Attach them all

*** RowColumn Widget on top of the form.

***

*** Note the radioBehavior setting

**/


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

{

    sprintf(str,"Button %d",i);

    aToggle[i] = XtVaCreateManagedWidget(str,

        xmToggleButtonWidgetClass, aRowCol,

        XmNradioBehavior, TRUE,

        NULL) ;
```



```
}
```

```
XmToggleButtonSetState(aToggle[0],True, False);
```

```
/**
```

```
*** Call the function "bye" when the PushButton receives
```

```
*** an activate message; i.e. when the pointer is moved to
```

```
*** the button and Button1 is pressed and released.
```

```
**/
```

```
XtAddCallback( aButton, XmNactivateCallback, bye, (XtPointer) NULL);
```

```
XtRealizeWidget(top);
```

```
XtAppMainLoop(app);
```

```
return(0);
```

```
}
```

```
void bye(Widget w, XtPointer clientdata, XtPointer calldata)
```

```
{  
  
exit(0);  
  
}
```

See Figure 34.8 for the output of Listing 34.8.

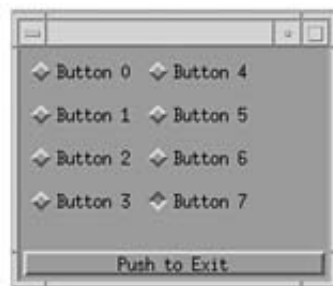


Figure 34.8. *Using the RowColumn Widget.*

XmForm Widgets

The beginning of this chapter introduced you to the workings of a Form Widget. This is the most flexible, but complex, Widget in Motif.

Its resources include these:

- XmNtopAttachment
- XmNleftAttachment
- XmNrightAttachment

- `XmNbottomAttachment`

These values specify how a child is assigned a position. The following values correspond to the each of the sides of the Widget: `XmATTACH_NONE`: Does not attach this side to form.

`XmATTACH_FORM`: Attaches to the corresponding side on form.

`XmATTACH_WIDGET`: Attaches this side to the opposite side of a reference Widget--for example, the right side of this Widget to the left side of the reference Widget. A reference Widget is another child on the same form.

`XmATTACH_OPPOSITE_WIDGET`: Attaches this side to the same side of a reference Widget. This is rarely used.

`XmATTACH_POSITION`: Attaches a side by the number of pixels shown in `XmNtopPosition`, `XmNleftPosition`, `XmNrightPosition`, and `XmNbottomPosition` resources, respectively.

`XmATTACH_SELF`: Uses `XmNx`, `XmNy`, `XmNheight`, and `XmNwidth`.

- `XmNtopWidget`
- `XmNleftWidget`
- `XmNrightWidget`
- `XmNbottomWidget`

These resources are set to the corresponding Widgets for each side for the `XmATTACH_WIDGET` setting in an attachment:

- `XmNtopOffset`
- `XmNleftOffset`
- `XmNrightOffset`
- `XmNbottomOffset`

These resources are the number of pixels a side of a child is offset from the corresponding Form side. The offset is used when the attachment is `XmATTACH_FORM`.

TIP: Sometimes it's hard to get the settings for a Form Widget just right, or the Form Widget doesn't lay out the Widgets in what seems to be the proper setting for a child Widget. In these cases, try to manage and lay out the children in ascending or descending order from the origin of the Form Widget. That is, create the top-left Widget first and use it as an anchor to create the next child, then the next one to its right, and so on. There is no guarantee that this will work, so try from the bottom right, bottom left, or top right for your anchor positions. If this technique doesn't work, try using two forms on top of the form with which you are working. Forms are cheap; your time is not. It's better to just make a form when two or more Widgets have to reside in a specific layout.

While trying a new layout on a Form Widget, if you get error messages about failing after 10,000 iterations, you have conflicting layout requests to a child or children Widgets. Check the attachments very carefully before proceeding. This error message results from the Form Widget trying different layout schemes to accommodate your request.

TIP: At times, conflicting requests to a form will cause your application to slow down while it's trying to accommodate your request, not show the form, or both. At this time, try to remove your Form settings one at a time until the errors disappear. You should then be able to weed out the offending resource setting for that form.

Designing Layouts

When designing layouts, think about the layout before you start writing code. Let's try an order entry example. See Listing 34.9.

Listing 34.9. Setting up a simple hierarchy.

```
/*

** This listing shows how to set up a hierarchy.

*/

#include <X11/Intrinsic.h>

#include <Xm/Xm.h>

#include <Xm/Form.h>

#include <Xm/PushB.h>

#include <Xm/Label.h>

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

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

/** Some items for the list */

#define NUMITEMS 4

char *thisLabel[NUMITEMS] = { "Name", "User Id", "Home", "Usage" };

/* For the list Widget, we need compound strings */

XmString xarray[NUMITEMS];

#define USE_SCROLL

void bye(Widget w, XtPointer clientdata, XtPointer calldata);

int main(int argc, char **argv)

{
```

```
Widget top;
```

```
XtAppContext app;
```

```
Widget masterForm;
```

```
Widget buttonForm;
```

```
Widget labelForm;
```

```
Widget inputForm;
```

```
Widget labelRC;
```

```
Widget textRC;
```

```
Widget buttonRC;
```

```
Widget inputText[NUMITEMS];
```

```
Widget inputLabel[NUMITEMS];
```

```
Widget searchBtn;
```

```
Widget cancelBtn;
```

```
Arg    args[15];
```

```
int    i;
```

```
/**

*** Initialize the toolkit.

**/

top = XtAppInitialize(&app, "KBH", NULL, 0, (Cardinal *)&argc,

                    argv, NULL, args, 0);

/**

*** Create a Form on this top-level Widget. This is a nice Widget

*** to place other Widgets on top of.

**/

masterForm = XtVaCreateManagedWidget("MasterForm",

    xmFormWidgetClass, top,

    XmNheight,150,

    XmNwidth,200,
```

```
NULL) ;
```

```
buttonForm = XtVaCreateManagedWidget( "ButtonForm",
```

```
    xmFormWidgetClass, masterForm,
```

```
    XmNtopAttachment, XmATTACH_POSITION,
```

```
    XmNtopPosition, 75,
```

```
    XmNheight, 30,
```

```
    XmNbottomAttachment, XmATTACH_FORM,
```

```
    XmNleftAttachment, XmATTACH_FORM,
```

```
    XmNrightAttachment, XmATTACH_FORM,
```

```
    NULL) ;
```

```
labelForm = XtVaCreateManagedWidget( "LabelForm",
```

```
    xmFormWidgetClass, masterForm,
```

```
    XmNtopAttachment, XmATTACH_FORM,
```



```
XmNbottomAttachment,XmATTACH_POSITION,
```

```
XmNbottomPosition, 75,
```

```
XmNleftAttachment,XmATTACH_FORM,
```

```
XmNrightAttachment,XmATTACH_POSITION,
```

```
XmNrightPosition,50,
```

```
NULL);
```

```
inputForm = XtVaCreateManagedWidget("InputForm",
```

```
xmFormWidgetClass, masterForm,
```

```
XmNtopAttachment,XmATTACH_FORM,
```

```
XmNbottomAttachment,XmATTACH_POSITION,
```

```
XmNbottomPosition, 75,
```

```
XmNrightAttachment,XmATTACH_FORM,
```

```
XmNleftAttachment,XmATTACH_POSITION,
```

```
XmNleftPosition,50,
```

```
        NULL) ;

/**

*** Now create the RowColumn manager Widgets

** /

buttonRC = XtVaCreateManagedWidget("buttonRC",

        xmRowColumnWidgetClass, buttonForm,

        XmNbottomAttachment,XmATTACH_FORM,

        XmNtopAttachment,XmATTACH_FORM,

        XmNleftAttachment,XmATTACH_FORM,

        XmNrightAttachment,XmATTACH_FORM,

        XmNOrientation, XmHORIZONTAL,

        NULL) ;
```

```
labelRC = XtVaCreateManagedWidget("buttonRC",  
  
    xmRowColumnWidgetClass, labelForm,  
  
    XmNbottomAttachment,XmATTACH_FORM,  
  
    XmNtopAttachment,XmATTACH_FORM,  
  
    XmNleftAttachment,XmATTACH_FORM,  
  
    XmNrightAttachment,XmATTACH_FORM,  
  
    XmNOrientation, XmVERTICAL,  
  
    NULL);
```

```
textRC = XtVaCreateManagedWidget("buttonRC",  
  
    xmRowColumnWidgetClass, inputForm,  
  
    XmNbottomAttachment,XmATTACH_FORM,  
  
    XmNtopAttachment,XmATTACH_FORM,  
  
    XmNleftAttachment,XmATTACH_FORM,  
  
    XmNrightAttachment,XmATTACH_FORM,
```

```
        XmNorIENTATION, XmVERTICAL,

        NULL);

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

{

    inputLabel[i] = XtVaCreateManagedWidget(thisLabel[i],

        xmLabelWidgetClass, labelRC,  NULL);

    inputText[i] = XtVaCreateManagedWidget(thisLabel[i],

        xmTextWidgetClass, textRC,  NULL);

}

searchBtn = XtVaCreateManagedWidget("Search",

    xmPushButtonWidgetClass, buttonRC,  NULL);

cancelBtn = XtVaCreateManagedWidget("Cancel",

    xmPushButtonWidgetClass, buttonRC,  NULL);
```

```
XtAddCallback( cancelBtn, XmNactivateCallback, bye, (XtPointer) NULL);

/** Add the handler to search here. */

XtRealizeWidget(top);

XtAppMainLoop(app);

return(0);

}

void bye(Widget w, XtPointer clientdata, XtPointer calldata)

{

exit(0);

}
```

The output of this application is shown in Figure 34.9. Notice how the labels do not line up with the Text Widget. There is a problem in the hierarchy of the setup. See the hierarchy of the application in Figure 34.10.

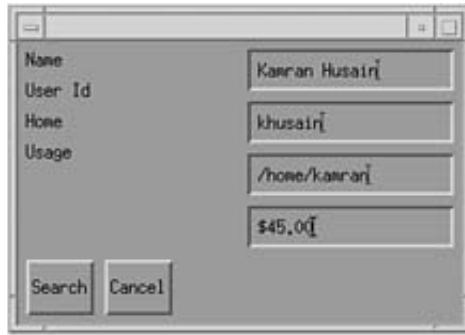


Figure 34.9. *The output of Listing 34.9.*

The Form Widgets are created to maintain the relative placements of all of the Widgets that correspond to a type of functionality. The RowColumn Widgets allow the placement of items on themselves. The best route to take in this example would be to lay one Text Widget and one label on one RowColumn Widget and have three RowColumn Widgets in all, one for each instance up to NUM_ITEMS. This will ensure that each label lines up with its corresponding Text Widget.

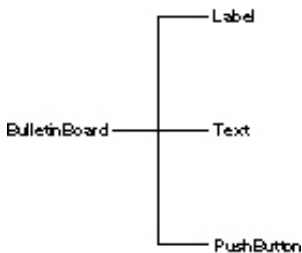


Figure 34.10. *The hierarchy of Listing 34.9.*

Here are a few points to note about laying out applications:

- Think about what the form or dialog is trying to achieve. Draw it on paper if you have to. Coding is the easy part; determining what to do is much harder.
- Be consistent. Users will love you for it. If Alt-X is a shortcut for Exit in one screen, do not make it a Cut operator in another. Keep controls on the same side of all dialogs and forms. Use separators to separate different functions on the same window.
- Choose a color scheme for your end users. What may be cool to you may be grotesque to the end user. They may not even be using a color monitor in some rare cases. A combination of white, gray, and black may be your best bet in this case if you do not want to deal with different color schemes in your code.

- Colors on your monitor may not be the same on the end user's monitor.
- Do not assume that the user has the same resolution monitor as yourself. Keep fonts clean, and buttons big enough for a large cursor. Allow windows to be resized as much as possible to enable users to customize their desktops.
- Assume nothing. If the user can size your window to an unworkable size, either limit the size in the resize callback to the lowest size or do not allow sizing at all.
- Offer some help for the user. In the future, Help will be required as a standard option on menu bars, so plan ahead.
- Avoid clutter. Too many options and entries on one huge form tend to confuse and baffle the user. Consider a two-or-more-tiered approach via dialogs. Default everything as much as possible.
- Allow the program to be more forgiving. Sometimes an "Are you sure?" dialog with an option to change a list of parameters can be endearing to the user. On the other hand, some users hate this type of checking.

Menus

Designing a Widget hierarchy is especially important when working with Motif menus. Motif menus are a collection of Widgets, so there is no "menu" Widget for a menu. You create menus using a hierarchy of different types of Widgets: RowColumn, PushButton, CascadeButton, ToggleButton, Label, and Separator.

There are three kinds of menus in Motif:

- Pop-up: This appears as a list of items when a pointer button is pressed on a Widget.
- Pull-down: This appears when a button on an existing menu is pressed.
- Option: This enables the user to select from a list of options with the current selection visible at all times.

The procedure to create a menu is different for each type of menu.

Pop-Up Menus

To create a pop-up menu, follow these steps:

1. Include the correct header files. You will need the header files for the menu:
Label.h RowColumn.h PushB.h Separator.h BulletinB.h CascadeB.h
2. Create the menu pane with a call to XmCreatePopupMenu. This is a convenience call to create RowColumn and MenuShell Widgets with the proper settings.
3. Create the buttons on the menu pane. Use XmPushButtons, XmToggleButton, XmSeparator, and XmCascadeButtons.
4. Attach callback functions to the Widgets.

Listing 34.10 shows how to set up a simple pop-up menu.

Listing 34.10. Setting up pop-up menus.

```
/*  
  
** This listing shows how to set up a pop-up menu.  
  
** Use the left mouse button on the area below the label.  
  
*/  
  
#include <X11/Intrinsic.h>  
  
#include <Xm/Xm.h>  
  
#include <Xm/BulletinB.h>  
  
#include <Xm/PushB.h>  
  
#include <Xm/Label.h>  
  
#include <Xm/Text.h>  
  
#include <Xm/RowColumn.h>  
  
#include <Xm/CascadeB.h>  
  
#include <Xm/Separator.h>
```



```
void showSelection(Widget w, XtPointer clientdata, XtPointer calldata);

XtEventHandler showMenu(Widget w, XtPointer menu, XEvent *x, Boolean f);

/**** Some items for the list ****/

#define NUMITEMS 3

char *thisLabel[NUMITEMS] = { "CD", "Developer", "Release" };

/* For the list widget, we need compound strings */

XmString xarray[NUMITEMS];

Widget menu; /* <-- for the menu */

Widget menuBtn[NUMITEMS]; /* <-- for the menu */

#define USE_SCROLL
```

```
void bye(Widget w, XtPointer clientdata, XtPointer calldata);

int main(int argc, char **argv)

{

Widget top;

XtAppContext app;

Widget masterForm;

Widget masterLabel;

Arg args[15];

Widget tmp;

int i;

/**

*** Initialize the toolkit.

**/
```

```
top = XtAppInitialize(&app, "KBH", NULL, 0, (Cardinal *)&argc,

    argv, NULL, args, 0);

/**

*** Create a Bulletin Board on this-top level widget. This is a nice widget

*** to place other widgets on top of.

**/

masterForm = XtVaCreateManagedWidget("bb1",

    xmBulletinBoardWidgetClass, top,

    XmNheight,150,

    XmNwidth,200,

    NULL);

masterLabel = XtVaCreateManagedWidget("Click Me",

    xmLabelWidgetClass, masterForm,
```

```
        XmNheight, 50,

        XmNwidth, 200,

        NULL);

/* XtAddCallback( cancelBtn, XmNactivateCallback, bye, (XtPointer) NULL); */

menu = XmCreatePopupMenu(masterLabel, "menu", NULL, 0);

XtRealizeWidget(menu);

/** Add the event handler for managing the popup */

tmp  = XtCreateManagedWidget("Sample", xmLabelWidgetClass, menu, NULL, 0);

tmp  = XtCreateManagedWidget("Swpl", xmSeparatorWidgetClass, menu, NULL, 0);

tmp  = XtCreateManagedWidget("Btn1", xmPushButtonWidgetClass, menu, NULL, 0);

XtAddCallback(tmp, XmNactivateCallback, showSelection, NULL);

tmp  = XtCreateManagedWidget("Btn2", xmPushButtonWidgetClass, menu, NULL, 0);
```

```
XtAddCallback(tmp,XmNactivateCallback, showSelection, NULL);

tmp  = XtCreateManagedWidget("Btn3", xmPushButtonWidgetClass, menu, NULL, 0);

XtAddCallback(tmp,XmNactivateCallback, showSelection, NULL);


XtAddEventHandler( masterForm, ButtonPressMask, True, (XtEventHandler)showMenu,

    menu);

tmp  = XtCreateManagedWidget("Quit", xmPushButtonWidgetClass, menu, NULL, 0);

XtAddCallback(tmp,XmNactivateCallback, bye, NULL);


/** Add the handler to search here. */

XtRealizeWidget(top);

XtAppMainLoop(app);

return(0);

}
```

```
/**

*** This function will display the pop-up menu on the screen

**/

XtEventHandler showMenu(Widget w, XtPointer menu, XEvent *event, Boolean f)

{

printf ("\n showing position ");

switch (event->xbutton.button)

{

case Button1: printf(" button 1"); break;

case Button2: printf(" button 2"); break;

case Button3: printf(" button 3"); break;

default: printf(" %d", event->xbutton.button); break;

}
```

```

if((event->xbutton.button == Button1)

|| (event->xbutton.button == Button2))

{

    XmMenuPosition(menu, (XButtonPressedEvent *)event);

    printf(" Managing %ld",      (long)menu);

    XtManageChild(menu);

}

}

/**

*** This echos the selection on the controlling terminal

**/

void showSelection(Widget w, XtPointer clientdata, XtPointer calldata)

{

    printf("\n %s ", XtName(w));

```

```

}

void bye(Widget w, XtPointer clientdata, XtPointer calldata)

{

exit(0);

}

```

Note three important items about this listing:

- You can use `printf` functions within Motif applications. The output goes to the controlling terminal by default. This is invaluable in debugging.
- The menu is not visible by itself. An event handler on the menu's parent is registered before the menu can be displayed. This enables the menu to be displayed whenever a button is pressed.
- The `XmMenuPosition` call sets the position of the pop-up menu. It is then managed (that is, after placement).

The Menu Bar

A menu bar is a horizontal bar that is always available to the user. Motif uses the `RowColumn Widget` as a bar with cascading buttons on it for each option.

The procedure for creating a menu bar is as follows:

1. Include the correct header files. You will need the header files for the menu:
`Label.h RowColumn.h PushB.h Separator.h BulletinB.h CascadeB.h`
2. Create the menu bar with a call to `XmCreateMenuBar()`.
3. Create the pull-down menu panes with a call to `XmCreatePulldownMenu()`.
4. For each pull-down pane, create a cascade button on the menu bar. Use the menu bar as the parent. A cascade button is used to link the items in a menu with the menu bar itself.

5. Attach the menu pane to its corresponding cascade button. Use the `XmNsubMenuId` resource of the cascade button to attach the appropriate menu pane.
6. Create the menu entries in the menu panes.

Listing 34.11 shows you how to set up a simple menu application.

Listing 34.11. Creating a menu bar.

```
/*  
  
** This listing shows how to set up a menu bar and pulldown menus  
  
*/  
  
#include <X11/Intrinsic.h>  
  
#include <Xm/Xm.h>  
  
#include <Xm/Form.h>  
  
#include <Xm/PushB.h>  
  
#include <Xm/Label.h>  
  
#include <Xm/Text.h>  
  
#include <Xm/RowColumn.h>  
  
#include <Xm/CascadeB.h>  
  
#include <Xm/Separator.h>
```

```
void doBackup(Widget w, XtPointer x, XtPointer c)

{

printf("\n Start backup ");

}

void doRestore(Widget w, XtPointer x, XtPointer c)

{

printf("\n Start restore ");

}

void bye(Widget w, XtPointer x, XtPointer c)

{

exit(0);

}

void selectTape(Widget w, XtPointer x, XtPointer c)

{
```

```
printf("\n Select tape drive...");

}

void selectDisk(Widget w, XtPointer x, XtPointer c)

{

printf("\n Select Floppy disks...");

}


void helpBtn(Widget w, XtPointer x, XtPointer c)

{

printf("\n ... No help available ");

}


int main (int argc, char*argv[])

{

Widget top;
```

```
XtAppContext app;

Widget masterForm;

Widget menu;      /* for the menu bar */

Widget submenu; /* for pulldown menus */

Widget cascade; /* for cascade button on the menu bar */

Widget filler;

Arg    args[10];

int    n;

/**

*** Initialize the toolkit.

**/

top = XtAppInitialize(&app, "KBH", NULL, 0, (Cardinal *)&argc,

                     argv, NULL, args, 0);

/**
```

```
*** Create a Form on this top-level Widget. This is a nice Widget
```

```
*** to place other Widgets on top of.
```

```
**/
```

```
masterForm = XtVaCreateManagedWidget("Form1",
```

```
    xmFormWidgetClass, top,
```

```
    XmNheight,150,
```

```
    XmNwidth,200,
```

```
    NULL);
```

```
n = 0;
```

```
XtSetArg(args[n],XmNheight, 50); n++;
```

```
menu = XmCreateMenuBar(masterForm,"mastermenu",
```

```
    args, n);
```

```
submenu = XmCreatePulldownMenu(menu,"submenupanel", NULL, 0);
```

```
cascade = XtVaCreateManagedWidget("File",

    xmCascadeButtonWidgetClass, menu,

    XmNsubMenuId, submenu,

    NULL);

filler = XtVaCreateManagedWidget("Backup",

    xmPushButtonWidgetClass, submenu,  NULL);

XtAddCallback( filler, XmNactivateCallback, doBackup, (XtPointer) NULL);

filler = XtVaCreateManagedWidget("Restore",

    xmPushButtonWidgetClass, submenu,  NULL);

XtAddCallback( filler, XmNactivateCallback, doRestore, (XtPointer) NULL);

filler = XtVaCreateManagedWidget("Quit",

    xmPushButtonWidgetClass, submenu,  NULL);
```

```
XtAddCallback( filler, XmNactivateCallback, bye, (XtPointer) NULL);

submenu = XmCreatePulldownMenu(menu, "submenupane2", NULL, 0);

cascade = XtVaCreateManagedWidget("Options",

    xmCascadeButtonWidgetClass, menu,

    XmNsubMenuId, submenu,

    NULL);

filler = XtVaCreateManagedWidget("Tape",

    xmPushButtonWidgetClass, submenu, NULL);

XtAddCallback( filler, XmNactivateCallback, selectTape, (XtPointer) NULL);

filler = XtVaCreateManagedWidget("Floppy",
```

```
        xmPushButtonWidgetClass, submenu,  NULL);

XtAddCallback( filler, XmNactivateCallback, selectDisk, (XtPointer) NULL);


submenu = XmCreatePulldownMenu(menu, "submenupane3", NULL, 0);


cascade = XtVaCreateManagedWidget("Help",

        xmCascadeButtonWidgetClass, menu,

        XmNsubMenuId, submenu,

        NULL);

filler = XtVaCreateManagedWidget("Floppy",

        xmPushButtonWidgetClass, submenu,  NULL);

XtAddCallback( filler, XmNactivateCallback, selectDisk, (XtPointer) NULL);


XtVaSetValues( menu,
```



```

        XmNmenuHelpWidget, cascade,

        NULL);

XtAddCallback( cascade, XmNactivateCallback, helpBtn ,NULL);


XtManageChild(menu);

/** Add the handler to search here. */

XtRealizeWidget(top);

XtAppMainLoop(app);

return(0);

}

```

Note that the Motif programming style requires you to provide the Help button (if you have any) to be right-justified on the menu bar. This Help Cascade button should then be set to the `XmNmenuHelpWidget` of a menu bar. The menu bar will automatically position this Widget to the right-hand side of the visible bar. See Listing 34.12 to learn how to create pull-down menu items on a menu bar.

Listing 34.12. Creating menu bars with pull-down menu items.

```

/*

```

```
** This listing shows how to set up a menu bar and pull-down menus
```

```
*/
```

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

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

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

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

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

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

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

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

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

```
void doBackup(Widget w, XtPointer x, XtPointer c)
```

```
{
```

```
printf("\n Start backup ");
```

```
}
```

```
void doRestore(Widget w, XtPointer x, XtPointer c)
```

```
{
```

```
printf("\n Start restore ");
```

```
}
```

```
void bye(Widget w, XtPointer x, XtPointer c)
```

```
{
```

```
exit(0);
```

```
}
```

```
void selectTape(Widget w, XtPointer x, XtPointer c)
```

```
{
```

```
printf("\n Select tape drive...");
```

```
}
```

```
void selectDisk(Widget w, XtPointer x, XtPointer c)
```

```
{

printf("\n Select Floppy disks...");

}


void helpBtn(Widget w, XtPointer x, XtPointer c)

{

printf("\n ... No help available ");

}


int main (int argc, char*argv[])

{

Widget top;

XtAppContext app;

Widget masterForm;

Widget menu;          /* for the menu bar */
```

```
Widget submenu; /* for pull-down menus */

Widget cascade; /* for CascadeButton on the menu bar */

Widget filler;

Arg    args[10];

int    n;

/**

*** Intialize the toolkit.

**/

top = XtAppInitialize(&app, "KBH", NULL, 0, (Cardinal *)&argc,

                    argv, NULL, args, 0);

/**

*** Create a Form on this top-level Widget. This is a nice Widget

*** to place other Widgets on top of.

**/
```

```
masterForm = XtVaCreateManagedWidget("Form1",

    xmFormWidgetClass, top,

    XmNheight,150,

    XmNwidth,200,

    NULL);

n = 0;

XtSetArg(args[n],XmNheight, 50); n++;

menu = XmCreateMenuBar(masterForm,"mastermenu",

    args, n);

/***** The change is here *****/

n = 0;

XtSetArg(args[n],XmNtearOffModel, XmTEAR_OFF_ENABLED); n++;
```

```
submenu = XmCreatePulldownMenu(menu, "submenupanel", args, n);

/***** The change is here *****/

cascade = XtVaCreateManagedWidget("File",

    xmCascadeButtonWidgetClass, menu,

    XmNsubMenuId, submenu,

    NULL);

filler = XtVaCreateManagedWidget("Backup",

    xmPushButtonWidgetClass, submenu, NULL);

XtAddCallback( filler, XmNactivateCallback, doBackup, (XtPointer) NULL);

filler = XtVaCreateManagedWidget("Restore",

    xmPushButtonWidgetClass, submenu, NULL);

XtAddCallback( filler, XmNactivateCallback, doRestore, (XtPointer) NULL);
```

```
filler = XtVaCreateManagedWidget("Quit",

    xmPushButtonWidgetClass, submenu, NULL);

XtAddCallback( filler, XmNactivateCallback, bye, (XtPointer) NULL);


submenu = XmCreatePulldownMenu(menu, "submenupane2", NULL, 0);


cascade = XtVaCreateManagedWidget("Options",

    xmCascadeButtonWidgetClass, menu,

    XmNsubMenuId, submenu,

    NULL);


filler = XtVaCreateManagedWidget("Tape",

    xmPushButtonWidgetClass, submenu, NULL);

XtAddCallback( filler, XmNactivateCallback, selectTape, (XtPointer) NULL);
```



```
filler = XtVaCreateManagedWidget("Floppy",

    xmPushButtonWidgetClass, submenu,  NULL);

XtAddCallback( filler, XmNactivateCallback, selectDisk, (XtPointer) NULL);


submenu = XmCreatePulldownMenu(menu,"submenupane3", NULL, 0);


cascade = XtVaCreateManagedWidget("Help",

    xmCascadeButtonWidgetClass, menu,

    XmNsubMenuId, submenu,

    NULL);

filler = XtVaCreateManagedWidget("Floppy",

    xmPushButtonWidgetClass, submenu,  NULL);

XtAddCallback( filler, XmNactivateCallback, selectDisk, (XtPointer) NULL);
```

```
XtVaSetValues( menu,

               XmNmenuHelpWidget, cascade,

               NULL );

XtAddCallback( cascade, XmNactivateCallback, helpBtn ,NULL);


XtManageChild(menu);

/** Add the handler to search here. */

XtRealizeWidget(top);

XtAppMainLoop(app);

return(0);

}
```

The Options Menu

An Options menu enables the user to select from a list of items while displaying the most recently selected item. The procedure for creating an Options menu is similar to creating menu bars:

1. Include the correct header files. You will need the header files for the menu:
Label.h RowColumn.h PushB.h Separator.h BulletinB.h CascadeB.h
2. Create the menu bar with a call to `XmCreateOptionsMenu()`.
3. Create the pull-down menu panes with a call to `XmCreatePulldownMenu()`.
4. For each pull-down pane, create a `CascadeButton` on the menu bar.
5. Attach the menu pane to its corresponding `CascadeButton`. Use the `XmNsubMenuId` resource of the `CascadeButton` to attach the appropriate menu pane.
6. Create the menu entries in the menu panes.

Accelerators and Mnemonics

An accelerator for a command is the keystroke that invokes the callback for that particular menu item. For example, for opening a file, you could use Ctrl-O. The resource for this accelerator could be set in the resource file as

```
*Open*accelerator: Ctrl<Key>O
```

The corresponding menu item should read "Open Ctrl+O" to let the user know about this shortcut. Note the + instead of -. You can also set this resource via the command in the `.Xresources` file:

```
*Open*acceleratorText: "Ctrl+O"
```

Using the `.Xresource` file is the preferred way of setting these resources.

Mnemonics are a short form for letting the user select menu items without using the mouse. For example, the user could press <meta>F to invoke the File menu. These are also usually set in the `.Xresource` file. The syntax for the File menu to use the <meta>F key would be this:

```
*File*mnemonic:F
```

TIP: With Linux on your PC, meta means the Alt key.

Dialog Boxes

A dialog box is used to convey information about something to the user and requests a canned response in return. For

example, a dialog box might say "Go ahead and Print" and present three buttons--OK, Cancel, and Help. The user must then select one of the three buttons to process the command.

A typical dialog box displays an icon, a message string, and usually three buttons (OK, Cancel, and Help). Motif provides predefined dialog boxes for the following categories:

- Errors
- Information
- Warnings
- Working
- Questions

Each of these dialog box types displays a different icon: a question mark for the Question dialog box, an exclamation mark for an Information dialog box, and so on. The following convenience functions facilitate the creation of dialog boxes:

- XmCreateErrorsDialog
- XmCreateInformationDialog
- XmCreateWarningDialog
- XmCreateWorkingDialog
- XmCreateQuestionDialog

The infamous "OK to quit?" dialog box can be implemented as shown in Listing 34.13. There is another example in Listing 34.17.

Listing 34.13. Code fragment to confirm quit command.

```
/*

** Confirm quit one last time

*/

void reallyQuit(Widget w, XtPointer clientdata, XtPointer calldata)

{
```

```
exit(0);

}

void confirmQuit(Widget w, XtPointer clientdata, XtPointer calldata)

{

static Widget quitDlg = NULL;

static char *msgstr = "Really Quit?"


if (quitDlg != NULL)

{

/* first time called */

quitDlg = XmCreateQuestionDialog(w, "R U Sure", NULL, 0);

XtVaSetValues(quitDlg, XtVaTypedArg,

               XmNmessageString,

               XmRString,

               msgstr,
```

```

        strlen(msgstr),

        NULL);

XtAddCallback(quitDlg, reallyQuit, NULL);

}

XtManageChild(quitDlg);

}

```

Append this code fragment to the end of any sample listings in this chapter to get instant checking before you actually quit the application. Note that the `quitDlg` dialog box is set to `NULL` when the function is first called. It does not have to be re-created on every call after the first one; it is only managed for all subsequent calls to this function.

Modes of a Dialog Box

A dialog box can have four modes of operation, called modality. The mode is set in the `XmNdialogStyle` resource. These are the possible values:

- Nonmodal: The user can ignore the dialog box and work with any other window on the screen. Resource value is `XmDIALOG_MODELESS`.
- Primary Application Modal: All input to the window that invoked the dialog box is locked out. The user can use the rest of the windows in the application. Resource value is `XmDIALOG_PRIMARY_APPLICATION_MODAL`.
- Full Application Modal: All input to all the windows in the application that invoked the dialog box is locked out. The user cannot use the rest of the windows in the application. Resource value is `XmDIALOG_FULL_APPLICATION_MODAL`.
- System Modal: All input is directed to the dialog box. The user cannot interact with any other window in the system. Resource value is `XmDIALOG_SYSTEM_MODAL`.

The dialog boxes provided by Motif are based on the `XmMessageBox` Widget. Sometimes it is necessary to get to the Widgets in a dialog box. This is done with a call to `XmMessageBox GetChild(Widget dialog, typeOfWidget)`; where `typeOfWidget` can be one of the following:

```
XmDIALOG_HELP_BUTTON XmDIALOG_CANCEL_BUTTON
XmDIALOG_SEPARATOR   XmDIALOG_MESSAGE_LABEL
XmDIALOG_OK_BUTTON   XmDIALOG_SYMBOL_LABEL
```

The dialog box may have more Widgets that can be addressed. Check the man pages for the descriptions of these Widgets. For example, to hide the Help button on a dialog box, use the call

```
XtUnmanageChild(XmMessageBoxGetChild(dlg, XmDIALOG_HELP_BUTTON));
```

or, in the case of adding a callback, use

```
XtAddCallback(XmMessageBoxGetChild(dlg, XmDIALOG_OK_BUTTON),

              XmnActivateCallback, yourFunction);
```

A typical method of creating custom dialog boxes is to use existing ones. Then, using the `XmMessageBoxGetChild` function, you can add or remove any function you want. For example, by replacing the message string Widget with a Form Widget, you have a place for laying out Widgets however you need to.

Events

An event is a message sent from the X server to the application that some condition in the system has changed. This could be a button press, a keystroke, requested information from the server, or a timeout. An event is always relative to a window and starts from the bottom up. It propagates up the window hierarchy until it gets to the root window, where the root window application makes the decision to either use or discard it. If an application in the hierarchy does use the event or does not allow upward propagation of events, the message is used at the window itself. Only device events are propagated upward (such as keyboard or mouse)--not configuration events.

An application must request an event of a particular type before it can begin receiving events. Each Motif application calls `XtAppInitialize` to create this connection automatically.

Events contain at least the following information:

- The type of event
- The display where it happened
- The window of the event, called the event window

- The serial number of the last event processed by the server

Look in the file `<X11/Xlib.h>` for a description of the union called `XEvent`, which enables access to these values. The file `<X11/X.h>` contains the descriptions of constants for the types of events. All event types share the header:

```
typedef struct {

    int type;

    unsigned long serial;    /* # of last request processed by server */

    Bool send_event;    /* true if this came from a SendEvent request */

    Display *display; /* display the event was read from */

    Window window;    /* window on which event was requested in event mask */

} XAnyEvent;
```

Expose Events

The server generates an Expose when a window that was covered by another is brought to the top of the stack, or even partially exposed. The structure for this event type is

```
typedef struct {

    int type;                /* type of event */

    unsigned long serial;    /* # of last request processed by server */

    Bool send_event;    /* true if this came from a SendEvent request */

    Display *display; /* display the event was read from */

}
```



```

Window window;

int x, y;

int width, height;

int count;          /* if nonzero, at least this many more */

} XExposeEvent;

```

Note how the first five fields are shared between this event and `XAnyEvent`. Expose events are guaranteed to be in sequence. An application may get several Expose events from one condition. The `count` field keeps a count of the number of Expose events still in the queue when the application receives this one. Thus, it can be up to the application to wait to redraw until the last Expose event is received (that is, `count == 0`).

Pointer Events

A Pointer event is generated by a mouse press, release, or movement. The type of event is called `XButtonEvent`. Recall that the leftmost button is `Button1`, but it can be changed. This is the structure returned by this button press and release:

```

typedef struct {

    int type;          /* of event */

    unsigned long serial; /* # of last request processed by server */

    Bool send_event; /* true if this came from a SendEvent request */

    Display *display; /* display the event was read from */

    Window window;      /* "event" window it is reported relative to */

```

```

Window root;                /* root window that the event occurred on */

Window subwindow; /* child window */

Time time;                /* milliseconds */

int x, y;                  /* pointer x, y coordinates in event window */

int x_root, y_root;        /* coordinates relative to root */

unsigned int state;         /* key or button mask */

unsigned int button;        /* detail */

Bool same_screen;          /* same screen flag */

} XButtonEvent;

typedef XButtonEvent XButtonPressedEvent;

typedef XButtonEvent XButtonReleasedEvent;

```

The event for a movement is called `XMotionEvent`, with the type field set to `MotionNotify`:

```

typedef struct {

    int type;                /* MotionNotify */

```

```
    unsigned long serial;    /* # of last request processed by server */

    Bool send_event; /* true if this came from a SendEvent request */

    Display *display; /* display the event was read from */

    Window window;      /* "event" window reported relative to */

    Window root;         /* root window that the event occurred on */

    Window subwindow; /* child window */

    Time time;           /* milliseconds */

    int x, y;            /* pointer x, y coordinates in event window */

    int x_root, y_root;  /* coordinates relative to root */

    unsigned int state;  /* key or button mask */

    char is_hint;        /* detail */

    Bool same_screen; /* same screen flag */

} XMotionEvent;

typedef XMotionEvent XPointerMovedEvent;
```

Keyboard Events

A keyboard event is generated when the user presses or releases a key. Both types of events, `KeyPress` and `KeyRelease`, are returned in a `XKeyEvent` structure:

```
typedef struct {

    int type;                /* of event */

    unsigned long serial;    /* # of last request processed by server */

    Bool send_event;        /* true if this came from a SendEvent request */

    Display *display;        /* display the event was read from */

    Window window;           /* "event" window it is reported relative to */

    Window root;            /* root window that the event occurred on */

    Window subwindow;        /* child window */

    Time time;              /* milliseconds */

    int x, y;               /* pointer x, y coordinates in event window */

    int x_root, y_root;      /* coordinates relative to root */

    unsigned int state;      /* key or button mask */

    unsigned int keycode;    /* detail */

    Bool same_screen;        /* same screen flag */
```

```
} XKeyEvent;
```

```
typedef XKeyEvent XKeyPressedEvent;
```

```
typedef XKeyEvent XKeyReleasedEvent;
```

The `keycode` field gives information on whether the key was pressed or released. These constants are defined in `<X11/keysymdef.h>` and are vendor-specific. These are called `KeySym` and are generic across all X servers. For example, the F1 key could be described as `XK_F1`. The function `XLookupString` converts a `KeyPress` event into a string and a `KeySym` (a portable key symbol). The call is

```
int XLookupString(XKeyEvent *event,

                 char *returnString,

                 int max_length,

                 KeySym *keysym,

                 XComposeStatus *compose);
```

The returned ASCII string is placed in `returnString` for up to `max_length` characters. The `keysym` contains the key symbol. Generally, the `compose` parameter is ignored.

Window Crossing Events

The server generates crossing `EnterNotify` events when a pointer enters a window and `LeaveNotify` events when a pointer leaves a window. These are used to create special effects for notifying the user that the window has focus. The `XCrossingEvent` structure looks like the following:

```
typedef struct {
```

```

int type;          /* of event */

unsigned long serial; /* # of last request processed by server */

Bool send_event; /* true if this came from a SendEvent request */

Display *display; /* display the event was read from */

Window window;      /* "event" window reported relative to */

Window root;         /* root window that the event occurred on */

Window subwindow; /* child window */

Time time;           /* milliseconds */

int x, y;            /* pointer x, y coordinates in event window */

int x_root, y_root; /* coordinates relative to root */

int mode;            /* NotifyNormal, NotifyGrab, NotifyUngrab */

int detail;

/*

* NotifyAncestor, NotifyVirtual, NotifyInferior,

* NotifyNonlinear, NotifyNonlinearVirtual

```

```

    */

    Bool same_screen; /* same screen flag */

    Bool focus;        /* boolean focus */

    unsigned int state; /* key or button mask */

} XCrossingEvent;

typedef XCrossingEvent XEnterWindowEvent;

typedef XCrossingEvent XLeaveWindowEvent;

```

These are generally used to color a window on entry and exit to provide feedback to the user as he moves the pointer around.

Event Masks

An application requests events of a particular type by calling the `XAddEventHandler()` function. The prototype for this function is

```

XAddEventHandler( Widget ,

                EventMask ,

                Boolean maskable,

                XtEventHandler handlerfunction,

                XtPointer clientData);

```

The handler function is of the form

```
void handlerFunction( Widget w, XtPointer clientData,

                    XEvent *ev, Boolean *continueToDispatch);
```

The first two arguments are the `clientdata` and `Widget` passed in `XtAddEventHandler`. The `ev` argument is the event that triggered this call. The last argument enables this message to be passed to other message handlers for this type of event. This should be defaulted to `TRUE`.

You would use the following call on a `Widget w` to be notified of all pointer events of the type `ButtonMotion` and `PointerMotion` on this `Widget`.

```
extern void handlerFunction( Widget w, XtPointer clientData,

                            XEvent *ev, Boolean *continueToDispatch);

XtAddEventHandler( w, ButtonMotionMask | PointerMotionMask, FALSE,

                  handlerFunction, NULL );
```

These are the possible event masks:

- `NoEventMask`
- `KeyPressMask`
- `KeyReleaseMask`
- `ButtonPressMask`
- `ButtonReleaseMask`

- EnterWindowMask
- LeaveWindowMask
- PointerMotionMask
- PointerMotionHintMask
- Button1MotionMask
- Button2MotionMask
- Button3MotionMask
- Button4MotionMask
- Button5MotionMask
- ButtonMotionMask
- KeymapStateMask
- ExposureMask
- VisibilityChangeMas
- StructureNotifyMask
- ResizeRedirectMask
- SubstructureNotifyMask
- SubstructureRedirectMask
- FocusChangeMask
- PropertyChangeMask
- ColormapChangeMask
- OwnerGrabButtonMask

Listing 34.14 is a sample application that shows how to track the mouse position.

Listing 34.14. Tracking a pointer.

*/ **

```
** This application shows how to track a pointer

*/

#include <X11/Intrinsic.h>

#include <Xm/Xm.h>

#include <Xm/Form.h>

#include <Xm/PushB.h>

#include <Xm/Label.h>


static int track;


void upMouse(Widget w, XtPointer clientdata, XEvent *x, Boolean *f)

{

track = 0;

}
```

```
void  moveMouse(Widget w, XtPointer clientdata, XEvent *x, Boolean *f)

{

    if (track == 1)

        {

            printf("\n x: %d, y: %d", x->xmotion.x, x->xmotion.y);

        }

}

void  downMouse(Widget w, XtPointer clientdata, XEvent *x, Boolean *f)

{

    track = 1;

}

void  bye(Widget w, XtPointer clientdata, XtPointer calldata);

int  main(int  argc, char **argv)
```

```
{

Widget top;

XtAppContext app;

Widget aForm;

Widget aLabel;

Widget aButton;

Arg    args[5];

/**

*** Initialize the toolkit.

**/

top = XtAppInitialize(&app, "KBH", NULL, 0, (Cardinal *)&argc,

                    argv, NULL, args, 0);

/**
```

```
*** Create a Form on this top-level Widget. This is a nice Widget

*** to place other Widgets on top of.

**/

aForm = XtVaCreateManagedWidget("Form1",

    xmFormWidgetClass, top,

    XmNheight,90,

    XmNwidth,200,

    NULL);

/**

*** Add a button on the form you just created. Note how this Button

*** Widget is connected to the form that it resides on. Only

*** left, right, and bottom edges are attached to the form. The

*** top edge of the button is not connected to the form.

**/
```

```
aButton = XtVaCreateManagedWidget("Push to Exit",

    xmPushButtonWidgetClass, aForm,

    XmNheight,20,

    XmNleftAttachment,XmATTACH_FORM,

    XmNleftOffset,20,

    XmNrightAttachment,XmATTACH_FORM,

    XmNrightOffset,20,

    XmNbottomAttachment,XmATTACH_FORM,

    NULL);

/**

*** Now let's create the label for us.

*** The alignment is set to right-justify the label.

*** Note how the label is attached to the parent form.

**/
```

```
aLabel = XtVaCreateManagedWidget("This is a Label",

    xmLabelWidgetClass, aForm,

    XmNalignment, XmALIGNMENT_END,

    XmNleftAttachment, XmATTACH_FORM,

    XmNrightAttachment, XmATTACH_FORM,

    XmNtopAttachment, XmATTACH_FORM,

    XmNbottomAttachment, XmATTACH_WIDGET,

    XmNbottomWidget, aButton,

    NULL);

/**

*** Now add the event handlers for tracking the mouse on the

*** label. Note that the LeaveWindowMask is set to release

*** the pointer for you should you keep the button pressed
```

```
*** and leave the window.

**/

XtAddEventHandler( aLabel, ButtonPressMask, FALSE, downMouse, NULL);

XtAddEventHandler( aLabel, ButtonMotionMask, FALSE, moveMouse, NULL);

XtAddEventHandler( aLabel, ButtonReleaseMask | LeaveWindowMask,

                  FALSE, upMouse, NULL);

/**

*** Call the function "bye" when the PushButton receives

*** an activate message; i.e. when the pointer is moved to

*** the button and Button1 is pressed and released.

**/

XtAddCallback( aButton, XmNactivateCallback, bye, (XtPointer) NULL);

XtRealizeWidget(top);

XtAppMainLoop(app);

return(0);
```



```

}

void bye(Widget w, XtPointer clientdata, XtPointer calldata)

{

exit(0);

}

```

Managing the Queue

Managing the X server is critical if you have to handle a large number of incoming events. The `XtAppMainLoop()` function handles all the incoming events via the following functions:

- `XtAppPending` checks the queue to see if any events are pending.
- `XtAppNextEvent` removes the next event from the queue.
- `XtDispatchEvent` passes the message to the appropriate window.

The loop can do something else between checking and removing messages via the replacement code segment:

```

while (!done)

{

while (XtAppPending( applicationContext))

{

```

```

        XtAppNextEvent( applicationContext, &ev));

        XtDispathEvent( &ev));

    }

    done = interEventFunction();

}

```

There are some caveats with this scheme:

- This is a nonblocking function. It must be fed at all times with events, or it will take over all other applications' time.
- There is no guarantee when your inter-event function will be run if the queue is flooded with events.
- Note the `while` loop for checking messages. It's more efficient to flush the queue first and then call your function rather than calling it once every time you check for messages.
- The inter-event function must be fast or you will see the user interface slow down. If you want to give some response back to your user about what's going on while in a long inter-event function, you can make a call to `XmUpdateDisplay(Display *)`. This will handle only the Expose events in the queue so that you can update some status display.

TIP: Consider using the `select` call to handle incoming events of file descriptors. This is a call that enables an application to wait for events from various file descriptors on read-ready, write-ready, or both. The file descriptors can be sockets, too! See the man page for more information on the `select` call. Open all the files with an `open` call. Get the file descriptor for the event queue. Use the `Select` macros to set up the parameters for `select` call `ret = return from the select function: switch (ret) case 0: process the event queue case 1: ... process the file descriptor`

Work Procedures

These are functions called by the event-handler loop whenever no events are pending in the queue. The function is expected to return a Boolean value indicating whether it has to be removed from the loop after it is called. If `TRUE`, it wants to be removed; if `FALSE`, it wants to be called again. For example, you could set up a disk file transfer to run in the background, which will keep returning `FALSE` until it is done, at which time it will return `TRUE`.

The work procedures are defined as

```
Boolean yourFunction(XtPointer clientdata);
```

The way to register a work procedure is to call

```
XtWorkProcId      XtAppAddWorkProc ( XtAppContext app,

                                     XtWorkProc  functionPointer,

                                     XtPointer    clientData);
```

The return ID from this call is the handle to the work procedure. It is used to remove the work procedure with a call to the function `XtRemoveWorkProc(XtWorkProcId id);`

Using Timeouts

A timeout is used to perform some task at (almost) regular intervals. Applications set up a timer callback function that is called when a requested time interval has passed. This function is defined as

```
void thyTimerCallback( XtPointer clientdata, XtInterval *tid);
```

where `clientdata` is a pointer to client-specific data. The setup function for the timeout returns the timer ID and is defined as

```
XtIntervalId XtAddTimeOut ( XtAppContext app,

                           int milliseconds,

                           XtTimerCallback TimerProcedure,

                           XtPointer clientdata);
```

This call sets up a timer to call the `TimerProcedure` function when the requested milliseconds have passed. It will

do this only once. If you want cyclic timeouts--for example, in a clock application--you have to explicitly set up the next function call in the timer handler function itself. So generally, the last line in a timer handler is a call to set a timeout for the next time the function wants to be called.

Linux is not designed for real-time applications, and you can't expect a deterministic time interval between successive timer calls. Some heavy graphics updates can cause delays in the timer loop. For user-interface applications, the delays are probably not a big drawback; however, consult your vendor before you attempt to write a time-critical control application. Depending on your application, your mileage may vary. See Listing 34.15 for an example of setting up cyclic timers.

Listing 34.15. Setting up cyclic timers.

```
/*

** This application shows how to set a cyclic timer

*/

#include <X11/Intrinsic.h>

#include <Xm/Xm.h>

#include <Xm/Form.h>

#include <Xm/PushB.h>

#include <Xm/Text.h>

int counter;

char buf[32];
```

```
#define ONE_SECOND 1000L /* **APPROXIMATELY** 1000 milliseconds..*/

/* Timing is *not* precise in Motif... so do not rely on this time

** for a time-critical application. Use interrupt handlers instead.

*/

void makeTimer(Widget w, XtIntervalId id);

void bye(Widget w, XtPointer clientdata, XtPointer calldata);

int main(int argc, char **argv)

{

Widget top;

XtAppContext app;

Widget aForm;

Widget aText;
```

```
Widget aButton;

Arg    args[5];


/**

*** Initialize the toolkit.

**/

top = XtAppInitialize(&app, "KBH", NULL, 0, (Cardinal *)&argc,

                     argv, NULL, args, 0);


/**

*** Create a Form on this top-level Widget. This is a nice Widget

*** to place other Widgets on top of.

**/

aForm = XtVaCreateManagedWidget("Form1",

                                 xmFormWidgetClass, top,
```

```
        XmNheight,90,

        XmNwidth,200,

        NULL);

/**

*** Add a button on the form you just created. Note how this Button

*** Widget is connected to the form that it resides on. Only

*** left, right, and bottom edges are attached to the form. The

*** top edge of the button is not connected to the form.

**/

aButton = XtVaCreateManagedWidget("Push to Exit",

        xmPushButtonWidgetClass, aForm,

        XmNheight,20,

        XmNleftAttachment,XmATTACH_FORM,

        XmNleftOffset,20,
```

```
        XmNrightAttachment,XmATTACH_FORM,

        XmNrightOffset,20,

        XmNbottomAttachment,XmATTACH_FORM,

        NULL);

/**

*** Now let's create the label for us.

*** The alignment is set to right-justify the label.

*** Note how the label is attached to the parent form.

**/

aText = XtVaCreateManagedWidget("This is a Label",

        xmTextWidgetClass, aForm,

        XmNalignment, XmALIGNMENT_CENTER,

        XmNleftAttachment,XmATTACH_FORM,
```



```
        XmNrightAttachment,XmATTACH_FORM,

        XmNtopAttachment,XmATTACH_FORM,

        XmNbottomAttachment,XmATTACH_WIDGET,

        XmNbottomWidget,aButton,

        NULL);

/**

*** Now add the timer handler

**/

counter = 0;

makeTimer(aText, (XtIntervalId )NULL);

/**

*** Call the function "bye" when the PushButton receives

*** an activate message; i.e. when the pointer is moved to
```

```
*** the button and Button1 is pressed and released.

**/

XtAddCallback( aButton, XmNactivateCallback, bye, (XtPointer) NULL);

XtRealizeWidget(top);

XtAppMainLoop(app);

return(0);

}

void bye(Widget w, XtPointer clientdata, XtPointer calldata)

{

exit(0);

}

/** This function creates the timer code. */

void makeTimer(Widget w, XtIntervalId id)
```

```
{

Widget tmp;

extern int counter;


sprintf(buf, "%4d", counter++);

XmTextSetString(w, buf);


if (counter < 10) /** reinvoke yourself if < 10 times **/

XtAppAddTimeOut( XtWidgetToApplicationContext(w), ONE_SECOND,

                (XtTimerCallbackProc)makeTimer, (XtPointer)w);

}
```

Handling Other Sources

The `XtAddInput` function is used to handle inputs from sources other than the event queue. The definition is

```
XtInputId XtAddInput( XtAppContext  app,

                    int             LinuxfileDescriptor,
```

```

        XtPointer    condition,

        XtInputCallback    inputHandler,

        XtPointer    clientdata);

```

The return value from this call is the handle to the `inputHandler` function. This is used to remove the call via the call

```
XtRemoveInput( XtInput Id);
```

The `inputHandler` function itself is defined as

```
void InputHandler(XtPointer clientdata, int *fd, XtInputId *id);
```

Unlike timers, you have to register this function only once. Note that a pointer to a file descriptor is passed in to the function. The file descriptor must be a Linux file descriptor. You do not have support for Linux IPC message queues or semaphores through this scheme. The IPC mechanism is considered dated and is limited to one machine. Consider using sockets instead.

NOTE: AIX enables pending on message queues via the `select` call. Look at the AIX man pages for this call.

The Graphics Context

Each Widget draws itself on the screen using its set of drawing parameters called the graphics context (GC). For drawing on a Widget, you can use the X primitive functions if you have its window and its graphics context. It's easier to limit your artwork to the `DrawingArea` Widget, which is designed for this purpose. You can think of the GC as your paintbrush and the Widget as the canvas. The color and thickness of the paintbrush are just some of the factors that determine how the paint is transferred to the canvas.

The function call to create a GC is

```
GC XCreateGC (Display dp, Drawable d, unsigned long mask, XGCValue *values);
```

For use with a Widget `w`, this call would look like this:

```
GC gc;

XGCValue gcv;

unsigned long mask;

gc = XCreate(XtDisplay(w), DefaultRootWindow(XtDisplay(w)),

            mask, gcv);
```

Also, you can create a GC for a Widget directly with a call to `XtGetGC()`. The prototype for this function is

```
gc = XtGetGC (Widget w, unsigned long mask, XGCValue *values);
```

The values for the mask parameter are defined as an ORed value of the following definitions:

- `GCFunction`
- `GCPlaneMask`
- `GCForeground`
- `GCBackground`
- `GCLineWidth`
- `GCLineStyle`
- `GCCapStyle`
- `GCJoinStyle`
- `GCFillStyle`

- GCFillRule
- GCTile
- GCStipple
- GCTileStipXOrigin
- GCTileStipYOrigin
- GCFont
- GCSubWindowMode
- GCGraphicsExposures
- GCClipXOrigin
- GCClipYOrigin
- GCClipMask
- GCDashOffset
- GCDashList
- GCArcMode

So, if a call is going to set the Font and Clipping mask, the value of the mask will be `(GCFont | GCClipMask)`. The data structure for setting the graphics context is as follows:

```
typedef struct {

    int function;           /* logical operation */

    unsigned long plane_mask; /* plane mask */

    unsigned long foreground; /* foreground pixel */

    unsigned long background; /* background pixel */

    int line_width;         /* line width */
```

```
int line_style;          /* LineSolid, LineOnOffDash, LineDoubleDash */

int cap_style;           /* CapNotLast, CapButt,

                          CapRound, CapProjecting */

int join_style;          /* JoinMiter, JoinRound, JoinBevel */

int fill_style;          /* FillSolid, FillTiled,

                          FillStippled, FillOpaqueStippled */

int fill_rule;           /* EvenOddRule, WindingRule */

int arc_mode;            /* ArcChord, ArcPieSlice */

Pixmap tile;             /* tile pixmap for tiling operations */

Pixmap stipple;          /* stipple 1 plane pixmap for stippling */

int ts_x_origin;         /* offset for tile or stipple operations */

int ts_y_origin;

Font font;               /* default text font for text operations */

int subwindow_mode;      /* ClipByChildren, IncludeInferiors */

Bool graphics_exposures; /* boolean, should exposures be generated */
```

```

int clip_x_origin;      /* origin for clipping */

int clip_y_origin;

Pixmap clip_mask; /* bitmap clipping; other calls for rects */

int dash_offset; /* patterned/dashed line information */

char dashes;

} XGCValues;

```

If you want to set a value in a GC, you have to take two steps before you create the GC:

1. Set the value in the `XGCValue` structure.
2. Set the mask for the call.

Let's look at the values of the functions in a bit more detail.

GCFunction

This determines how the GC paints to the screen. The `dst` pixels are the pixels currently on the screen, and the `src` pixels are those that your application is writing using the GC.

```

GXclear dst = 0

GXset    dst = 1

GXand          dst = src AND dst

Gxor  dst = src OR dst

```



```
GXcopy      dst = src
```

```
GXnoop      dst = dst
```

```
Gxnor dst = NOT(src OR dst)
```

```
Gxxor dst = src XOR dst
```

```
GXinvert dst = NOT dst
```

```
GxcopyInverted dst = NOT src
```

The function for a GC is changed via a call to `XSetFunction (Display *dp, GC gc, int function)`, where `function` is set to one of the values just mentioned. The default value is `GXcopy`. There are several other masks that you can apply. They are listed in the `<X11/X.h>` file.

GCPlaneMask

The plane mask sets which planes of a drawable can be set by the GC. This is defaulted to `AllPlanes`, thereby enabling the GC to work with all planes on a Widget.

GCForeground and GCBackground

These are the values of the pixels to use for the foreground and background colors, respectively. The call to manipulate these is

```
XSetForeground(Display *dp, GC gc, Pixel pixel);
```

```
XSetBackGround(Display *dp, GC gc, Pixel pixel);
```

GCLineWidth

This is the number of pixels for the width of all lines drawn via the GC. It is defaulted to zero, which is the signal to the server to draw the thinnest line possible.

```
GCLineStyle GCDashOffset GCDashList
```

This determines the style of the line drawn on-screen. `LineSolid` draws a solid line using the foreground color, `LineOnOffDash` draws an intermittent line with the foreground color, and `LineDoubleDash` draws a line that is composed of interlaced segments of the foreground and background colors. The `GCDashOffset` and `GCDashList` values determine the position and length of these dashes.

GCCapStyle

This determines how the server draws the ends of lines. `CapNotLast` draws up to, but does not include, the endpoint pixels of a line; `CapButt` draws up to the endpoints of a line (inclusive); `CapRound` tries to round off the edges of a thick line (three or more pixels wide); and `CapProjecting` extends the endpoint a little.

GCJoinStyle

This is used to draw the endpoints of a line. It can be set to `JointMiter` for a 90-degree joint, `JoinBevel` for a beveled joint, or `JoinRound` for a rounded joint.

```
GCFillStyle, GCTile, GCStipple
```

The fill style can be set to `FillSolid`, which specifies the fill color to be the foreground color; `FillTiled` specifies a pattern of the same in the `Tile` attribute; and `FillStipple` specifies a pattern in the `Stipple` attribute. `FillStipple` uses the foreground color where a bit is set to 1 and nothing when a bit is set to 0, whereas `FillOpaqueStippled` uses the foreground color when a bit is set to 1 and the background color when a bit is set to 0.

GCFont

This specifies the fontlist to use. (See the section "Using Fonts and FontLists," later in this chapter.)

`GCArcMode`

This defines the way an arc is drawn on-screen (see the next section).

Drawing Lines, Points, Arcs, and Polygons

Motif applications can access all the graphics primitives provided by `Xlib`. All `Xlib` functions must operate on a window or a `Pixmap`; both are referred to as a drawable. Widgets have a window after they are realized. You can access this window with a call to `XtWindow()`. An application can crash if `Xlib` calls are made to a window that is not realized. The way to check is via a call to `XtIsRealized()` on the `Widget`, which will return `TRUE` if it is realized and `FALSE` if it is not. Use the `XmDrawingArea` `Widget`'s callbacks for rendering your graphics, because it is designed for this purpose. The callbacks available to you are:

- `XmNresizeCallback`: Invoked when the `Widget` is resized.
- `XmNexposeCallback`: Invoked when the `Widget` receives an `Expose` event.
- `XmNinputCallback`: Invoked when a button or key is pressed on the `Widget`.

All three functions pass a pointer to the `XmDrawingAreaCallbackStruct`.

Drawing a Line

To draw a line on-screen, use the `XDrawLine` or `XDrawLines` function call. Consider the example shown in Listing 34.16.

Listing 34.16. Drawing lines and points.

```
/*

** This application shows how to draw lines and points

** by tracking the pointer

*/

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

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

#include <Xm/Form.h>

#include <Xm/PushB.h>

#include <Xm/DrawingA.h>


/**

*** used for tracking and drawing via the mouse

**/

static int track;

static int lastx;

static int lasty;

static GC thisGC;

XGCValues gcv;

Widget aForm;

Widget aDraw;
```

```
Widget aButton;

/**

*** Connect the mouse down and up movements

**/

void upMouse(Widget w, XtPointer clientdata, XEvent *e, Boolean *f)

{

    if (track == 1)

    {

        XDrawLine(XtDisplay(w), XtWindow(w),

                  thisGC, lastx, lasty, e->xbutton.x, e->xbutton.y);

    }

    track = 0;

}
```

```
void lostMouse(Widget w, XtPointer clientdata, XEvent *x, Boolean *f)

{

track = 0;

}

/**

*** This function tracks the movement of the mouse by

*** drawing points on its location while a button is

*** pressed.

**/

void moveMouse(Widget w, XtPointer clientdata, XEvent *e, Boolean *f)

{

if (track == 1)

{

printf("\n x: %d, y: %d", e->xmotion.x, e->xmotion.y);
```

```
        XDrawPoint(XtDisplay(w),XtWindow(w),

                    thisGC, e->xmotion.x, e->xmotion.y);

    }

}

void  downMouse(Widget w, XtPointer clientdata, XEvent *e, Boolean *f)

{

    track = 1;

    lastx = e->xbutton.x;

    lasty = e->xbutton.y;

}

void  bye(Widget w, XtPointer clientdata, XtPointer calldata);

int main(int  argc, char **argv)

{
```

```
Widget top;

XtAppContext app;

Arg    args[5];


/**

*** Initialize the toolkit.

**/

top = XtAppInitialize(&app, "KBH", NULL, 0, (Cardinal *)&argc,

                     argv, NULL, args, 0);


/**

*** Create a Form on this top-level Widget. This is a nice Widget

*** to place other Widgets on top of.

**/

aForm = XtVaCreateManagedWidget("Form1",
```



```
    xmFormWidgetClass, top,

    XmNheight, 200,

    XmNwidth, 200,

    NULL);

/**

*** Add a button on the form you just created. Note how this Button

*** Widget is connected to the form that it resides on. Only

*** left, right, and bottom edges are attached to the form. The

*** top edge of the button is not connected to the form.

**/

aButton = XtVaCreateManagedWidget("Push to Exit",

    xmPushButtonWidgetClass, aForm,

    XmNheight, 20,

    XmNleftAttachment, XmATTACH_FORM,
```

```
        XmNleftOffset, 20,

        XmNrightAttachment, XmATTACH_FORM,

        XmNrightOffset, 20,

        XmNbottomAttachment, XmATTACH_FORM,

        NULL);

/**

*** Now let's create the label for us.

*** The alignment is set to right-justify the label.

*** Note how the label is attached to the parent form.

**/

aDraw = XtVaCreateManagedWidget("paperusitto",

        xmDrawingAreaWidgetClass, aForm,

        XmNalignment, XmALIGNMENT_END,
```

```

XmNleftAttachment,XmATTACH_FORM,

XmNrightAttachment,XmATTACH_FORM,

XmNtopAttachment,XmATTACH_FORM,

XmNbottomAttachment,XmATTACH_WIDGET,

XmNbottomWidget,aButton,

NULL);

```

```

gcv.foreground = BlackPixel(XtDisplay(aDraw), DefaultScreen(XtDisplay(aDraw)));

gcv.background = WhitePixel(XtDisplay(aDraw), DefaultScreen(XtDisplay(aDraw)));

gcv.line_width = 2;

thisGC = XtGetGC( aDraw,

GCForeground | GCBackground | GCLineWidth,

(XGCValues *) &gcv);

/**

```

```
*** Now add the event handlers for tracking the mouse on the

*** label. Note that the LeaveWindowMask is set to release

*** the pointer for you should you keep the button pressed

*** and leave the window and disables tracking.

**/

XtAddEventHandler( aDraw, ButtonPressMask, FALSE, downMouse, NULL);

XtAddEventHandler( aDraw, ButtonMotionMask, FALSE, moveMouse, NULL);

XtAddEventHandler( aDraw, ButtonReleaseMask,FALSE, upMouse, NULL);

XtAddEventHandler( aDraw, LeaveWindowMask, FALSE, lostMouse, NULL);

/**

*** Call the function "bye" when the PushButton receives

*** an activate message; i.e. when the pointer is moved to

*** the button and Button1 is pressed and released.

**/

XtAddCallback( aButton, XmNactivateCallback, bye, (XtPointer) NULL);
```

```
XtRealizeWidget(top);
```

```
XtAppMainLoop(app);
```

```
return(0);
```

```
}
```

```
void reallyQuit(Widget w, XtPointer clientdata, XtPointer calldata)
```

```
{
```

```
exit(0);
```

```
}
```

```
/**
```

```
*** pesky quit routine
```

```
**/
```

```
void bye(Widget w, XtPointer clientdata, XtPointer calldata)
```

```

{

static Widget quitDlg = NULL;

static char *msgstr = "Are you sure you want to Quit?";


if (quitDlg != (Widget )NULL)

{

    /* first time called */

    quitDlg = XmCreateQuestionDialog(w,"R U Sure", (XtPointer *)NULL,0);

    XtVaSetValues(quitDlg,

        XmNdialogStyle, XmDIALOG_FULL_APPLICATION_MODAL,

        XtVaTypedArg, XmNmessageString, XmRString,

        msgstr,

        strlen(msgstr),

        NULL);

```

```

        XtAddCallback(quitDlg, XmNokCallback, reallyQuit, NULL);

    }

XtManageChild(quitDlg);

}

```

The output from Listing 34.16 is shown in Figure 34.11.

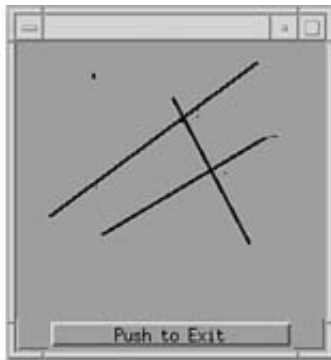


Figure 34.11. *Drawing points and lines.*

This is an example of the primitives required to draw one line on the Widget. Note the number of GCValues that have to be set to achieve this purpose. The XDrawLine function definition is as follows:

```

XDrawLine( Display *dpy,

           Drawable d,

```

```

GC gc,

int x1,

int y1,

int x2,

int y2);

```

It's more efficient to draw multiple lines in one call. Use the `XDrawLines` function with a pointer to an array of points and its size.

The mode parameter can be set to

- `CoordModeOrigin`, to use the values relative to the drawable's origin.
- `CoordModePrevious`, to use the values as deltas from the previous point. A delta is the change in x and y coordinates between this coordinate and the previous one.

The first point is always relative to the drawable's origin. To draw boxes, use the `XDrawRectangle` function:

```

XDrawRectangle( Display *display, Drawable dwindow,

                GC gc, int x, int y,

                unsigned int width, unsigned int height);

```

will draw a rectangle at (x,y) of geometry (width, height). To draw more than one box at a time, use the `XDrawRectangles()` function. This function is declared as

```

XDrawRectangles( Display *display, Window dwindow,

                 GC gc, XRectangle *xp, int number);

```


where `xp` is a pointer to an array of number rectangle definition structures.

For filled rectangles, use the `XFillRectangle` and `XFillRectangles` calls, respectively.

Drawing a Point

To draw a point on-screen, use the `XDrawPoint` or `XDrawPoints` function call. This is similar to line-drawing functions. (Refer to Listing 34.16.)

Drawing an Arc

To draw circles, arcs, and similar shapes, use the `XDrawArc` function:

```
XDrawArc(Display *display, Window dwindow,

        GC gc, int x, int y,

        unsigned int width, unsigned int height,

        int a1, int a2);
```

This function is very flexible. It draws an arc from an angle `a1` starting from the 3 o'clock position to angle `a2`. The units for angles are in one sixty-fourths ($1/64$) of a degree. The arc is drawn counterclockwise. The largest value is 64×360 units because the angle arguments are truncated. The width and height define the bounding rectangle for the arc. The `XDrawArcs()` function is used to draw multiple arcs, given pointers to the array. The prototype for this function is

```
XDrawArcs (Display *display, Window dwindow,

        GC gc, XArc *arcptr, int number);
```

To draw polygons, use the call

```
XDrawSegments( Display *display, Window dwindow, GC gc,
```

```
XSegment *segments, int number);
```

The `XSegment` structure includes four short members--`x1`, `y1`, `x2`, and `y2`--which define the starting and ending points of all segments. For connected lines, use the `XDrawLines` function shown earlier. For filled polygons, use the `XFillPolygon()` function call.

Using Fonts and FontLists

Fonts may be the trickiest aspect of Motif to master. See the section on fonts in Chapter 23, "Using Motif," before reading this section to familiarize yourself with font definitions.

The function `XLoadQueryFont(Display *dp, char *name)` returns an `XFontStruct` structure. This structure defines the extents for the character set. This is used to set the values of the `Font` field in a `GC`.

To draw a string on the screen, use

```
XDrawString ( Display *dp, Drawable dw, GC gc,

              int x, int y, char *str, int len);
```

which uses only the foreground color. To draw with the background and foreground colors, use

```
XDrawImageString ( Display *dp, Drawable dw, GC gc,

                  int x, int y, char *str, int len);
```

The X Color Model

The X Color Model is based on an array of colors called a colormap. Applications refer to a color by its index into this colormap. The indices are placed in the application's frame buffer, which contains an entry for each pixel of the display. The number of bits in the index defines the number of bitplanes. The number of bitplanes defines the number of colors that can be displayed on-screen at one time. For example, one bit per pixel gives two colors, four bits per pixel gives 16 colors, and eight bits per pixel gives 256 colors.

Applications generally inherit the colormap of their parent. They can also create their own colormap using the `XCreateColormap` call. The call is defined as

```
Colormap XCreateColormap( Display *display, Window   dwindow,
```

```
Visual *vp, int requested);
```

This allocates the number of requested color entries in the colormap for a window. Generally, the visual parameter is derived from the macro

```
DefaultVisual (Display *display, int screenNumber);
```

where `screenNumber = 0` in almost all cases. (See the previous chapter on Screens, Displays, and Windows for a definition of screens.) Colormaps are a valuable resource in X and must be freed after use. This is done via the call `XFreeColormap(Display *display, Colormap c);`.

Applications can get the standard colormap from the X server via the `XGetStandardColormap()` call, and set it via the `XSetStandardColormap()` call. These are defined as

```
XGetStandardColormap( Display *display, Window dwindow,

                      XStandardColormap *c, Atom property);
```

and

```
XSetStandardColormap( Display *display, Window dwindow,

                      XStandardColormap *c, Atom property);
```

Once applications have a Colormap to work with, they have to follow two steps:

1. Define the colormap entries.

The property atom can take the values of `RGB_BEST_MAP`, `RGB_GRAY_MAP`, or `2RGB_DEFAULT_MAP`. These are names of colormaps stored in the server. They are not colormaps in themselves.

2. Set the colormap for a window via the call

```
XSetWindowColormap ( Display *display, Window dwindow, Colormap c );
```

For setting or allocating a color in the Colormap, use the `XColor` structure defined in `<X/Xlib.h>`.

To see a bright blue color, use the segment

```
XColor color;

color.red = 0;

color.blue = 0xffff;

color.green = 0;
```

Then add the color to the Colormap using the call to the function:

```
XAllocColor(Display *display,

             Window dwindow,

             XColor *color );
```

See Listing 34.17 for a sample function to set the color of a Widget.

Listing 34.17. Convenience function for getting colors.

```
/**

*** Convenience function to get colors

**/

Pixel GetPixel( Widget w, int r, int g, int b)

{
```

```
Display *dpy;

int      scr;

Colormap cmp;

XColor      clr;


dpy = XtDisplay(w);

scr = DefaultScreen(dpy);

cmp = DefaultColormap(dpy);


clr.red = (short)r;

clr.green = (short)g;

clr.blue = (short)b;

clr.flags = DoRed | DoGreen | DoBlue;


/**
```

```

*** Note that the default black pixel of the display and screen

*** is returned if the color could not be allocated.

**/

return(XAllocColor(dpy,cmp,&clr) ? clr.pixel : BlackPixel(dpy,scr));

}

```

The default white and black pixels are defined as

```

Pixel BlackPixel( Display *dpy, int screen);

Pixel WhitePixel( Display *dpy, int screen);

```

and will work with any screen as a fallback.

The index (Pixel) returned by this function is not guaranteed to be the same every time the application runs. This is because the colormap could be shared between applications requesting colors in different orders. Each entry is allocated on a next-available-entry basis. Sometimes, if you overwrite an existing entry in a cell, you might actually see a change in a completely different application. So be careful.

Applications can query the RGB components of a color by calling the function

```

XQueryColor( Display *display, Colormap *cmp, Xcolor *clr);

```

For many colors at one time, use

```

XQueryColors( Display *display, Colormap *cmp,

              Xcolor *clr,          int number);

```

At this time the application can modify the RGB components. Then you can store them in the colormap with the call

```
XStoreColor( Display *display, Colormap *cmp, XColor *clr);
```

Recall that X11 has some strange names for colors in the `/usr/lib/rgb.txt` file. Applications can get the RGB components of these names with a call to

```
XLookupColor( Display *display, Colormap cmp,
              char *name, XColor *clr, XColor *exact);
```

The name is the string to search for in the `rgb.txt` file. The returned value `clr` contains the next closest existing entry in the colormap. The exact color entry contains the exact RGB definition in the entry in `rgb.txt`. This function does not allocate the color in the colormap. To do that, use the call

```
XAllocNamedColor( Display *display, Colormap cmp,
                  char *name, XColor *clr, XColor *exact);
```

Pixmaps, Bitmaps, and Images

A Pixmap is like a window, but is off-screen and therefore invisible to the user. This is usually the same depth of the screen. You create a Pixmap with the call

```
XCreatePixmap (Display *dp,
               Drawable dw,
               unsigned int width,
               unsigned int height,
```

```
unsigned int depth);
```

A drawable can be either a Window (on-screen) or a Pixmap (off-screen). Bitmaps are Pixmap of a depth of one pixel. Look in

```
/usr/include/X11/bitmaps
```

for a listing of some of the standard bitmaps.

The way to copy Pixmap from memory to the screen is via the call to XCopyArea. The prototype for this call is

```
XCopyArea( Display dp,

Drawable Src, Drawable Dst,

GC gc, int src_x, int src_y,

unsigned int width, unsigned int height,

int dst_x, int dst_y);
```

The caveat with this XCopyArea is that the depth of the Src and Dst drawables have to be of the same depth. To show a bitmap on a screen with depth greater than 1 pixel, you have to copy the bitmap one plane at a time. This is done via the call

```
XCopyPlane( Display dp,

Drawable Src, Drawable Dst,

GC gc, int src_x, int src_y,

unsigned int width, unsigned int height,
```



```
int dst_x, int dst_y, unsigned long plane);
```

where the plane specifies the bit plane to which this one-bit-deep bitmap must be copied. The actual operation is largely dependent on the modes set in the GC.

For example, to show the files in the `/usr/include/bitmaps` directory that have three defined values for a sample file called `gumby.h`:

- `gumby_bits` = pointer to an array of character bits.
- `gumby_height` and `gumby_width` = integer height and width.

First, create the bitmap from the data using the `XCreateBitmapFromData()` call. To display this one-plane-thick image, copy the image from this plane to plane 1 of the display. You can actually copy to any plane in the window. A sample call could be set for copying from your Pixmap to the Widget's plane 1 in the following manner:

```
XCOPYPlane( XtDisplay(w), yourPixmap, XtWindow(w), gc,
            0,0, your_height, your_width, 0,0,1);
```

where it copies from the origin of the Pixmap to the origin of plane 1 of the window.

There are other functions for working with images in X. These include the capability to store device-dependent images on disk and the Xpm format.

Summary

This chapter covered the following topics:

- The basics of writing Motif applications
- The special naming conventions in Motif and X
- Writing and compiling your first Motif application
- An overview of the Motif Widget hierarchy
- Working with various common Widgets
- Designing layouts

- Creating pop-up menus and menu bars
- Creating simple dialog boxes
- How to use the mouse in event handling
- How to use colors in X
- How to draw lines and points

This chapter could easily expand into a book. (Please, do not tempt me!) I have only covered the basics of writing Motif applications. However, given the vast number of tools in Linux, you can see how you can port any existing Motif application code to and from a Linux machine. Similarly, a Linux machine can also prove to be a good development platform for developing Motif applications.

- [- 35 -](#)
 - [XView Programming](#)
 - [A Note About CDE](#)
 - [Overview](#)
 - [Requirements](#)
 - [Listing](#)
 - [35.1. Sample makefile for creating XView applications.](#)
 - [Header Files](#)
 - [NOTE](#)
 - [Sample Application](#)
 - [Listing](#)
 - [35.2. A simple application.](#)
 - [NOTE](#)
 - [Initialization](#)
 - [Creating Objects](#)
 - [Exiting an Application](#)
 - [Frames](#)
 - [Listing](#)
 - [35.3. Header and footer frames.](#)
 - [Command Frames](#)
 - [Listing](#)
 - [35.4. Using command frames.](#)
 - [NOTE](#)
 - [TIP](#)
 - [Setting Colors on Frames](#)
 - [Listing](#)
 - [35.5. Using CMS.](#)
 - [TIP](#)
 - [CAUTION](#)
 - [Canvases](#)
 - [Listing](#)
 - [35.6. Using canvases and scrollbars.](#)
 - [TIP](#)
 - [Buttons](#)
 - [Listing](#)
 - [35.7. Using menus, buttons, and choices.](#)
 - [NOTE](#)
 - [List Items](#)
 - [Listing](#)
 - [35.8. Using lists to display data.](#)
 - [Scale Bars](#)
 - [Listing](#)
 - [35.9. Using slider control.](#)

- [TIP](#)
- [Text Windows](#)
- [Listing](#)
- [35.10. Using text items.](#)
- [Where to Go from Here](#)
- [Summary](#)

- 35 -

XView Programming

by Kamran Husain

IN THIS CHAPTER

- A Note About CDE
- Overview
- Requirements
- Frames
- Command Frames
- Setting Color on Frames
- Canvases
- Buttons
- List Items
- Scale Bars
- Text Windows
- Where to Go from Here

In this chapter you will learn how to program in an older, but still widely found, OPEN LOOK-based user interface manager called XView. You will find this distribution helpful when you work with older code or

when you port code from the OPEN LOOK style to Motif.

A Note About CDE

In March 1993, the Common Open Software Environment (COSE) committees adopted the Common Desktop Environment (CDE). CDE is based on the Motif interface. Sun Microsystems Inc., the primary developer of OPEN LOOK applications, agreed to conform to CDE as well. In short, this means that OPEN LOOK interface-based applications will soon be out of style. However, applications with an OPEN LOOK interface still exist and have to be ported to Motif eventually. A good knowledge of how OPEN LOOK applications work will be very beneficial to you if you ever have to port old existing code to conform to CDE.

Overview

To a programmer, the XView toolkit is an object-oriented toolkit. Think of XView objects as building blocks from which the user can create complicated applications, and think of each block as part of a package. Each package provides properties that you can modify to configure the object.

The XView toolkit consists of the objects shown in Figure 35.1. The subclasses are derived from the classes to their left. For example, `Icon` is a subclass of `Window`. Each class is also referred to as a package.

Some objects are visible and some are not. The visible objects provide the windows, scrollbars, and so on. The invisible objects, such as the font, display, or server, provide frameworks that aid in the display or layout of visible objects.

When you create an object, you get a handle to the object back from the XView libraries. Handles are opaque pointers to structures. This means that you can pass information via functions to these objects via their handles but you cannot see their structures directly.

The following functions enable you to manipulate all XView objects:

- `xv_init()` Establishes the connection to the server, initializes the notifier (message handler), and loads the resource databases
- `xv_create()` Creates an object
- `xv_destroy()` Destroys an object
- `xv_find()` Finds an object with given criteria; if not found, creates the object
- `xv_get()` Gets an attribute value
- `xv_set()` Sets an attribute value

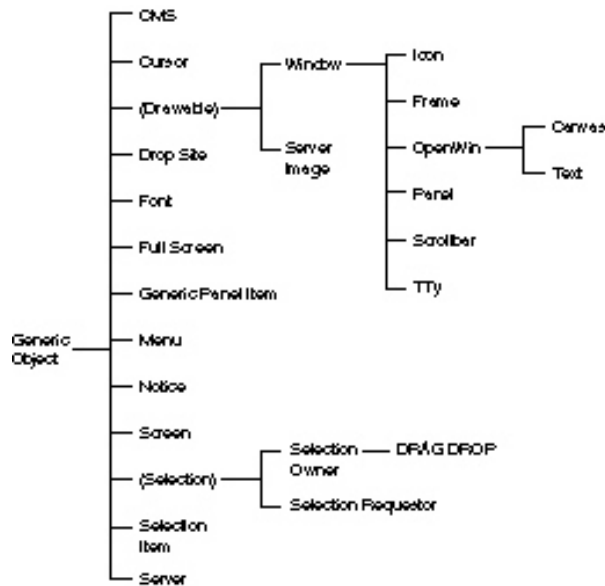


Figure 35.1. *XView class hierarchy.*

There are three categories of attributes: generic attributes apply to all objects; common attributes are shared by some, but not all, objects; and specific attributes belong to one class of objects only. Attributes that are specific to a type of object are prefixed with the name of the object; for example, FRAME_*, ICON_*, MENU_*, and so on. Common and generic attributes are prefixed by XV_. For example, XV_HEIGHT applies to all objects, but FRAME_HEIGHT applies only to frame objects.

Each attribute may have different types of values. For example, the following code sets a panel_item.

```

panel_item = (Panel_item) xv_create( masterpanel, PANEL_CYCLE,

XV_HEIGHT,      100,

XV_WIDTH,       50,

PANEL_LABEL_X,   100,

PANEL_LABEL_Y,   100,

PANEL_LABEL_STRING, "Help",

```

```

    PANEL_CHOICE_STRINGS, "About ... ",

    "How to use Help",

    "Search Index",

    NULL,

    NULL ) ;

```

Note how the types of values are mixed in this function call. All the attributes except `PANEL_CHOICE_STRINGS` take a single argument. The `PANEL_CHOICE_STRINGS` attribute takes a list of arguments. The list is terminated with a `NULL` value.

We will go over the details of each object in this chapter.

Requirements

To create an XView program you must link in the XView and OPENLOOK graphics library, which include all the toolkit functions for you. You will also need the X11 library. The command line to use the gcc compiler for a simple XView application is

```
$ gcc sample.c -lxview -lolgx -lX11 -o sample
```

This compile command relies on the fact that your libraries are located in `/usr/lib` or you have links to this location. The libraries are located in the `/usr/openwin` directories.

See the sample makefile in Listing 35.1 that you can use with the applications in this chapter. Note that this is not a fully functional makefile; you will have to modify it for your applications. The excerpt shown in Listing 35.1 is for the `LIST35_1.c` sample file in this chapter.

Listing 35.1. Sample makefile for creating XView applications.

```
CC= gcc
```

```
LIBPATH=/usr/openwin/lib
```

```
INCPATH=/usr/openwin/include
```

```
LIBS= -lxview -lolgx -lX11
```

```
LIST33_1: LIST33_1.c
```

```
$(CC) $< -I$(INCPATH) -L$(LIBPATH) $(LIBS) -o $@
```

```
LIST33_2: LIST33_2.c
```

```
$(CC) $< -I$(INCPATH) -L$(LIBPATH) $(LIBS) -o $@
```

```
LIST33_3: LIST33_3.c
```

```
$(CC) $< -I$(INCPATH) -L$(LIBPATH) $(LIBS) -o $@
```



```
LIST33_4: LIST33_4.c
```

```
$(CC) $< -I$(INCPATH) -L$(LIBPATH) $(LIBS) -o $@
```

The `-lxview` in `LIBS` refers to the `libxview.a` library. The `libxview.a` library contains the code for all the windows manipulation, and `libolgx.a` contains the OPENLOOK graphics library. The `libX11.a` is required by the `libxview.a` library, and `libolgx.a` is required by the `libxview.a` library.

Header Files

The basic definitions you must use for XView are located in two files: `xview/generic.h` and `xview/xview.h` in the `/usr/openwin/include` directory tree. The header files required by other packages, such as `FONT` or `FRAME`, are declared in files of the same name as the package. For example, for the `FONT` package you must use the `xview/font.h` header file. You can include these files more than once.

NOTE: In some source distributions, the file `generic.h` is not explicitly called out in the source files. In order to compile source files under Linux, you will need the `generic.h` file.

Sample Application

Take a look at the simple application shown in Listing 35.2, which places a window with a Quit button on it.

Listing 35.2. A simple application.

```
/*

** A sample program to show you how to present items for

** selection to the user.
```

```
* *
```

```
*/
```

```
#include <xview/generic.h>
```

```
#include <xview/xview.h>
```

```
#include <xview/frame.h>
```

```
#include <xview/panel.h>
```

```
#include <xview/cms.h>
```

```
Frame frame;
```

```
#define FORE 0
```

```
#define BACK 2
```

```
int main(int argc, char *argv[])
```

```
{

Cms    cms;

Panel panel;

void quit();


xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);


cms = (Cms ) xv_create((int)NULL,CMS, /* NULL -> use the default Frame*/

    CMS_SIZE, CMS_CONTROL_COLORS + 4,

    CMS_CONTROL_CMS, True,

    CMS_NAMED_COLORS, "LightBlue", "Blue", "Red", "Green", NULL,

    NULL );
```

```
frame = (Frame)xv_create((int)NULL, FRAME,  
  
    FRAME_LABEL, argv[1],  
  
    XV_WIDTH, 200,  
  
    XV_HEIGHT, 100,  
  
    NULL);  
  
xv_set(frame,  
  
    WIN_CMS, cms,  
  
    WIN_FOREGROUND_COLOR, CMS_CONTROL_COLORS + FORE,  
  
    WIN_BACKGROUND_COLOR, CMS_CONTROL_COLORS + BACK,  
  
    NULL);  
  
panel = (Panel)xv_create(frame, PANEL, NULL);
```

```
(void) xv_create(panel, PANEL_BUTTON,  
  
    PANEL_LABEL_STRING, "Quit",  
  
    PANEL_NOTIFY_PROC, quit,  
  
    NULL);
```

```
xv_main_loop(frame);
```

```
exit(0);
```

```
}
```

```
void quit()
```

```
{
```

```
xv_destroy_safe(frame);
```

```
}
```

NOTE: At the risk of being too literal, don't forget to run the applications in this chapter from an X terminal. The programs will not run without the X server. Additionally, you should run these programs from the OPENLOOK window manager (`olvwm`) to ensure proper operation and look-and-feel of these XView implementations

The output from this application is shown in Figure 35.2. There are several things that you should note about this sample application.

- The XV toolkit is initialized as soon as possible in the application with the `xv_init` call.
- All attribute values to the `xv_create()` function call are terminated with a `NULL` parameter.
- The `(Frame)` cast is used to override the default returned value from `xv_create()`.
- The `<xview/generic.h>` header file is used to get all the required definitions for the file.

Initialization

You should initialize the XView system as soon as possible in any application. The `xv_init()` call does this for you. By default, `xv_init()` uses the `DISPLAY` environment variable for you. By passing the `argc` and `argv` values you can override the default values for the application from the command line. You can use `xv_init()` only once in an application; the libraries ignore all other calls. Normally you'd override the `DISPLAY` variable if you wanted to display the window on a different machine.

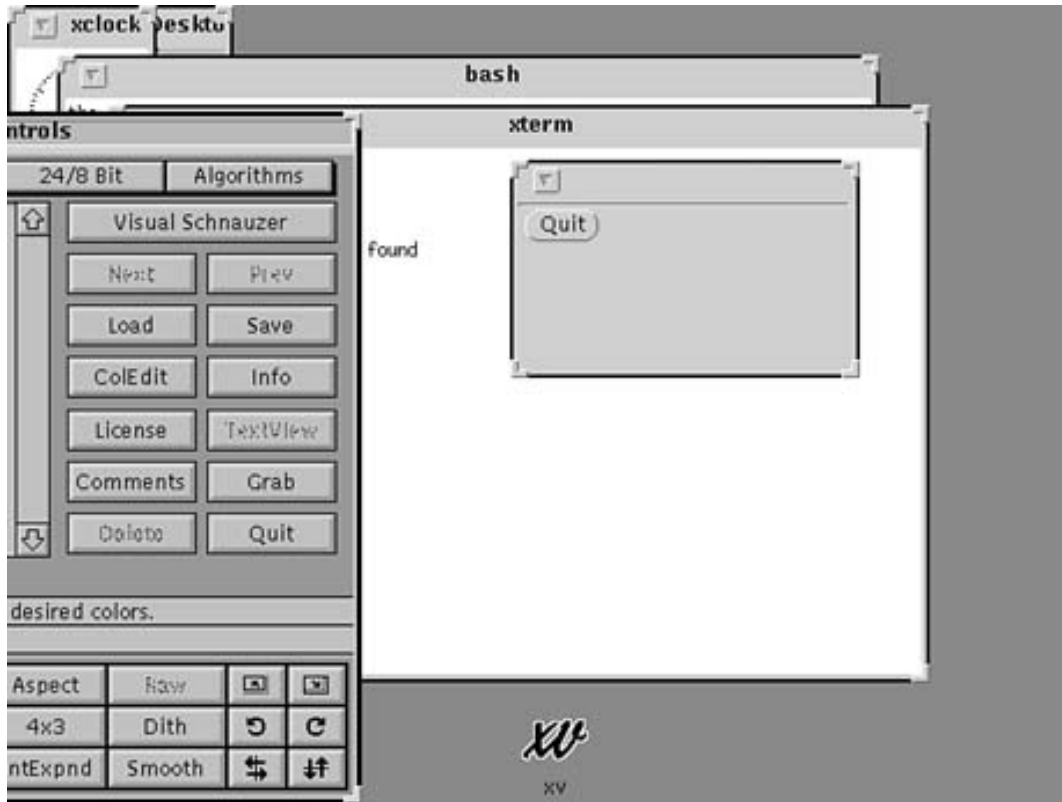


Figure 35.2. A sample XView application.

You can use two types of parameters for the first argument to `xv_init()`: `XV_INIT_ARGS`, which leaves the `argc` and `argv` unchanged, or `XV_INIT_ARGC_PTR_ARGV`, which modifies `argc` and `argv` to remove all XView-specific arguments. With `XV_INIT_ARGS`, you pass `argc` into `xv_init` and with `XV_INIT_ARGC_PTR_ARGV` you pass the address of `argc` to `xv_init()`.

Creating Objects

The `xv_create` function is used to create all the objects in an application. The syntax for the `xv_create` function is

```
xv_object xv_create(xv_object owner, xv_package pkg, void *attr)
```

where the `owner` is the parent of the object being created, and of type `pkg` given the attributes listed in variable length arguments starting with `attr`. Sometimes you can use a `NULL` value in place of the `owner` parameter to indicate that the `owner` value can be substituted for screen or server as appropriate. However, in some calls the `owner` parameter must point to a valid object, so the `NULL` value will generate an error.

The attributes for the newly created object inherit their behavior from their parents. The attributes can be overridden by values included in the variable list specified in `attr`.

The values of attributes are set in the following decreasing order of precedence:

- A call to `xv_set` will override any other type of setting
- Any command-line arguments
- Values in the `.Xdefaults` file
- Values in the attributes of an `xv_create` call
- Window Manager defaults

Exiting an Application

The best way to get out of an XView application is to destroy the topmost object. Use the `xv_destroy_safe()` function call, which waits for the destruction of all derived objects and cleans up after itself. You can also use `xv_destroy()` to get out immediately with the risk of not giving up system resources but be able to exit very quickly. If you don't give up resources, they will not be freed for use by any other applications in the system and will use up valuable memory space.

Frames

A frame is a container for other windows. A frame manages the geometry of subwindows that do not overlap. Some examples include canvases, text windows, panels, and scrollbars. You saw a base frame in the output of `LIST35_2.c` (refer to Figure 35.2 and Listing 35.2).

Frames enable you to specify three types of outputs on three areas. The topmost area is the name on the top of the frame called the header. The bottom of the frame is divided into two sections; one is left-justified and the other is right-justified. Figure 35.3 shows the output from Listing 35.3, which shows you how to write to these areas.

Listing 35.3. Header and footer frames.

```
/*

**

** Listing to show headers and footers.

**
```



```
* /
```

```
#include <xview/generic.h>
```

```
#include <xview/xview.h>
```

```
#include <xview/frame.h>
```

```
#include <xview/panel.h>
```

```
/*
```

```
**
```

```
* /
```

```
Frame frame;
```

```
/*
```

```
**
```

```
*/
```

```
int main(int argc, char *argv[])
```

```
{
```

```
Panel panel;
```

```
void quit();
```

```
xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);
```

```
frame = (Frame)xv_create((int)NULL, FRAME,
```

```
    FRAME_LABEL, "Title Here",
```

```
    FRAME_SHOW_FOOTER, TRUE,
```

```
    FRAME_LEFT_FOOTER, "left side",
```

```
    FRAME_RIGHT_FOOTER, "right side",
```

```
        XV_WIDTH, 200,

        XV_HEIGHT, 100,

        NULL);

panel = (Panel)xv_create(frame, PANEL,NULL);


(void) xv_create(panel, PANEL_BUTTON,

        PANEL_LABEL_STRING, "Quit",

        PANEL_NOTIFY_PROC, quit,

        NULL);


xv_main_loop(frame);

exit(0);

}
```

```

void quit()

{

xv_destroy_safe(frame);

}

```

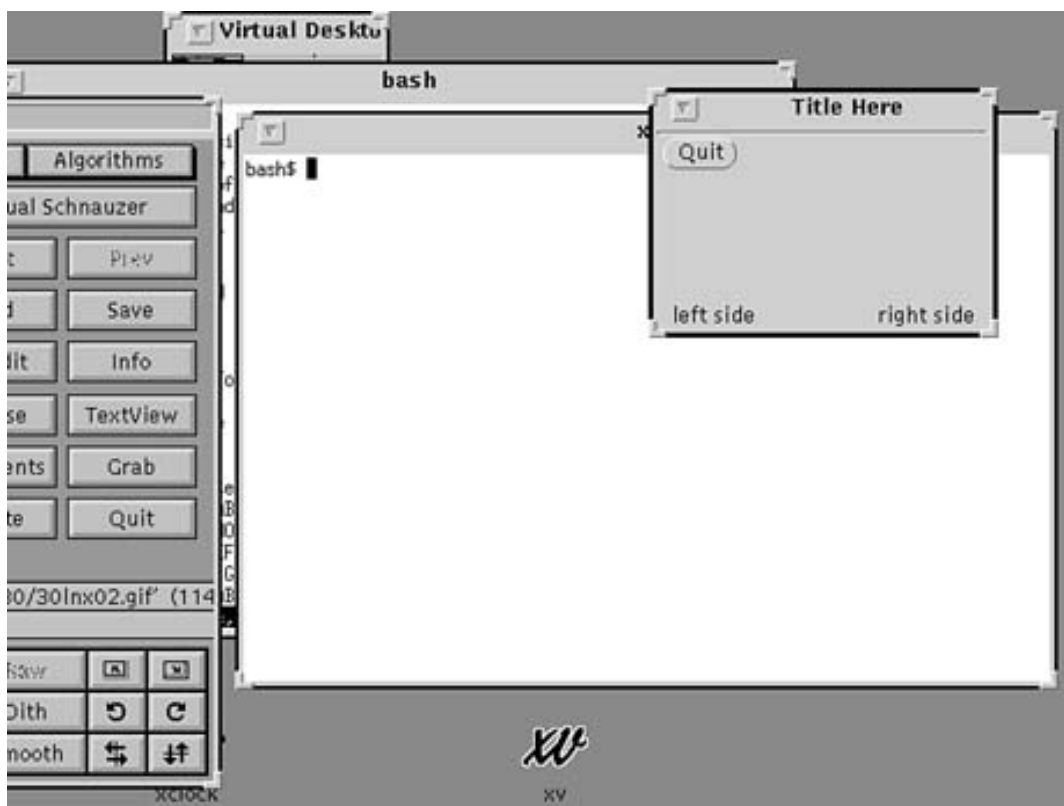


Figure 35.3. Header and footer frames.

The parameters used to create these footers are shown in the following lines:

```
Frame frame;
```

```

frame = (Frame) xv_create((int)NULL, FRAME,

    FRAME_LABEL, "Title Here",

    FRAME_SHOW_FOOTER, TRUE,

    FRAME_LEFT_FOOTER, "left side",

    FRAME_RIGHT_FOOTER, "right side",

    XV_WIDTH, 200,

    XV_HEIGHT, 100,

    NULL);

```

You have to turn the footer display on with the `FRAME_SHOW_FOOTER` attribute set to `TRUE`. The other values in this call actually set the values of the header and footer.

Command Frames

Command frames are usually used to perform a quick function and then disappear. These frames are usually pop-up frames like the pushpin dialog boxes you saw in Chapter 24, "OPEN LOOK and OpenWindows." If the pushpin is pressed in, the dialog box remains "pinned" to the screen; otherwise, the dialog box will go away after the user performs the section.

Listing 35.4 shows you a program to create command frames.

Listing 35.4. Using command frames.

```
/*
```

```
** Sample Application to show command frames.
```

```
**
```

```
*/
```

```
#include <xview/generic.h>
```

```
#include <xview/xview.h>
```

```
#include <xview/frame.h>
```

```
#include <xview/panel.h>
```

```
/*
```

```
** Global Frames
```

```
*/
```

```
Frame frame;
```

```
Frame popup;
```

```
/*  
  
**  
  
** Declare the used functions here  
  
**  
  
*/  
  
void show_greeting(Frame *fp);  
  
int  show_popup();  
  
int  push_it();  
  
void quit();  
  
/*  
  
** The main function
```

```
*/
```

```
int main(int argc, char *argv[])
```

```
{
```

```
Panel panel;
```

```
Panel fpanel;
```

```
/*
```

```
**  Initialize the toolkit
```

```
*/
```

```
xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);
```

```
/*
```

```
**  Create top level frame.
```



```
* /

frame = (Frame)xv_create((int)NULL, FRAME,

FRAME_LABEL, "Title Here",

FRAME_SHOW_FOOTER, TRUE,

FRAME_LEFT_FOOTER, "Popup",

FRAME_RIGHT_FOOTER, argv[1],

XV_WIDTH, 200,

XV_HEIGHT, 100,

NULL);

/*

** Create the popup Frame.

*/

popup = (Frame) xv_create(frame, FRAME_CMD,
```

```
FRAME_LABEL, "Popup",

XV_WIDTH, 100,

XV_HEIGHT, 100,

NULL);

/*

** Create panel for popup

*/

fpanel = (Panel)xv_get(popup, FRAME_CMD_PANEL,NULL);

/*

** Add buttons to popup

*/
```

```
(void) xv_create(fpanel, PANEL_BUTTON,

PANEL_LABEL_STRING, "Greet",

PANEL_NOTIFY_PROC,  show_greeting,

NULL);
```

```
(void) xv_create(fpanel, PANEL_BUTTON,  
  
PANEL_LABEL_STRING, "Push Me",  
  
PANEL_NOTIFY_PROC,  push_it,  
  
NULL);
```

/*

```
** Create panel
```

$$* /$$

```
panel = (Panel)xv_create(frame, PANEL,NULL);
```

```
/*

** Add buttons to main application frame

*/

(void) xv_create(panel, PANEL_BUTTON,

PANEL_LABEL_STRING, "Hello",

PANEL_NOTIFY_PROC, show_popup,

NULL);


(void) xv_create(panel, PANEL_BUTTON,

PANEL_LABEL_STRING, "Quit",

PANEL_NOTIFY_PROC, quit,

NULL);
```

```
        xv_main_loop(frame);

exit(0);

}

void quit()

{

xv_destroy_safe(frame);

}


void show_greeting(Frame *fp)

{

printf ("Greet you? How?\n");
```

```
}
```

```
show_popup(Frame item, Event *ev)
```

```
{
```

```
xv_set(popup, XV_SHOW, TRUE, NULL);
```

```
}
```

```
push_it(Panel_item item, Event *ev)
```

```
{
```

```
int ret;
```

```
ret = (int)xv_get(popup, FRAME_CMD_PIN_STATE) ;
```

```
if (ret == FRAME_CMD_PIN_IN)

{

    printf("Pin already in.. bye\n");

    xv_set(popup, XV_SHOW, TRUE, NULL); /* refresh anyway */

}

else

{

    printf("Pin out.. pushing it in\n");

    xv_set(popup, FRAME_CMD_PIN_STATE, FRAME_CMD_PIN_IN, NULL);

    xv_set(popup, XV_SHOW, TRUE, NULL); /* refresh anyway */

}

}
```

The output from Listing 35.4 is shown in Figure 35.4.

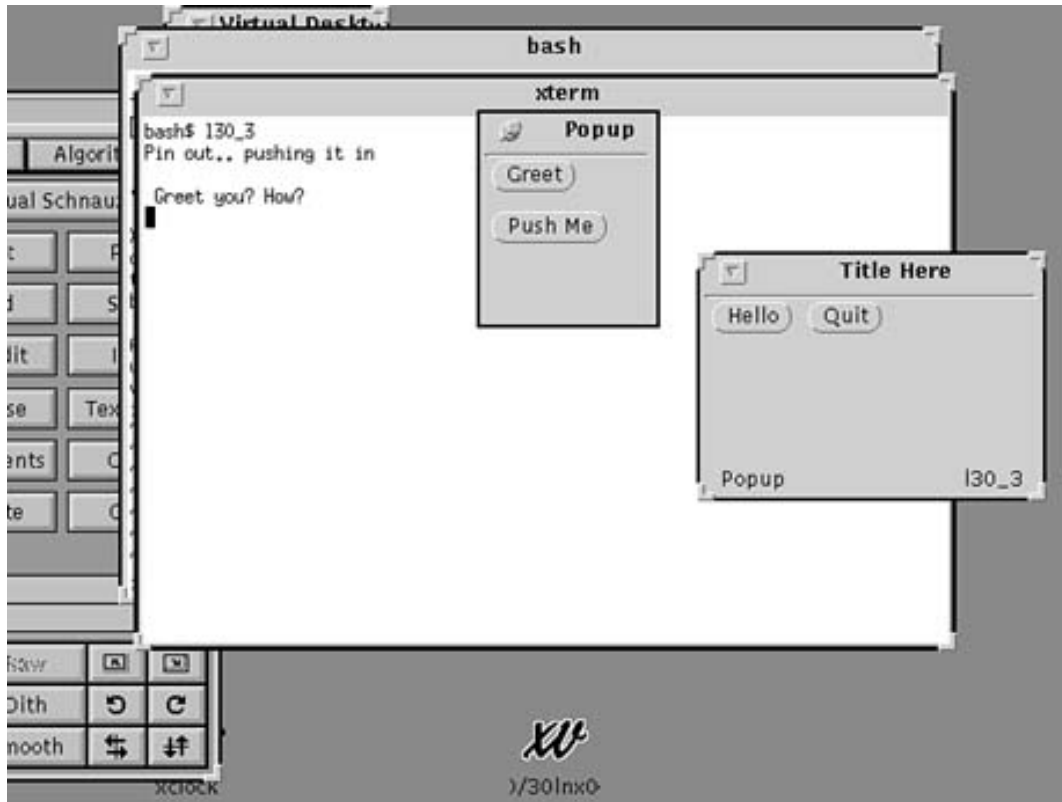


Figure 35.4. *Using command frames.*

Look at the important lines of the program in Listing 35.4 in detail. By examining these lines you will learn the following:

- How to create pop-up menus
- How to add buttons to a panel
- How to handle callback functions for xview objects

There are two frames in this application: `frame` and `popup`. These frames are declared at the top of the application with the statements

```
Frame frame;
```

```
Frame popup;
```

We also declared the following functions in this application:

- `void show_greeting(Frame *fp)`: This function is called when the Greeting button is

pressed.

- `int show_popup()`: This function is called when the Hello button is pressed.
- `int push_it()`: This function is called when the Push Me button is pressed.
- `void quit()`: This function is called when the Quit button is pressed.

The main `xv_init()` and frame creation for the program is as in Listing 35.3. Let's concentrate on the pop-up menu examples.

First, the pop-up frame is created with the following lines:

```
popup = (Frame) xv_create(frame, FRAME_CMD,
FRAME_LABEL, "Popup",
XV_WIDTH, 100,
XV_HEIGHT, 100,
NULL);
```

This call will create the pop-up frame with `frame` as the owner. The pop-up frame is not displayed immediately. You can create several pop-up frames this way and display them only when they are needed.

NOTE: Note that if you do not set the `XV_WIDTH` and `XV_HEIGHT` parameters for this `xv_create()` call, the pop-up screen will occupy the entire screen.

Next we create a panel for this pop-up with the call

```
fpanel = (Panel)xv_get(popup, FRAME_CMD_PANEL, NULL);
```

Then we add the Greet and Push Me buttons to this new `fpanel`. This is done by the `xv_create` calls, which are shown next:

```
(void) xv_create(fpanel, PANEL_BUTTON,  
  
PANEL_LABEL_STRING, "Greet",  
  
PANEL_NOTIFY_PROC,  show_greeting,  
  
NULL);
```

```
(void) xv_create(fpanel, PANEL_BUTTON,  
  
PANEL_LABEL_STRING, "Push Me",  
  
PANEL_NOTIFY_PROC,  push_it,  
  
NULL);
```

At this point you are ready to create the main application frame, show it, and go into the main loop. The important call that does this is shown next. The function `show_popup()` is assigned to be called when the Hello button is pressed.

```
(void) xv_create(panel, PANEL_BUTTON,  
  
PANEL_LABEL_STRING, "Hello",  
  
PANEL_NOTIFY_PROC,  show_popup,
```

```
NULL);
```

Now look at the functions that are called when each button is pressed. The `show_greeting()` function simply prints out a string. (You can use your imagination here for the contents of the string for your own application.) The `show_popup()` function will use the call to the `xv_set()` function to actually make the pop-up frame visible.

```
xv_set(popup, XV_SHOW, TRUE, NULL);
```

Now for the function that will emulate the behavior of pushing in the pin. This is the `push_it()` function. The `FRAME_CMD_PIN_STATE` parameter requests the state of the pushpin on the dialog box. The state for the pin is defined as `FRAME_CMD_PIN_IN` if the pushpin is already pushed in. This is the state for which you check. If the pushpin is not in this state, it is pushed in with the `xv_set(popup, FRAME_CMD_PIN_STATE, FRAME_CMD_PIN_IN, NULL);` function call.

TIP: A command frame has no resize corners by default. To turn these corners on, set `FRAME_SHOW_RESIZE_CORNERS` to `TRUE`.

Setting Colors on Frames

The colors on an XView frame object are defaulted to the `OpenWindows.WindowColor` resource. This resource is inherited by all subframes as well. You can override these colors with the CMS package. The CMS package is created by a call to `xv_create()`:

```
cms = (Cms *) xv_create(parent, CMS, attrs, NULL);
```

A CMS can contain as many colors as are allowed in the largest color map you can create. You can have several color maps referencing the same color; in fact, the system can share the location of colors between two independent applications. For this reason, you should allocate all your colors once, at CMS creation, to allocate all the colors in your color map to prevent another application from changing the colors you absolutely need.

For example, to create a CMS with four named colors, you would use the following function call:

```

cms = (Cms *)xv_create(parent, CMS,

    CMS_SIZE, 4,

    CMS_NAMED_COLORS, "Violet", "Yellow", "Blue", "Orange",

    NULL);

```

The CMS_SIZE value asks for a four-entry color table that is indexed from 0 to 3, with the values of the named colors "Violet", "Yellow", "Blue", and "Orange". The foreground color for a frame is the first indexed color in a CMS, and the background color for a frame is the last indexed (n-1) color in a CMS. Setting a CMS_SIZE will give you either an error or a monochromatic display. Of course, to avoid runtime errors you must know that the colors you just specified by name do exist in the `/usr/lib/rgb.txt` file.

Listing 35.5 is an example of an application that sets the colors. This will let you set the foreground and background colors of a frame and all its children.

Listing 35.5. Using CMS.

```

#include <xview/generic.h>

#include <xview/xview.h>

#include <xview/frame.h>

#include <xview/panel.h>

#include <xview/cms.h>

```

```
Frame frame;
```

```
#define FORE 3
```

```
#define BACK 0
```

```
int main(int argc, char *argv[])
```

```
{
```

```
    Cms      cms;
```

```
    Panel panel;
```

```
void quit();
```

```
xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);
```

```
cms = (Cms ) xv_create((int)NULL,CMS, /* NULL -> use the default Frame*/

CMS_SIZE, CMS_CONTROL_COLORS + 4,

CMS_CONTROL_CMS, True,

CMS_NAMED_COLORS, "LightBlue", "Blue", "Red", "Green", NULL,

NULL);


frame = (Frame)xv_create((int)NULL, FRAME,

FRAME_LABEL, argv[1],

XV_WIDTH, 200,

XV_HEIGHT, 100,

NULL);


xv_set(frame,

WIN_CMS, cms,
```

```
WIN_FOREGROUND_COLOR, CMS_CONTROL_COLORS + FORE,

WIN_BACKGROUND_COLOR, CMS_CONTROL_COLORS + BACK,

NULL);

panel = (Panel)xv_create(frame, PANEL,NULL);

(void) xv_create(panel, PANEL_BUTTON,

PANEL_LABEL_STRING, "Quit",

PANEL_NOTIFY_PROC, quit,

NULL);

xv_main_loop(frame);

exit(0);
```

{

}

CAUTION: Use `xv_set()` to override the colors on a frame. Any color requests on a frame at the time of creation are overridden by values of the `.Xdefaults` resources values.

- The Paint window, which contains the actual painted data
- The View window, which has the scrollbars but no painted data
- The canvas subwindow, which contains the union of the View window and Paint window

Look at a simple example in Listing 35.6 of how to use scrollbars and how to paint on a paint window. (I have added line numbers for readability.)

Listing 35.6. Using canvases and scrollbars.

```
1      /*  
  
2      ** An example of a scrolled window  
  
3      */  
  
4      #include <X11/Xlib.h>  
  
5      #include <xview/generic.h>  
  
6      #include <xview/xview.h>  
  
7      #include <xview/frame.h>  
  
8      #include <xview/panel.h>  
  
9      #include <xview/canvas.h>  
  
10     #include <xview/scrollbar.h>  
  
11     #include <xview/xv_xrect.h>
```

```
12      /*

13      ** Declare our callback functions for this application.

14      */

15      Frame frame;

16      void redraw(Canvas c, Xv_Window pw, Display *dp, Window xwin,

17                  Xv_xrectlist *rp) ;

18      int main(int argc, char *argv[])

19      {

20          Canvas canvas;

21          Panel panel;

22          Scrollbar h_s, v_s;

23          void quit();
```

```
24      xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

25      frame = (Frame)xv_create((int)NULL, FRAME,

26          FRAME_LABEL, argv[1],

27          XV_WIDTH, 400,

28          XV_HEIGHT, 200,

29          (int)NULL);

30      /*

31      ** Create the canvas.

32      */

33      canvas = (Canvas) xv_create(frame, CANVAS,

34          CANVAS_REPAINT_PROC, redraw,
```

```
35         CANVAS_X_PAINT_WINDOW, TRUE,

36         CANVAS_AUTO_SHRINK, FALSE,

37         CANVAS_AUTO_EXPAND, TRUE,

38         CANVAS_WIDTH, 500,

39         CANVAS_HEIGHT, 500,

40         XV_WIDTH, 400,

41         XV_HEIGHT, 200,

42         NULL);

43     /*

44     ** Create the splittable scrollbars

45     */

46     h_s = (Scrollbar)xv_create(canvas, SCROLLBAR,

47         SCROLLBAR_DIRECTION, SCROLLBAR_HORIZONTAL,
```

```
48         SCROLLBAR_SPLITTABLE, TRUE,

49         NULL);

50     v_s = (Scrollbar)xv_create(canvas, SCROLLBAR,

51         SCROLLBAR_DIRECTION, SCROLLBAR_VERTICAL,

52         SCROLLBAR_SPLITTABLE, TRUE,

53         NULL);

54     xv_main_loop(frame);

55     exit(0);

56 }

57 void redraw(Canvas c, Xv_Window pw, Display *dp, Window xwin,
```

```
58         Xv_xrectlist *rp)

59     {

60         GC gc;

61         int wd, ht;

62         int i;

63         int j;

64         int dx;

65         int dy;


66         gc = DefaultGC(dp, DefaultScreen(dp));

67         wd = (int)xv_get(pw, XV_WIDTH);

68         ht = (int)xv_get(pw, XV_HEIGHT);


69         dy = ht / 10;
```

```
70         for (j = 0; j < ht; j += dy)

71             XDrawLine(dp, xwin, gc, j,0,j,ht);

72

73         dx = wd / 10;

74         for (i = 0; i < wd; i += dx)

75             XDrawLine(dp,xwin,gc, 0,i,wd,i);

76     }

77     void quit()

78     {

79         xv_destroy_safe(frame);

80     }
```

Lines 33 through 42 create the canvas. The `CANVAS_AUTO_EXPAND` and `CANVAS_AUTO_SHRINK` parameters maintain the relation of the canvas subwindow and the paint subwindow. These values default to `TRUE`. When both values are `TRUE`, the canvas and paint subwindows will expand or shrink based on the size of the window on which they are being displayed.

Setting the `CANVAS_AUTO_EXPAND` value to `TRUE` enables the paint subwindow to expand larger than the canvas subwindow. If the canvas subwindow expands to a bigger size than the paint subwindow, the paint subwindow is expanded to at least that size as well. If the canvas subwindow size shrinks, the paint subwindow does not shrink because it is already at the same size or bigger than canvas subwindows at that time.

Setting the `CANVAS_AUTO_SHRINK` value to `TRUE` forces the canvas object to always confirm that the paint subwindow's height and width are never greater than the canvas subwindow. In other words, the size of the paint subwindow is always changed to be at least the same or less than the size of the canvas subwindow.

You can explicitly set the size of the canvas window with the `CANVAS_WIDTH` and `CANVAS_HEIGHT` parameters. (See lines 38 and 39.) These canvas dimensions can be greater than the viewing window dimensions set with `XV_WIDTH` and `XV_HEIGHT` (lines 40 and 41).

We have to add the include file `<xview/scrollbar.h>` to get the definitions for the `scrollbar` package. These are created in lines 46 through 53. Note how we have to create two separate scrollbars, one vertical and one horizontal.

The scrollbars in this example show how they can split to provide multiple, tiled views of the data in the canvas window. To split a view, press the right mouse button on a scrollbar and you will be presented with a pop-up menu. Choose the Split View option to split the view or the Join View option to join two views together. You will not see a Join View option if a scrollbar does not dissect a view.

You can programmatically split a canvas view even if scrollbars are not present. Use the `OPENWIN_SPLIT` attribute in an `xv_set ()` function call. For example:

```
xv_window xv;

xv = (xv_window)xv_get(canvas, OPENWIN_NTH_VIEW, 0);

xv_set(canvas,

OPENWIN_SPLIT,

OPENWIN_SPLIT_VIEW, xv,
```



```

OPENWIN_SPLIT_DIRECTION,

OPENWIN_SPLIT_HORIZONTAL,

NULL) ;

```

TIP: You may want to group your `xv_set ()` function calls into distinct logical calls to set each type of parameter instead of one long convoluted list of parameters to one `xv_set ()` function. Splitting the code into these groups makes the code easier to read and debug.

Note that only `OPENWIN_*` types of attributes are allowed in the `xv_set ()` call with the `OPENWIN_SPLIT` parameter. Do not mix other types of attributes. To get the first view you can use a value of 0 to the `OPENWIN_NTH_VIEW` parameter. For the next view, use 1, and so on. To get an idea of how many views there are for this canvas use the call

```

int number;

number = (int)xv_get(canvas, OPENWIN_NVIEWS);

```

To get the paint window to do your own drawing, perhaps in response to other user input, you can use the `xv_get ()` function to get the paint window. For example:

```

xv_window xv_paint;

xv_paint = (xv_window)xv_get(canvas, CANVAS_VIEW_PAINT, null);

```

Listing 35.6 shows how to use the standard `Xlib` function calls to draw on the canvas. (See Figure 35.5.) You must use the include file `<X/Xlib.h>` for all the definitions and declarations. The `XDrawLine` function used in this example is somewhat simple. However, this example shows you how to set up your Graphics Context and use the standard `XDrawLine` function to draw a grid. You can use other X drawing

functions just as easily.

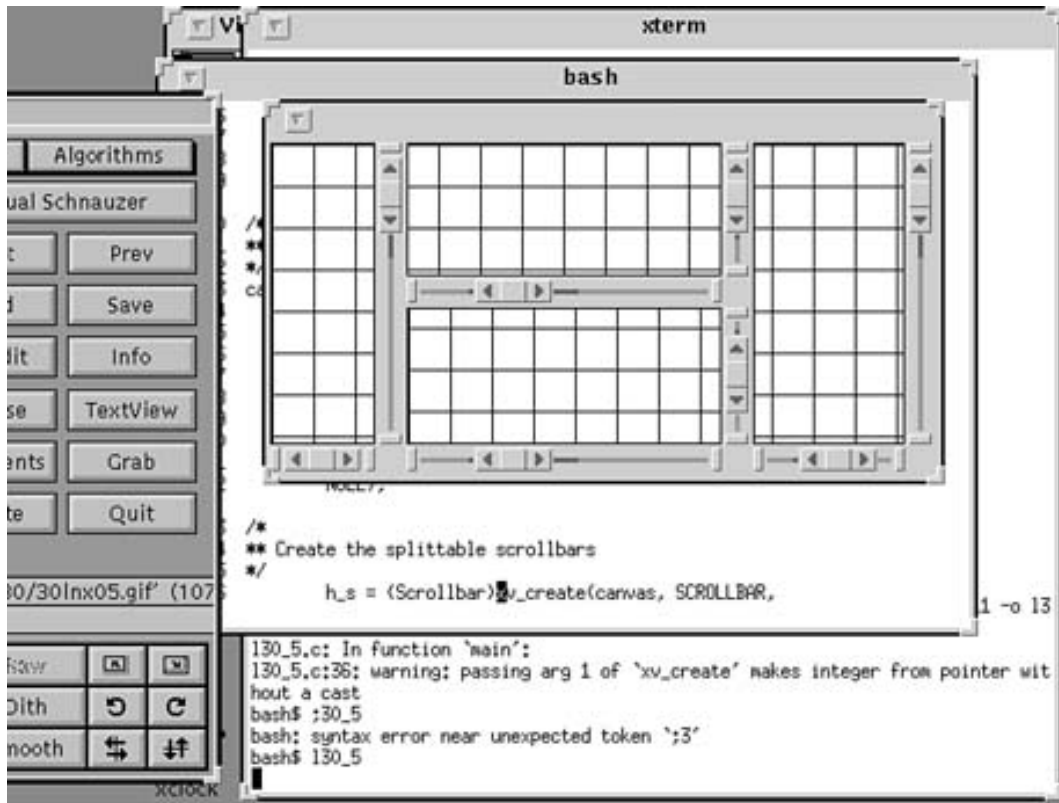


Figure 35.5. The scrolled window example.

Buttons

A button item enables a user to select an action. Several types of buttons are available to you as a programmer. Figure 35.6 shows how various buttons are used.

Four distinct examples are shown in Figure 35.6:

- The Menu Item is shown as "Y/N/Q."
- The 1 of N choice items from four items.
- The M of N choice of items from three items to match others.
- Choosing via four checkboxes.

The listing for generating Figure 35.6 is shown in Listing 35.7. We will go over this listing in detail.



Figure 35.6. *Using buttons.*

Listing 35.7. Using menus, buttons, and choices.

```
/*

** A sampler of some of the choices to present to a user

*/

#include <xview/generic.h>

#include <xview/xview.h>

#include <xview/frame.h>
```

```
#include <xview/panel.h>
```

```
#include <xview/openmenu.h>
```

```
Frame frame;
```

```
int menuHandler( Menu item, Menu_item selection);
```

```
int selected( Panel_item item, Event *ev);
```

```
void quit();
```

```
int main(int argc, char *argv[])
```

```
{
```

```
Rect  *rt;
```

```
Rect  *qrt;
```

```
Panel panel;
```

```
Panel quitbtn;
```

```
Panel oneN;
```

```
Panel manyN;
```

```
Panel chooser;
```

```
Menu menu1;
```

```
xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);
```

```
frame = (Frame)xv_create((int)NULL, FRAME,
```

```
FRAME_LABEL, argv[1],
```

```
XV_WIDTH, 400,
```

```
XV_HEIGHT, 200,
```

```
NULL);
```

```
panel = (Panel)xv_create(frame, PANEL,NULL);

quitbtn = (Panel)xv_create(panel, PANEL_BUTTON,

PANEL_LABEL_STRING, "Quit",

PANEL_NOTIFY_PROC, quit,

XV_X, 20,

NULL);

menu1 = (Menu) xv_create((int)NULL, MENU,

MENU_STRINGS, "Yes", "No", "Maybe", "Bye", NULL,

MENU_NOTIFY_PROC, menuHandler,

NULL);
```

```
xv_create (panel, PANEL_BUTTON,  
  
          PANEL_LABEL_STRING, "Y/N/Q",  
  
          PANEL_ITEM_MENU, menu1,  
  
          PANEL_NOTIFY_PROC, selected,  
  
          NULL);  
  
  
qrt = (Rect *) xv_get(quitbtn, XV_RECT);  
  
  
oneN = (Panel) xv_create(panel, PANEL_CHOICE,  
  
          PANEL_LABEL_STRING, "1 of N",  
  
          PANEL_CHOICE_STRINGS,  
  
          "extra", "large", "medium", "small", NULL,
```

```
    XV_X, 20,

    XV_Y, rect_bottom(qrt) + 20,

    NULL);

rt = (Rect *) xv_get(oneN, XV_RECT);

manyN = (Panel) xv_create(panel, PANEL_CHOICE,

    PANEL_LABEL_STRING, "M of N",

    PANEL_CHOICE_STRINGS,

        "tomato", "celery", "carrot" , NULL,

    PANEL_CHOOSE_ONE, FALSE,

    XV_X, 20,

    XV_Y, rect_bottom(rt) + 20,

    NULL);
```



```
rt = (Rect *) xv_get(manyN, XV_RECT);

chooser = (Panel) xv_create(panel, PANEL_CHECK_BOX,

    PANEL_LAYOUT, PANEL_HORIZONTAL,

    PANEL_LABEL_STRING, "Extras",

    PANEL_CHOICE_STRINGS,

        "fries", "potato", "Q. potatoe", "salad" , NULL,

    PANEL_CHOOSE_ONE, FALSE, /* Let `em have it all */

    XV_X, 20,

    XV_Y, rect_bottom(rt) + 20,

    NULL);

xv_main_loop(frame);
```

```
exit(0);

}

/*

** This function is called when you select an item

*/

int selected( Panel_item item, Event *ev)

{

printf(" %s .. \n ", xv_get(item, PANEL_LABEL_STRING));

}

/*

** This function handles the menu selection item.

** Shows you how to exit via menu item too.
```

```
*/
```

```
int menuHandler(Menu item, Menu_item thing)
```

```
{
```

```
printf("%s .. \n", xv_get(thing, MENU_STRING));
```

```
if (!strcmp((char *)xv_get(thing, MENU_STRING), "Bye")) quit();
```

```
}
```

```
/*
```

```
** Make a clean exit.
```

```
*/
```

```
void quit()
```

```
{
```

```
xv_destroy_safe(frame);
```

```
}
```

Take a look at the part where the "Y/N/Q" menu button was created. First we created the menu items on the menu as menu1. Note that we did not display all of the choices in the menu, just its header.

```
menu1 = (Menu) xv_create(NULL, MENU,

    MENU_STRINGS, "Yes", "No", "Maybe", "Bye", NULL,

    MENU_NOTIFY_PROC, menuHandler,

    NULL);
```

Then we created the panel button that will house this menu with the following lines:

```
xv_create (panel, PANEL_BUTTON,

    PANEL_LABEL_STRING, "Y/N/Q",

    PANEL_ITEM_MENU, menu1,

    PANEL_NOTIFY_PROC, selected,

    NULL);
```

That was it. Now you can click the right button on the "Y/N/Q" button to get the selection items as a pull-down menu. If you click the left button, the first item in the menu item will be displayed momentarily and selected. Two functions are assigned as callbacks in the previous code segments:

- `menuHandler()`: This function will show on your terminal the menu item selected.
- `selected()`: This function merely displays the menu item string. You could just as easily display another menu or other items instead of this simple example.

Now look at the example for the "1 of N" selection. As the name of this item suggests, you can choose only one of a given number of items. This is called an exclusive selection.

The following lines are used to create this exclusive selection item:

```
oneN = (Panel) xv_create(panel, PANEL_CHOICE,

    PANEL_LABEL_STRING, "1 of N",

    PANEL_CHOICE_STRINGS,

        "extra", "large", "medium", "small", NULL,

    XV_X, 20,

    XV_Y, rect_bottom(qrt) + 20,

    NULL);
```

Note how we used the core class's `XV_X` and `XV_Y` attributes to position this box below the Quit button. We got the position as a rectangle (`typedef Rect`) of the Quit button via the `xv_get` call given the `XV_RECT` attribute:

```
qrt = (Rect *) xv_get(quitbtn, XV_RECT);
```

The position given by `XV_X` and `XV_Y` was relative to the top-left position of the panel. This given position is known as absolute positioning because we are using hard-coded numbers to position items.

NOTE: To position items generally we can use two functions: `xv_row()` and `xv_col()`. These functions use the `WIN_ROW_GAP` and `WIN_COLUMN_GAP` to set the spaces between the items. The following example shows you how to position twelve items on a panel:

```
#define ROW 3
#define COL 4
extern char *name[ROW][COL];
int i, j;
for (i = 0; i < ROW; i++)
for (j = 0; j < COL; j++)
{
    xv_create(panel, PANEL_BUTTON,
    XV_X, xv_col(panel,j),
    XV_Y, xv_row(panel,i),
    PANEL_LABEL_STRING, name[i][j],
    NULL);
}
```

All items presented in this list are shown with the NULL-terminated list passed in with the `PANEL_CHOICE_STRINGS` attribute. The default function of `PANEL_CHOICE` is to enable only one selection. To get more than one selection if you have a list of choices, you can follow the same procedure you used for the exclusive selection panel. The difference between 1 of M and M of N lies in setting the value of the `PANEL_CHOOSE_ONE` to `FALSE`. This usage creates the M of N items shown in the following lines:

```
manyN = (Panel) xv_create(panel, PANEL_CHOICE,

PANEL_LABEL_STRING, "M of N",

PANEL_CHOICE_STRINGS,

"tomato", "celery", "carrot" , NULL,
```

```

    PANEL_CHOOSE_ONE, FALSE,

    XV_X, 20,

    XV_Y, rect_bottom(rt) + 20,

    NULL);

```

With 1 of M, we use the `XV_RECT` call to position this choice of many item's button on the screen.

Finally, this example showed you how to use check boxes to create the input items shown for our (United States) ex-vice president's choices of a side order. Checkboxes are always non-exclusive. The text to do this is shown in the following lines:

```

chooser = (Panel) xv_create(panel, PANEL_CHECK_BOX,

    PANEL_LAYOUT, PANEL_HORIZONTAL,

    PANEL_LABEL_STRING, "Extras",

    PANEL_CHOICE_STRINGS,

        "fries", "potato", "Q. potatoe", "salad" , NULL,

    XV_X, 20,

    XV_Y, rect_bottom(rt) + 20,

    NULL);

```

This set of checkboxes was also positioned to align with the `qv_get` and `rect_bottom()` calls.

List Items

Use the `PANEL_LIST` attribute to show lists of items. An example is shown in Figure 35.7. The corresponding listing is shown in Listing 35.8. Lists enable you to insert text (and graphics as glyphs) in them. You can have duplicates in a list. If you do not want to allow duplicates, set the `PANEL_LIST_INSERT_DUPLICATE` to `FALSE`.

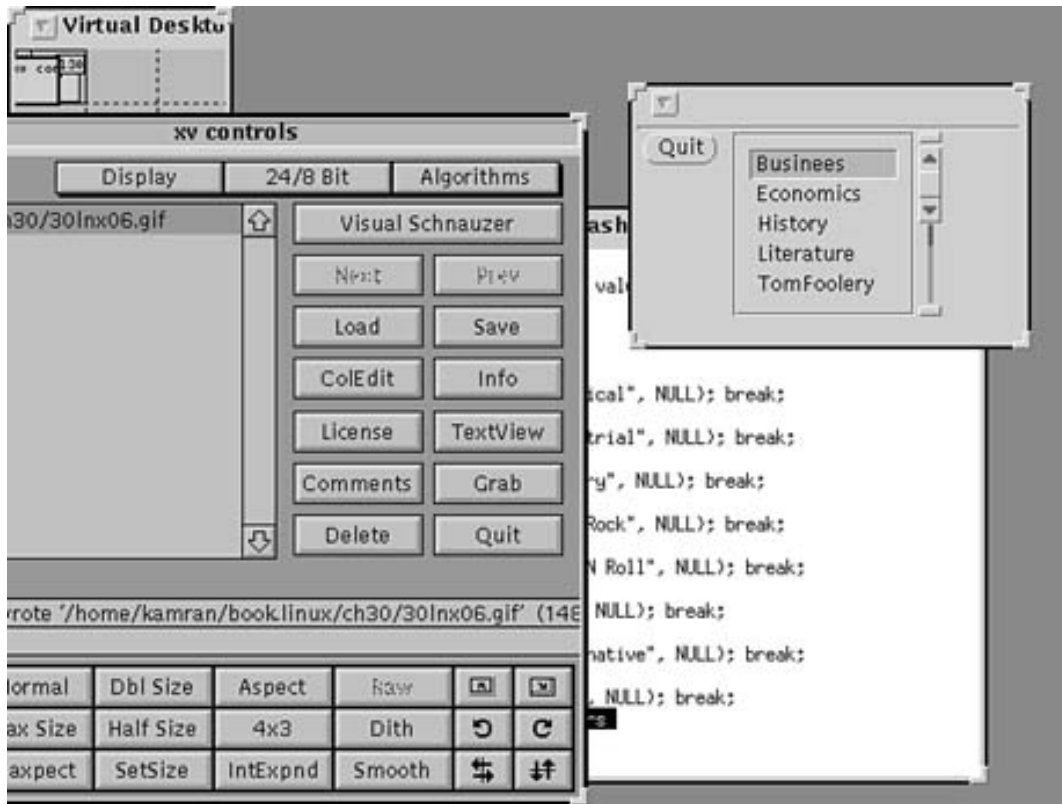


Figure 35.7. *Using lists to display data.*

Listing 35.8. Using lists to display data.

```
/*
```

```
** Using Lists
```

```
*/
```



```
#include <xview/generic.h>
```

```
#include <xview/xview.h>
```

```
#include <xview/frame.h>
```

```
#include <xview/panel.h>
```

```
Frame frame;
```

```
int main(int argc, char *argv[])
```

```
{
```

```
Panel panel;
```

```
void quit();
```

```
xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

frame = (Frame)xv_create((int)NULL, FRAME,

FRAME_LABEL, argv[1],

XV_WIDTH, 200,

XV_HEIGHT, 100,

NULL);

panel = (Panel)xv_create(frame, PANEL,NULL);

(void) xv_create(panel, PANEL_BUTTON,

PANEL_LABEL_STRING, "Quit",

PANEL_NOTIFY_PROC, quit,

NULL);
```

```
(void) xv_create(panel, PANEL_LIST,

    PANEL_LIST_STRINGS,

    "Business", "Economics", "History",

    "Literature", "TomFoolery", "Math",

    "Computer Sci.", "Engineering", NULL,

    NULL);

xv_main_loop(frame);

exit(0);

}
```

```

void quit()

{

xv_destroy_safe(frame);

}

```

Lists are ordered from 0 and up, so the first row is 0, the second row is 1, and so on. To delete the rows 7 through 9 from a long list, use the `xv_set` function:

```

xv_set(list_item,

        PANEL_LIST_DELETE_ROWS, 6, 3

        NULL);

```

In the preceding example you are requesting that 3 rows be deleted starting from row index number 6 (which is the seventh row). All other rows are adjusted upward after these rows are deleted.

To insert items into this list you can use `PANEL_LIST_INSERT` and `PANEL_LIST_STRING` calls. If you wanted to replace the third row with a string pointed to by a pointer called `buffer`, you would use the following function call:

```

xv_set(list_item,

        PANEL_LIST_DELETE, 2,

        PANEL_LIST_INSERT, 2,

```

```
PANEL_LIST_STRING, buffer,
```

```
NULL);
```

The `PANEL_NOTIFY_PROC` function for a list is called when an item is selected, deselected, added, or deleted. The prototype for this function call is

```
listCallback(

    Panel_item      item,

    char            *string,

    Xv_opaque       client_data,

    Panel_list_op    op,

    Event           *event,

    int             row);
```

The `item` is the panel list itself in this function call. The `string` is the label for the row, or `NULL` if no item is defined in the list for the row. The opaque `client_data` is a user-specified value specified at list creation time with the `PANEL_LIST_CLIENT_DATA` parameter. For example, the line

```
PANEL_LIST_CLIENT_DATA, 2, "Hello",
```

will assign the value of `client_data` to 2 for the row with the string "Hello" in it. Each `client_data` value must be assigned one line at a time.

The `op` parameter can be one of the following values:

- `PANEL_LIST_OP_SELECT` when the row is selected
- `PANEL_LIST_OP_DESELECT` when the row is deselected
- `PANEL_LIST_OP_VALIDATE` when a new row is added
- `PANEL_LIST_OP_DELETE` when the row has been deleted

You can take action based on the value of the `op` parameter in one handy function or have this function call other functions. For example, the following pseudocode illustrates how you could handle the `op` parameter:

```
switch (op)

{

    case    PANEL_LIST_OP_SELECT: selectHandler();

    break;

    case    PANEL_LIST_OP_DESELECT: unSelectHandler();

    break;

    case    PANEL_LIST_OP_VALIDATE: addRowHandler();

    break;

    case    PANEL_LIST_OP_DELETE: deleteRowHandler();
```

```
break;  
  
}
```

Scale Bars

Now look at how you create slider bars so the user can set the value of a variable. An example of this application is shown in Figure 35.8 and a corresponding listing is given in Listing 35.9.

Listing 35.9. Using slider control.

```
#include <xview/generic.h>  
  
#include <xview/xview.h>  
  
#include <xview/frame.h>  
  
#include <xview/panel.h>  
  
  
  
  
  
  
  
Frame frame;  
  
Panel_item stationName;  
  
  
  
  
  
  
  
void display_setting(Panel_item, int value, Event *ev);
```

```
int main(int argc, char *argv[])

{

Panel panel;

Panel_item slider;

void quit();


xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);


frame = (Frame)xv_create((int)NULL, FRAME,

FRAME_LABEL, argv[1],

XV_WIDTH, 400,

XV_HEIGHT, 100,

NULL);
```



```
panel = (Panel)xv_create(frame, PANEL,  
  
    PANEL_LAYOUT, PANEL_VERTICAL,  
  
    NULL);  
  
(void) xv_create(panel, PANEL_BUTTON,  
  
    PANEL_LABEL_STRING, "Quit",  
  
    PANEL_NOTIFY_PROC, quit,  
  
    NULL);  
  
slider = xv_create (panel, PANEL_SLIDER,  
  
    PANEL_LABEL_STRING, "Radio Station",  
  
    PANEL_MIN_VALUE, 88,  
  
    PANEL_MAX_VALUE, 108,  
  
    PANEL_NOTIFY_PROC, display_setting,
```

```
PANEL_VALUE, 99,

PANEL_NOTIFY_LEVEL, PANEL_ALL, /* not just at the end */

PANEL_SHOW_RANGE, TRUE,

PANEL_TICKS, 10,

PANEL_SLIDER_WIDTH, 100,

NULL);


stationName = xv_create(panel, PANEL_MESSAGE,

    PANEL_LABEL_STRING, "sample",

    NULL);


xv_main_loop(frame);

exit(0);
```

```
}
```

```
void quit()
```

```
{
```

```
xv_destroy_safe(frame);
```

```
}
```

```
/*
```

```
** This function is called when the slider value is changed.
```

```
*/
```

```
void display_setting(Panel_item item, int value, Event *ev)
```

```
{
```

```
switch (value)
```

```
{
```

```
case 89: xv_set(stationName,  
  
    PANEL_LABEL_STRING,"Classical", NULL); break;  
  
case 91: xv_set(stationName,  
  
    PANEL_LABEL_STRING,"Industrial", NULL); break;  
  
case 93: xv_set(stationName,  
  
    PANEL_LABEL_STRING,"Country", NULL); break;  
  
case 95: xv_set(stationName,  
  
    PANEL_LABEL_STRING,"Soft Rock", NULL); break;  
  
case 101: xv_set(stationName,  
  
    PANEL_LABEL_STRING,"Roll N Roll", NULL); break;  
  
case 104: xv_set(stationName,  
  
    PANEL_LABEL_STRING,"Pop", NULL); break;  
  
case 107: xv_set(stationName,  
  
    PANEL_LABEL_STRING,"Alternative", NULL); break;
```

```

default: xv_set(stationName,

PANEL_LABEL_STRING, "bzzz", NULL); break;    }

}

```

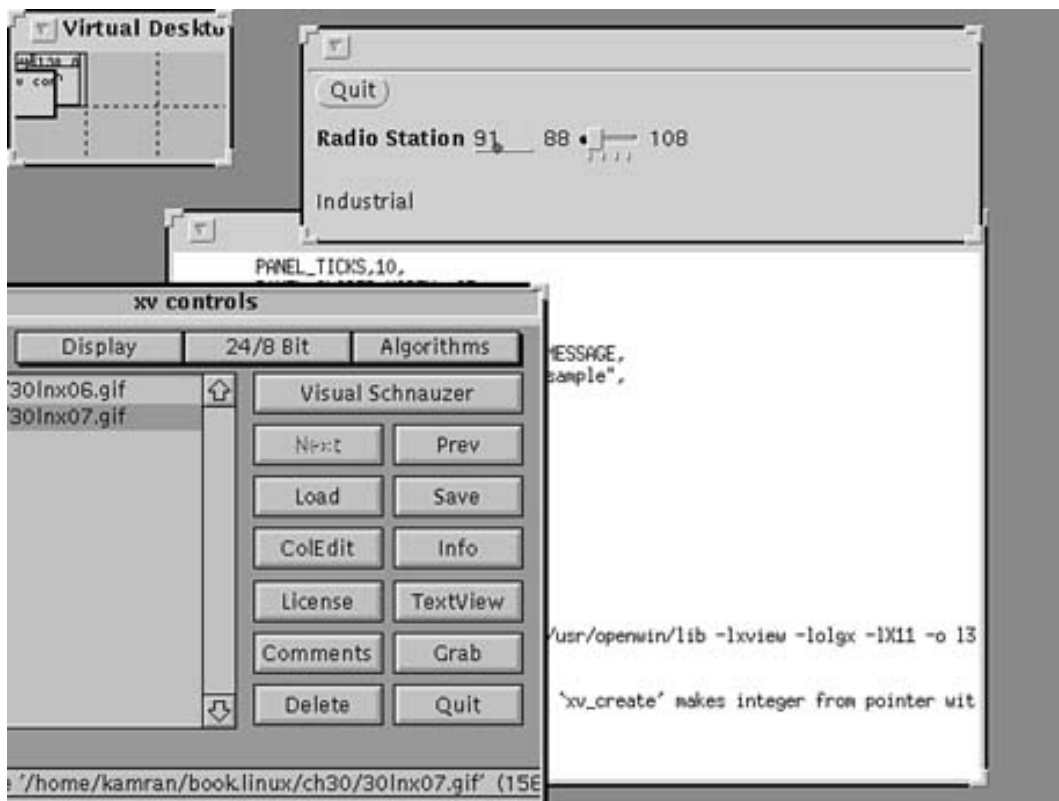


Figure 35.8. Using sliders.

To create a slider, assign the `PANEL_SLIDER` value to the `xv_create()` function call. How the slider is used and displayed is governed by the following attributes:

- `PANEL_MIN_VALUE` and `PANEL_MAX_VALUE`: The range of values that this slider can take. These values have to be integers. For the example in this book we used 88 and 108.
- `PANEL_SHOW_RANGE`: Sets the slider to show the value of the ranges allowed for the selection.
- `PANEL_NOTIFY_LEVEL`: Can be set to one of two values: `PANEL_ALL` if the callback procedure is called while the slider is moving, or `PANEL_DONE` only when the pointer button is released.
- `PANEL_DIRECTION`: Can be used to set the orientation of the slider to either horizontal or vertical.
- `PANEL_TICKS`: The number of ticks that show on the display. Set it to 0 if you do not want ticks to be shown. The number of ticks is adjusted as you size the slider. You fix the width of the slider by setting the `PANEL_SLIDER_WIDTH` to 100 (refer to Listing 35.8).

You can edit the selection value by clicking it and using the keyboard. This value will change the location of the slider as soon as you press the Enter key. Error values will be ignored.

Note how a message label displays the station name as the slider is being moved. To set the value of this label, make a call to `xv_set ()` and give the attribute `PANEL_LABEL_STRING` a string value. For example, if the value of the slider is 89, you can set the message to "Classical", as shown in the following lines:

```
case 89: xv_set(stationName,

PANEL_LABEL_STRING, "Classical", NULL); break;
```

TIP: You can create a gauge by using the `PANEL_GAUGE` package instead of `PANEL_SLIDER`. The dimensions of the gauge are set by the `PANEL_GAUGE_WIDTH` and `PANEL_GAUGE_HEIGHT` attributes. A user cannot change the value of the slider on a gauge because a gauge can be used only as a feedback item.

Text Windows

XView has a lot of options for displaying data. This section will cover only a few portions of this feature. Please refer to the man pages for Text in `/usr/openwin/man`. Let's get started with some of the basics, though. A sample application is shown in Listing 35.10, and its corresponding output is shown in Figure 35.9.

Listing 35.10. Using text items.

```
#include <xview/generic.h>
```

```
#include <xview/xview.h>
```

```
#include <xview/frame.h>
```

```
#include <xview/panel.h>
```

```
Frame frame;
```

```
int main(int argc, char *argv[])
```

```
{
```

```
Panel panel;
```

```
void quit();
```

```
    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);
```

```
frame = (Frame)xv_create((int)NULL, FRAME,  
  
    FRAME_LABEL, argv[1],  
  
    XV_WIDTH, 300,  
  
    XV_HEIGHT, 300,  
  
    NULL);  
  
panel = (Panel)xv_create(frame, PANEL,NULL);  
  
(void) xv_create(panel, PANEL_BUTTON,  
  
    PANEL_LABEL_STRING, "Quit",  
  
    PANEL_NOTIFY_PROC, quit,  
  
    NULL);
```



```
xv_create(panel, PANEL_TEXT,

        PANEL_LABEL_STRING, "Single",

        PANEL_VALUE, "Single Line of Text",

        NULL);

xv_create(panel, PANEL_MULTILINE_TEXT,

        PANEL_LABEL_STRING, "Multi",

        PANEL_DISPLAY_ROWS, 3,

        PANEL_VALUE_DISPLAY_LENGTH, 30,

        PANEL_VALUE, "Multiple Lines of Text \

in this example \

This is a line 1\

This is a line 2\

This is a line 3\
```

```
    of some long string",
```

```
    NULL);
```

```
xv_main_loop(frame);
```

```
exit(0);
```

```
}
```

```
void quit()
```

```
{
```

```
xv_destroy_safe(frame);
```

```
}
```

We created a single panel text entry item with the following lines by using the `PANEL_TEXT` package:

```
xv_create(panel, PANEL_TEXT,
```

```
PANEL_LABEL_STRING, "Single",

PANEL_VALUE, "Single Line of Text",

NULL) ;
```

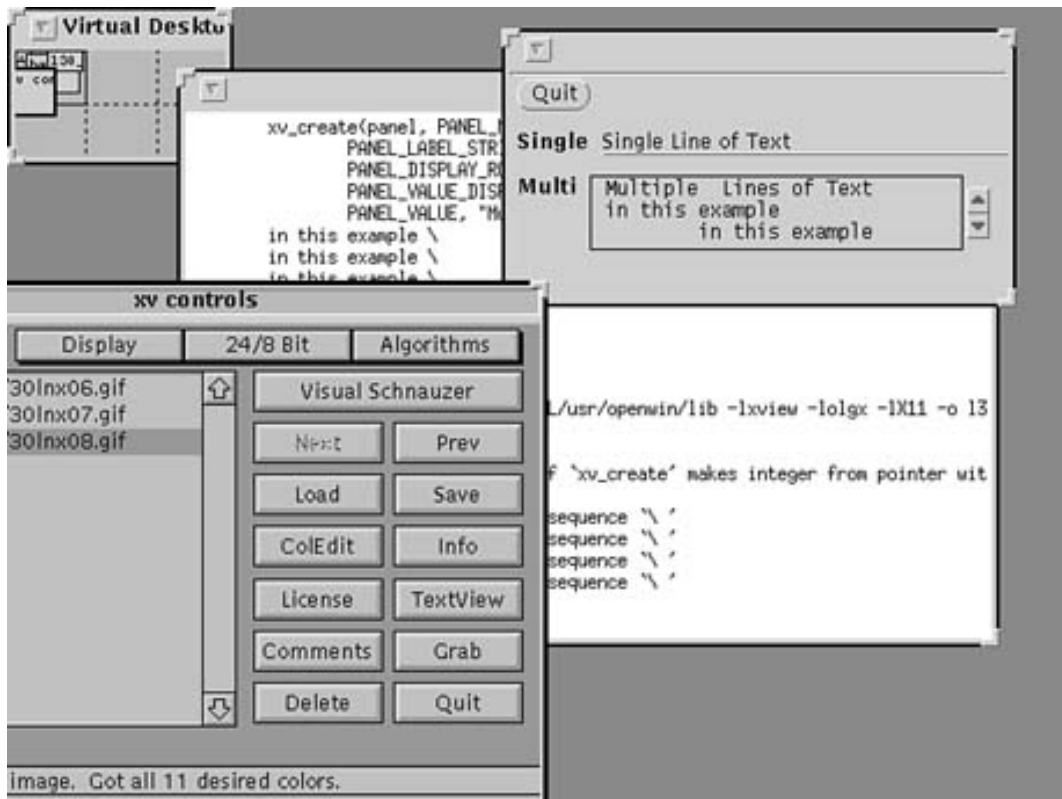


Figure 35.9. *Using text items.*

If the `PANEL_LAYOUT` value is set to `PANEL_VERTICAL`, the value is placed below the label. The default is `PANEL_HORIZONTAL`. The number of characters is set with `PANEL_VALUE_DISPLAY_LENGTH`. This value should not be less than 4. (This is not in the listing and is only for your information.)

If you want the user to enter private data such as password information, you can set the `PANEL_MASK_CHAR` value to something like an asterisk. This setting displays an asterisk for each character that the user types in. The value of the text remains what the user typed in.

You can have notification procedures for four types of input for a text item with the `PANEL_NOTIFY_LEVEL`. (See Table 35.1.)

Table 35.1. Notification procedures.

<i>Notification</i>	<i>Action to take on input</i>
PANEL_NONE	Never inform this package.
PANEL_NON_PRINTABLE	On each non-printable character.
PANEL_SPECIFIED	If the input character is found in a string specified by the attribute PANEL_NOTIFY_STRING.
PANEL_ALL	On all characters input.

You can also have multiple lines of text on a display. A sample of this is shown in Listing 35.10. Look at the following excerpted lines:

```
xv_create(panel, PANEL_MULTILINE_TEXT,

        PANEL_LABEL_STRING, "Multi",

        PANEL_DISPLAY_ROWS, 3,

        PANEL_VALUE_DISPLAY_LENGTH, 30,

        PANEL_VALUE, "Multiple  Lines of Text \

in this example\

This is a line 1\

This is a line 2\

This is a line 3\

of some long string",
```

```
NULL) ;
```

The `PANEL_MULTILINE_TEXT` package can have the following attributes set for it:\n
`PANEL_DISPLAY_ROWS` sets the number of lines that the viewing window will display

`PANEL_VALUE_DISPLAY_LENGTH` is the number of characters wide you want the display to be

Where to Go from Here

This chapter is a very brief introduction to the XView packages available under Linux. In this section you have learned a little about putting user interface items together on a panel. You should now have enough knowledge to start creating your own interfaces in XView. There are several other locations for getting more information about XView under Linux.

The following are XView packages for Linux:

- `xv32exmp`. Perhaps the most important example for the newbie. This package contains the examples for XView that demonstrate the Slingshot and UIT extensions, which are libraries that make it much easier to program a user interface under X.
- `xv32_a`. This package contains static libraries for developing XView applications (version 3.2). This is required if compiling XView apps with the `-g` or `-static` flags for debugging.
- `xv32_sa` and `xv32_so`. These are shared libraries for your compiled programs.
- `xvinc`. The include files you use when you are compiling XView programs.
- `xv0132`. Configuration files, programs, and other documentation for the OPEN LOOK Window Manager, `olwm`.
- `xvmenus`. Menus and help files for `olwm`.

Some cool binaries to look for in the `/usr/openwin/bin` directory are `workman`, which enables you to play music CDs on your CD player; `props`, for setting window parameters; and `perfmeter` for performance metering.

Look in the `/usr/openwin/man` directory for all the man pages for the XView package. The `/usr/openwin/include` file contains valuable information about some of the structures used by XView.

Summary

You use objects to build XView applications. Each object is a class and is referred to as a package. Each package has attributes that can have values. Attributes can be shared among other objects, be common to a few objects only, or be specific to one object.

You can retrieve an attribute's values by calling `xv_get ()` and set a value by calling `xv_set`. An attribute may be assigned more than one value. Each attribute can have a different type of value attached to it.

You can use standard `Xlib` function calls to perform drawing operations. This gives you tremendous flexibility in rendering your custom graphics on screens and XView packages.

The XView packages enable you to create and place objects on panels. You can place these objects using absolute positioning from the upper-left corner of a panel, relative to other objects, or in row/column order.

The `xv_create ()` call passes the type of object as a parameter to create XView objects. You can set other attributes by passing a `NULL`-terminated list to `xv_create ()`. Default attribute values that are not explicitly set by `xv_create ()` are inherited from the object's parent.

- [- 36 -](#)
 - [SmallTalk/X](#)
 - [What Is SmallTalk/X?](#)
 - [How to Install SmallTalk/X](#)
 - [NOTE](#)
 - [Invoking SmallTalk/X](#)
 - [Getting Around in ST/X](#)
 - [The Browsers Option](#)
 - [The System Browser](#)
 - [NOTE](#)
 - [The Class Hierarchy Browser](#)
 - [Implementors](#)
 - [Senders](#)
 - [The Changes Browser](#)
 - [NOTE](#)
 - [Directory Browser](#)
 - [The Workspace Option](#)
 - [The File Browser Option](#)
 - [The Projects Option](#)
 - [The Utilities Option](#)
 - [The Goodies Option](#)
 - [The Games & Demos Option](#)
 - [Editing in Browsers](#)
 - [Using the Inspector](#)
 - [Using the Debugger](#)
 - [Summary](#)

- 36 -

SmallTalk/X

by Rick McMullin

IN THIS CHAPTER

- What Is SmallTalk/X?
- How to Install SmallTalk/X
- Invoking SmallTalk/X
- Getting Around in ST/X
- The Browsers Option
- The Workspace Option
- The File Browser Option

- The Projects Option
- The Utilities Option
- The Goodies Option
- The Games & Demos Option
- Editing in Browsers
- Using the Inspector
- n Using the Debugger

This chapter describes the SmallTalk/X (ST/X), a fairly complete implementation of the SmallTalk-80 programming environment. Anyone who has used SmallTalk-80 or any other version of SmallTalk will be impressed with this freely available implementation. In this chapter we will see

- What SmallTalk/X is
- How to install SmallTalk/X
- How to invoke SmallTalk/X
- How to get around in SmallTalk/X

This chapter gives you an overview of the SmallTalk/X application. After reading the chapter you should be familiar with the facilities that SmallTalk/X provides and be able to navigate your way through the SmallTalk/X user interface.

What Is SmallTalk/X?

When describing SmallTalk/X, it is probably appropriate to start with a description of SmallTalk itself. SmallTalk is an object-oriented programming language that has been a continuing development project at ParcPlace Systems since the early 1970s. Although it was not the first object-oriented language, it was the first object-oriented language to gain wide use in the industry.

SmallTalk has been around for about 15 years now but it was not until recently that it started to become popular. Many universities now teach a SmallTalk course as part of their standard computer science curriculum, and many companies have seen the value that SmallTalk adds in terms of quick development.

SmallTalk/X was developed by Claus Gittinger and was first released in 1988. It is almost identical to the SmallTalk 80 implementation of the SmallTalk language. SmallTalk/X comes complete with an application launcher, several different browsers for browsing through the SmallTalk class hierarchy, and a very powerful debugging utility. The unique aspect of SmallTalk/X is that it can also behave as a SmallTalk-to-C translation utility. This is a very useful feature because this means that you will be able to combine the speed of development that SmallTalk provides with the speed of execution that C provides.

How to Install SmallTalk/X

Before installing SmallTalk/X, you must first retrieve it from `sunsite.unc.edu` in the `/pub/Linux/devel/lang/smalltalkx` directory. Once there, you will find the following files:

- **COPYRIGHT** As the filename suggests, this is the SmallTalk/X copyright information document.
- **INDEX** Provides a list of files in the directory.
- **LICENSE** The distribution license for SmallTalk/X.
- **README** Contains a brief discription of how to get SmallTalk/X running.
- **bitmaps.tar.Z** Contains sample graphics that can be used to dress up the visual appeal of your programs.
- **doc.tar.Z** The documentation set for SmallTalk/X. This package also includes several demos.
- **exe.tar.Z** The executable archive. Although this is really the only file you need to download to get SmallTalk/X up and running, installing the bitmap, documentation, goodies, and source files is also highly recommended.
- **goodies.tar.Z** Various extras thrown in for your enjoyment.
- **source.tar.Z** Support and library files needed to get the most out of SmallTalk/X.

To install SmallTalk/X, perform the following steps as `root`.

1. Create a directory called `/usr/local/lib/smalltalk`.
2. Copy the following files into the `/usr/local/lib/smalltalk` directory.

[bitmaps.tar.Z](#)
[doc.tar.Z](#)
[exe.tar.Z](#)
[goodies.tar.Z](#)
[source.tar.Z](#)

3. Uncompress and untar these files by entering the following commands:

```
uncompress *.Z
tar -xf bitmaps.tar
tar -xf doc.tar
tar -xf exe.tar
tar -xf goodies.tar
tar -xf source.tar
```

4. You can now delete all of the `tar` files by typing the following command:

```
rm -f *.tar
```

5. Finally, create the following links:

```
ln /usr/i486-linuxaout/lib/libX11.so.3.1.0 /usr/lib/libX11.so.3
ln /usr/i486-linuxaout/lib/libXt.so.3.1.0 /usr/lib/libXt.so.3
```

The SmallTalk/X program should now be installed and ready to go.

NOTE: If you do not have write access to the `/usr/local/lib` directory, install SmallTalk/X in some other directory by following the same steps listed above. If you do this you must set the `SMALLTALK_LIBDIR` variable to be equal to the new directory.

Invoking SmallTalk/X

You invoke SmallTalk/X by typing

```
smalltalk
```

in an Xterm window. When ST/X starts, it checks to see if there is an image file for it to use. If it cannot find an image file, it uses a file called `smalltalk.rc` to set up the default behavior for your environment. The image file that is loaded by default is called `st.img`, and contains a snapshot of what your ST/X environment looked like the last time you exited. This allows you to resume exactly where you left off. You can save a snapshot under any name with the extension `.img`. To invoke ST/X with an image other than `st.img`, type the following command at the prompt:

```
smalltalk -i nameofImage.img
```

Getting Around in ST/X

Once ST/X is invoked, two windows or views will appear. The Transcript view and the Launcher menu. The Transcript view is shown in Figure 36.1.

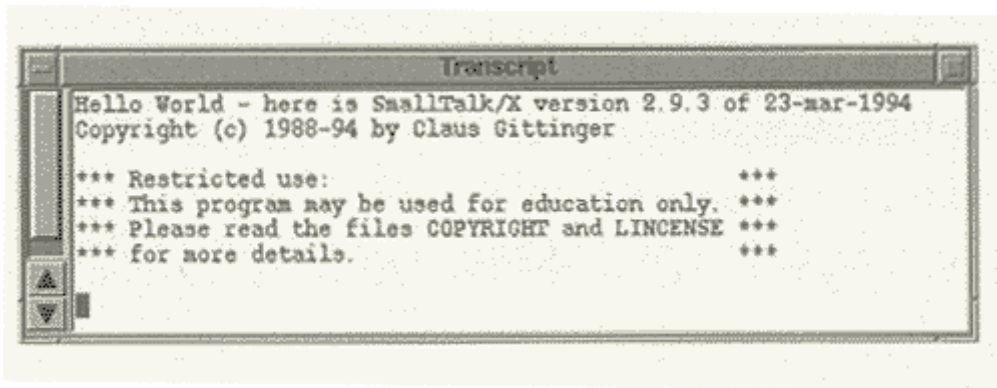


Figure 36.1. *The Transcript view.*

The Transcript is the console where relevant systems information is shown. The Launcher menu is shown in Figure 36.2.

The Launcher allows access to the tools you will need to program your application. Table 36.1 gives the options available from the Launcher and a brief description of each.

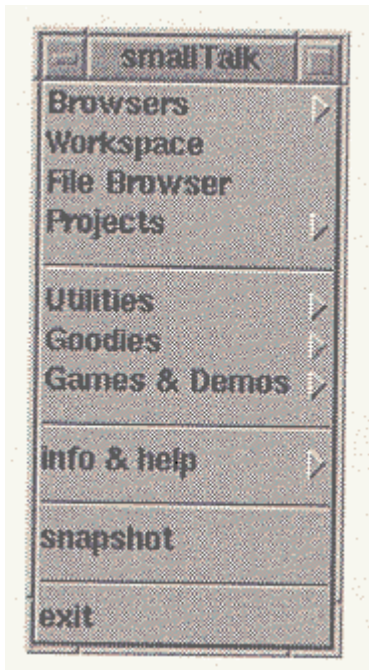


Figure 36.2. *The Launcher menu.*

Table 36.1. The Launcher menu options.

<i>Option</i>	<i>Description</i>
Browsers	The pull-right menu of this option gives you access to browsers, senders, and implementors.
Workspace	This option brings up a workspace view.
File Browser	This browser allows inspection and manipulation of files and directories.
Projects	This option allows you to choose an existing or new project.
Utilities	This contains tools specific to your programming needs.
Goodies	This contains other non-programming related tools.
Games & Demos	This contains some sample programs and games to play.
info & help	This contains topics that give you help and information on the ST/X environment and programming in SmallTalk.
snapshot	This option takes a snapshot of your present ST/X environment and asks for the name of the image file you wish to store the snapshot in.
exit	This option allows you to exit ST/X immediately or exit and save a snapshot of the current environment.

The following sections describe most of these options in more detail.

The Browsers Option

The Browsers option in the Launcher menu gives you access to different browsers or editors that let you read and manipulate classes, methods, changes, senders, and implementors. The suboptions available are

- System Browser
- Class Hierarchy Browser

- Implementors
- Senders
- Changes Browser
- Directory Browser

Each of these suboptions will be discussed in detail in this section.

The System Browser

The standard System Browser contains five subviews:

- Class category list
- Class list
- Method category list
- Method list
- Code view

The System Browser is shown in Figure 36.3.

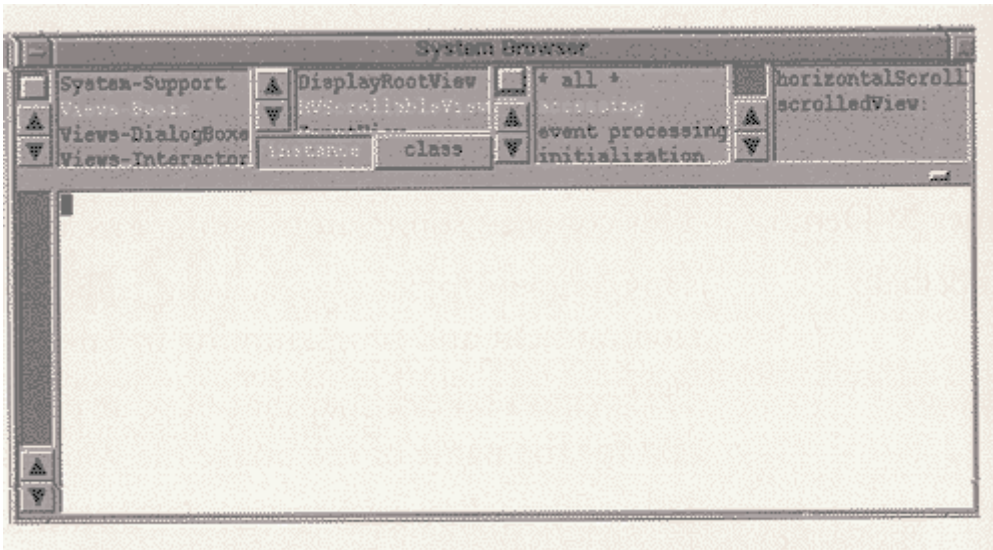


Figure 36.3. *The System Browser.*

Within the ST/X system, classes are assigned to a category. A category is simply an attribute used for grouping classes to make them easier to handle. To select a class category, click on the name of the category in the class category list. This is the leftmost section of the top half of the System Browser. This will display, in the class list subview, all classes belonging to that category. The class list subview is the second section from the far left of the system browser. You can also select one of two special categories: `* all *`, which selects all classes and lists them alphabetically; and `* hierarchy *`, which lists all classes in a tree by inheritance.

If you select a class in the class list, all method categories for that class will be displayed in the method category list, which is

the second section from the right in the top half of the System Browser. Like class categories, method categories are simply for grouping methods according to their function. When you select a method category, all methods in that category are shown in the method list view in the far right section of the browser. The special `* all *` category will show all methods in alphabetical order. Selecting a method from the method list will show the corresponding method's source code in the code view which is the bottom half of the System Browser.

The browser enables you to change either a class or its metaclass. There are two toggle buttons, class and instance, in the same section of the browser as the class list view. Instance, which is the default, makes the changes affect the class. Selecting class makes the changes affect the metaclass. A pop-up menu is available in each view by pressing the middle or menu mouse button while the pointer is in that view. The pop-up menu available in the class category view is shown in Figure 36.4, and the purpose of each function is shown in Table 36.2.

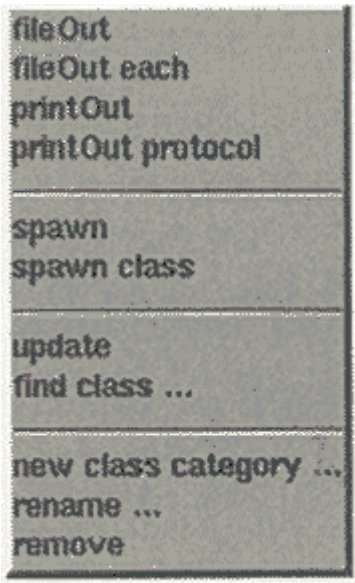


Figure 36.4. *The class category pop-up menu.*

Table 36.2. Class category pop-up menu functions.

<i>Function</i>	<i>Description</i>
fileOut	Saves all classes in the currently selected class category into one source file named <code>classCategory.st</code>
fileOut each	Saves all classes but puts each class into a separate file called <code>className.st</code>
printOut	Sends a printed representation of all classes selected to the printer including the method source code
printOut protocol	Sends a protocol-only representation of all classes in the category to the printer without the method's source code
spawn	Starts a class category browser without a class category list on the currently selected class category
spawn class	Starts a full class browser which allows you to edit all code for the selected class in one view
update	Rescans all classes in the system and updates the lists shown
find class	Pops up a dialog box to enter the name of a class you want to search for and have displayed
rename	Renames a category and changes the category attribute of all classes in the currently selected class category
remove	Removes all classes and subclasses in the current class category

The class list pop-up menu appears when you press the menu mouse button with the pointer in the class list view. The functions available from this menu are shown in Figure 36.5 and are explained in Table 36.3.

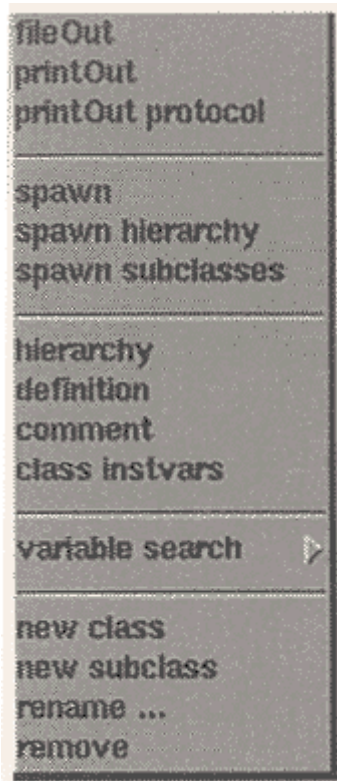


Figure 36.5. *The class list pop-up menu.*

Table 36.3. Class list pop-up menu functions.

<i>Function</i>	<i>Description</i>
fileOut	Saves the source code of the currently selected class in a file named <code>className.st</code> .
printOut	Sends the source code of the currently selected class to the printer.
printOut protocol	Sends a protocol description of the currently selected class to the printer. The output will contain the class description, class comment, and the classes' protocol and method comments.
spawn	Starts a class browser on the currently selected class.
spawn hierarchy	Starts a browser on all subclasses of the currently selected class.
hierarchy	Shows the hierarchy of the currently selected class in the code view.
definition	Shows the class definition in the code view and allows you to change the class definition.
comment	Shows the class comment in the code view and allows you to edit it.
class instvars	Shows the class-instance variables for the selected class and allows you to edit them.
variable search	Provides a search facility to find different variable references and all methods referencing the searched-for variable.
new class	Allows you to create a new class using as a template the currently selected class.
new subclass	Same as new class but it will create a subclass of the currently selected class.
rename	Changes the name of the currently selected class.

remove Removes the currently selected class and all of its subclasses. The method category pop-up menu appears when you press the menu mouse button while the pointer is in the method category view. The functions available from this menu are shown in Figure 36.6 and explained in Table 36.4.

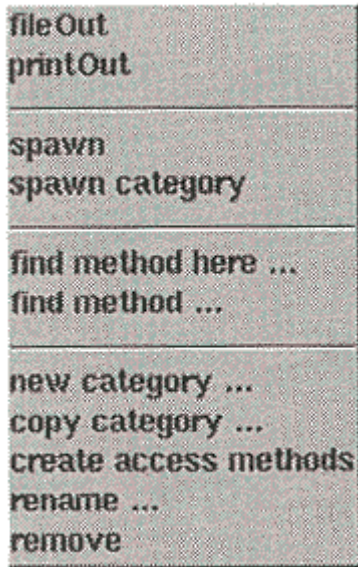


Figure 36.6. *The method category pop-up menu.*

Table 36.4. Method category pop-up menu functions.

<i>Function</i>	<i>Description</i>
fileOut	Saves the source code of the currently selected method category in a file named <code>className-category.st</code>
printOut	Sends the source code of the currently selected method category to the printer
spawn	Starts a method category browser on the currently selected method category of the currently selected class
spawn category	Starts a browser on all methods of the class which have the same category as the currently selected one
find method here	Searches for the method that implements a specified selector
find method	Searches in the class hierarchy for the first class implementing the selector you specify in the dialog box
new category	Enables you to add a new category to the list
copy category	Enables you to copy all methods in a class category to the currently selected class
create access methods	Creates methods to access instance variables
rename	Renames the currently selected method category
remove	Removes all methods in the currently selected class that are members of the currently selected method category

The method list pop-up menu appears when you press the menu mouse button while the pointer is in the method list view. The functions available from this menu are shown in Figure 36.7 and explained in Table 36.5.



Figure 36.7. *The method list pop-up menu.*

Table 36.5. Method list pop-up menu functions.

<i>Function</i>	<i>Description</i>
fileOut	Saves the currently selected method in a file named <code>className-selector.st</code>
printOut	Sends the source code of the currently selected method to the printer
spawn	Starts a browser for editing this method
senders	Starts a new browser on all methods sending a specific message
implementors	Starts a new browser on all methods implementing a specific message
globals	Starts a new browser on all methods that are accessing a global that is either a global variable or a symbol, as well as all methods sending a corresponding message
local senders	Same as <code>senders</code> but limits the search to the current class and its subclasses
new method	Enables you to create a new method from a template in the code view
change category	Enables you to change the category of the selected method
remove	Removes the currently selected method

When you add or remove instance variables to or from a system class description and accept (that is, save the changes), the system creates a new class instead of changing the old one. The original class still exists to give existing instances of the class a valid class even though it is no longer accessible by name. After the change, you can no longer edit the old class.

NOTE: It is recommended that you don't change the definition of system classes but only private ones. It is safer to use the `copy category` function to copy an existing class and its methods to a new class and modify the new class. This is especially important for classes which are used by the system itself since changes can lead to problems in the operation of the ST/X environment.

The code view is the lower half of the System Browser. It is here that you can modify the class or instance definitions as well

as methods. The pop-up menu for this area is the edit menu that appears in every text editing view in ST/X. The functions in this menu are discussed in the "Editing in Browsers" section of this chapter.

The Class Hierarchy Browser

When the Class Hierarchy Browser is selected, a dialog box appears which asks for the name of class. If you enter a valid class, the Class Hierarchy Browser appears for that class. This is the same as the System Browser except there is no class category list since this is for one specific class. The pop-up menus for each of the four subviews are the same as in the System Browser.

Implementors

When the Implementors option is selected, a dialog box appears which asks for a selector. A selector is the name of the type of operation a message requests of its receiver.

If you enter a valid selector, an Implementors view will be displayed. This view is similar to the one shown in Figure 36.8.

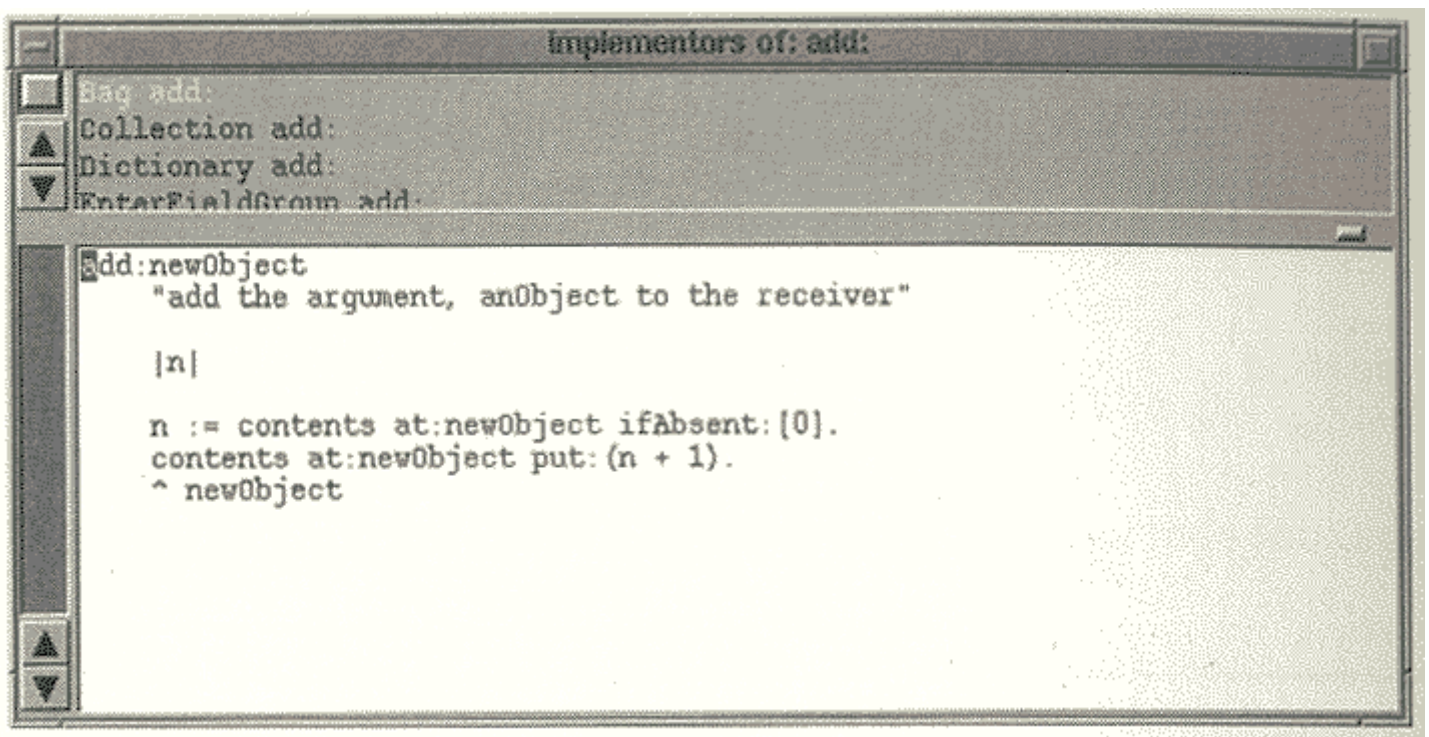


Figure 36.8. *The Implementors view.*

The Implementors view contains a list of the methods that implement the method specified by the selector. The pop-up menu for the top half of the Implementor view is the same as the pop-up menu for the method list subview which was discussed earlier in the section "The System Browser."

Senders

When the Senders option is selected, a dialog box appears that asks for a selector. If you enter a valid selector, then a Senders view will be displayed. This view is similar to the one shown in Figure 36.9.

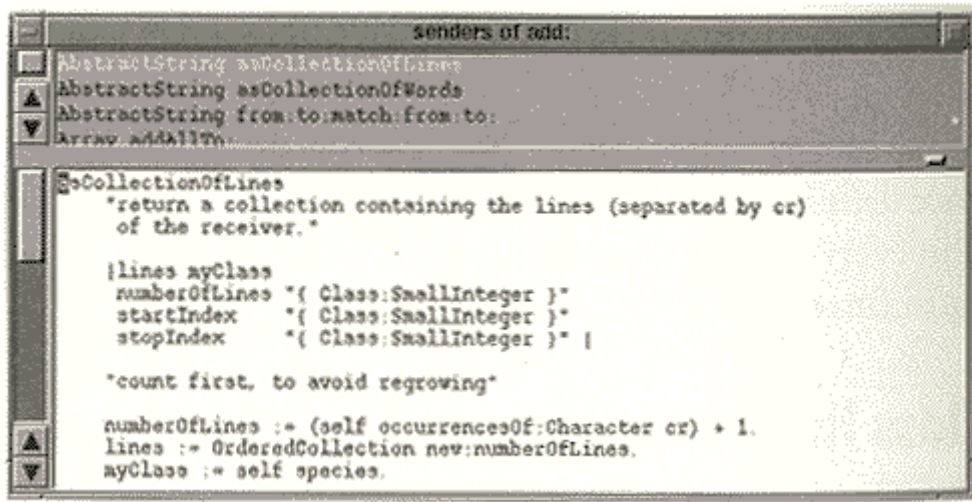


Figure 36.9. *Senders view.*

The Senders view contains a list of the methods that send the selected message. The pop-up menu for the top half of the Senders view is the same as the pop-up menu for the method list subview which was discussed in the section "The System Browser."

The Changes Browser

Each time you make a change to either the class hierarchy or to a method, ST/X writes a record to a changes file. The Changes Browser enables you to inspect and manipulate the changes file. There are two subviews in the Changes Browser; the change list and the contents view. The change list gives a list of all changes in chronological order. A sample Changes Browser is shown in Figure 36.10.

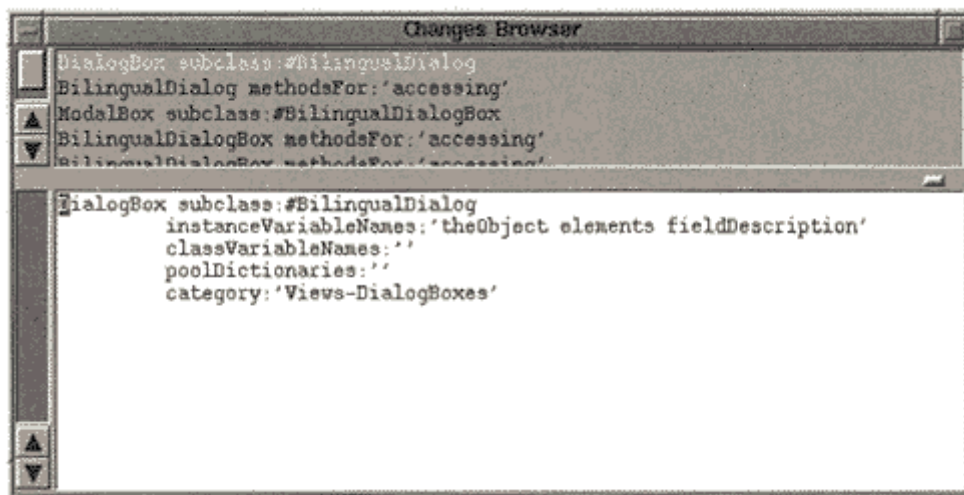


Figure 36.10. *The Changes Browser.*

To display a change, select one of the changes from the change list. The change browser then displays the contents of the change in the contents view.

The pop-up menu for the change list has the functions described in Table 36.6.

Table 36.6. The Change list pop-up menu.

<i>Function</i>	<i>Description</i>
apply change	Applies the currently selected change.
apply to end	Applies all the changes from the currently selected change to the end of the changes file.
apply all changes	Applies all the changes in the file.
delete	Deletes the currently selected change from the list.
delete to end	Deletes all changes from the currently selected change to the end of the file.
delete changes for this class to end	Deletes all changes affecting the same class as the currently selected change to the end of the changes file.
delete all changes	Deletes all changes in the file for the same class as the currently selected change.
update	Rereads the changes file.
compress	Compresses the change list and removes multiple changes of a method and leaves the most recent change compared to current.
version	Compares a method's source code in a change with the current version of the method and outputs a message in the Transcript view.
make a change patch	Appends the change to the end of the patches file which will be run and automatically applied at ST/X startup.
update sourcefile from change	This function is not currently implemented.
writeback	Writes the change list back to the <code>changefile</code> changes file. All delete/compress operations performed in the Change Browser will not affect the changes file unless this operation is performed.

The Change Browser can be used to recover from a system crash by reapplying all changes that were made after the last snapshot entry.

NOTE: To control the size of the changes file, it is a good idea to apply a compress periodically. This will remove all old changes for a method leaving the newest one.

Directory Browser

When you select the Directory Browser option, a browser with five subviews is displayed. The top half of the browser displays the current directory and all subdirectories and files contained in it. If you select a directory, it is expanded in the next section to the right across the top half of the browser. If you select a file, the contents of the file are displayed in the lower half of the Browser. The pop-up menu for the directory area has only two functions:

- `up--`Moves up to the directory above the one selected
- `goto directory--`Enables you to go to a specified directory

The content view has the same edit menu as all the other text editors and is discussed in the "Editing in Browsers" section, later in this chapter. A typical Directory Browser is shown in Figure 36.11.

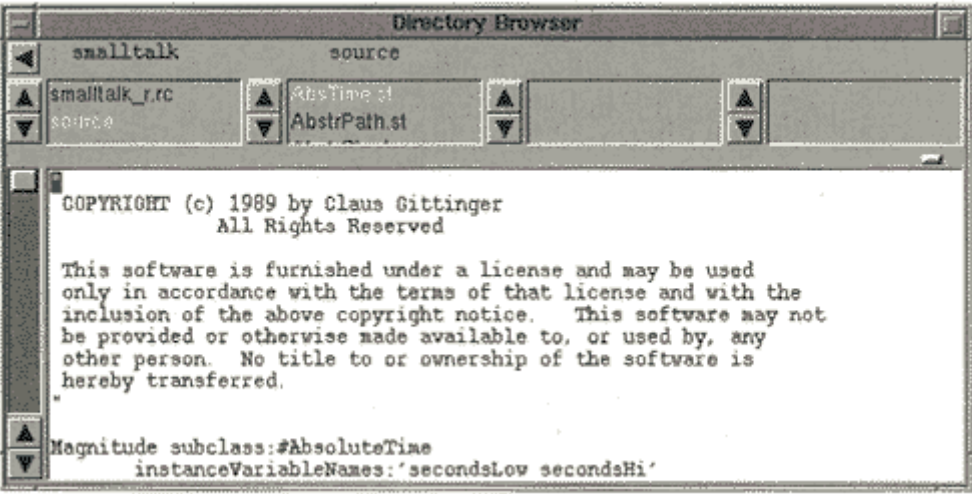


Figure 36.11. The Directory Browser.

The Workspace Option

The Workspace option displays a view from which you can enter and compile SmallTalk code. The Workspace is usually used as a testing area or scratch pad when coding. You can use it to test your SmallTalk code before building it into the code library using the System Browser code view.

The File Browser Option

The File Browser gives you the ability to inspect and manipulate files and directories. The File Browser is shown in Figure 36.12.

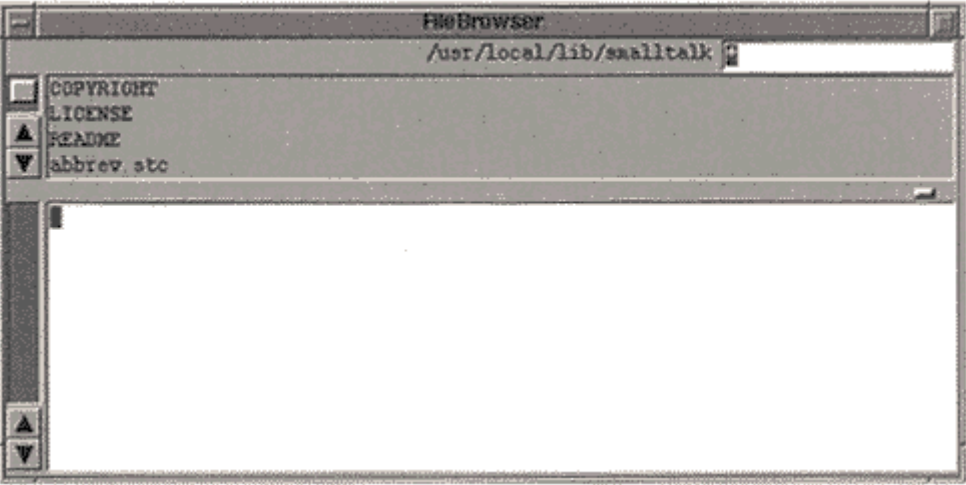


Figure 36.12. The File Browser.

It consists of four subviews that are described in Table 36.7.

Table 36.7. The File Browser subviews.

Subview	Purpose
---------	---------

path-label field	Shows the name of the current directory
file pattern field	Allows a search pattern to be entered for choosing files for the file list
file list	Shows a list of file and directory names
contents view	Shows the contents of a selected file

To inspect the contents of a file, double-click the left mouse button on the name of the file in the file list. To change directories, double-click on the directory name. Directory names are always shown in the file list.

You can use the file pattern field to display the list of files matching the specified pattern. The default is `*`, which shows all files. The search pattern can be changed by moving the pointer to the field, editing the pattern, and then pressing enter or choosing `accept` from the file pattern pop-up menu.

As in the other browsers we have discussed, each subview has a pop-up menu that is activated by the menu mouse button. The path-label pop-up menu is shown in Figure 36.13.

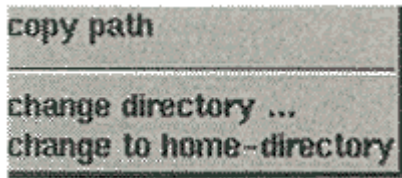


Figure 36.13. *The path-label pop-up menu.*

The functions available in this menu are described in Table 36.8.

Table 36.8. The path-label pop-up menu functions.

<i>Function</i>	<i>Purpose</i>
<code>copy path</code>	Copies the current pathname into the cut and paste buffer
<code>change directory</code>	Opens a dialog box to enter the name of the directory you want to change to
<code>change to home-dir</code>	Changes the file list to your home directory

The file list pop-up menu is shown in Figure 36.14.

The functions available in this menu are described in Table 36.9.

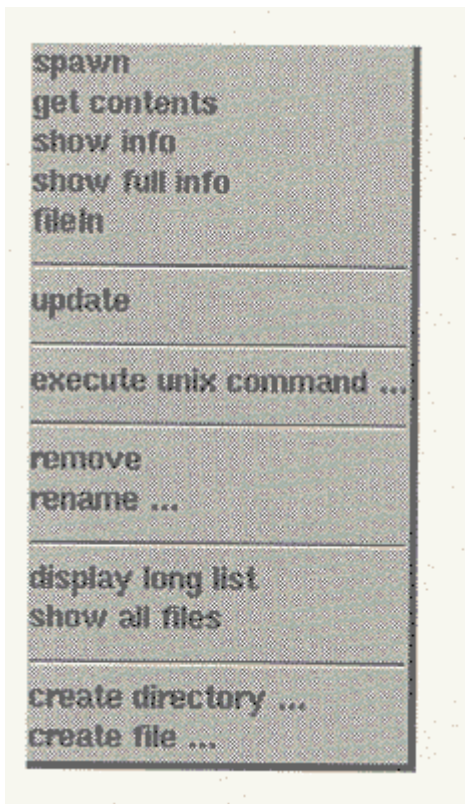


Figure 36.14. *The file list pop-up menu.*

Table 36.9. The file list pop-up menu functions.

<i>Function</i>	<i>Purpose</i>
spawn	Starts another file browser on the current directory or the directory selected in the file list.
get contents	Shows the contents of the currently selected file in the contents view.
show info	Displays a view with type, size, access, and owner information for the currently selected file or directory.
show full info	Displays the same as above with more details such as the last access, last modification date, and time.
fileIn	Loads the selected file into the system by reading and evaluating SmallTalk expressions from it.
update	Rereads the directory and updates the file list.
execute unix command	Allows execution of any UNIX command through a pop-up box.
remove	Removes the selected file(s) or directory(s).
rename	Renames the selected file.
display long list	Shows file information in the file list. This option toggles with <code>display short list</code> , which is the default.
show all files	Displays all the files including hidden files. This option toggles with <code>hide hidden files</code> , which is the default.
create directory	Creates a new directory.
create file	Creates a new file.

The pop-up menu for the contents view is the same edit menu as the other text editors and is discussed in the "Editing in Browsers" section of this chapter.

The Projects Option

The Projects option of the Launcher menu enables you to create a new project or select a previously created project. When the new project function is selected, a new project is automatically created for you and the new project object appears on your screen. If you select the select project function, a dialog box appears with a list of existing projects from which to choose. Simply select a project and it will be loaded in to the environment.

The Utilities Option

The Utilities option provides 13 tools that assist you in programming in the ST/X environment. Table 36.10 gives you a brief description of each tool.

Table 36.10. The Utilities option tools.

<i>Utility</i>	<i>Description</i>
Transcript	Opens the Transcript view.
Window tree	Displays a graphical tree representation of the window hierarchy of all windows that are active or in wait state at the time it was requested.
Class tree	Displays a graphical tree representation of the class hierarchy of the system.
Event monitor	Displays a view that monitors events.
Process monitor	Displays a view that gives information about all currently active or waiting processes. This information changes as the state of the processes change.
Memory monitor	Displays a graph that tells you the present memory usage and changes as the memory usage changes.
Memory usage	Displays a table of the classes and the number of instances of each, average size, bytes, and percentage of memory used by each.
Collect Garbage	Runs a Generation Scavenge algorithm to collect short term objects and destroy them. If an object survives long enough, it is moved to an area of memory where it remains until the user requests its collection.
Collect Garbage & compress	Same as Collect Garbage but also compresses to recover space.
Fullscreen hardcopy	Takes a picture of the screen and asks you for a name of a file with a .tiff extension in which to save the image.
Screen area hardcopy	Same as Fullscreen hardcopy but for only a specific area of the screen.
View hardcopy	Same as Fullscreen hardcopy but for one specific view only.
ScreenSaver	Enables you to choose from one of three different screen savers to use in the ST/X environment.

The Goodies Option

The Goodies option of the Launcher menu provides a pull-right menu of six different tools that are useful at any time, not just when you program in SmallTalk. The Goodies are described in Table 36.11.

Table 36.11. The Goodies.

<i>Tool</i>	<i>Description</i>
Clock	Displays an analog clock in a square with a toggle for the second hand.
Round Clock	Same as the clock but it's round and remains visible when it is minimized.
Directory View	Displays a pictorial representation of files and directories. A folder represents a directory and a document is a file.

Mail Tool	A tool for managing electronic mail.
News Tool	A repository for news, information, and documents.
Draw Tool	A fairly comprehensive tool for drawing diagrams, charts, pictures, and so on.

The Games & Demos Option

Contained in the pull-right menu of this option are games for your enjoyment and example applications that may be useful. The Games & Demos option menu is shown in Figure 36.15.

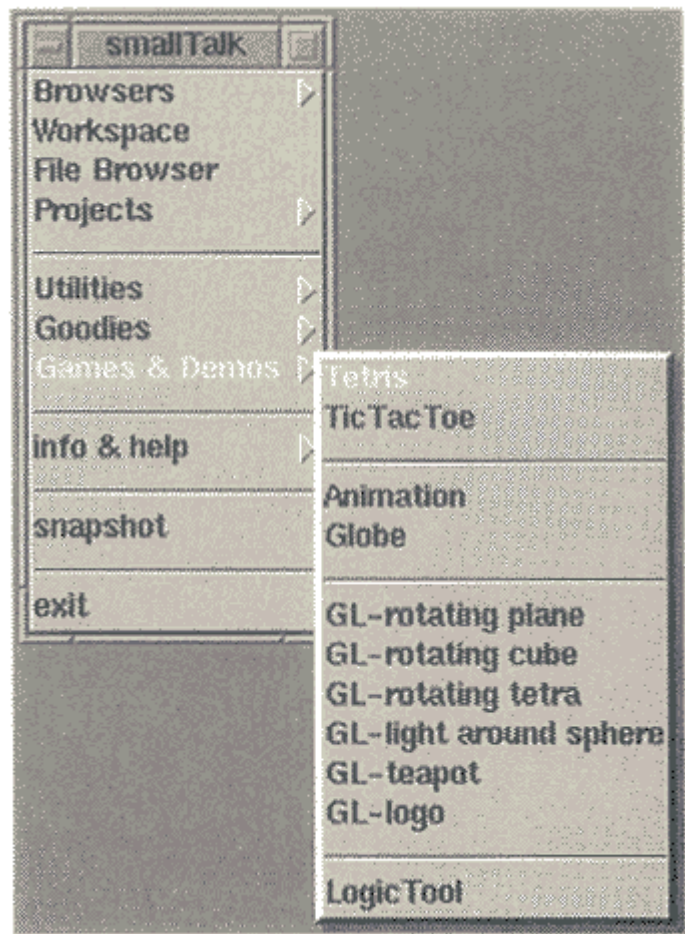


Figure 36.15. *The Games & Demos option menu.*

Editing in Browsers

All views that show text allow the usual editing functions of that text through a pop-up menu. The functions available in this menu are described in Table 36.12.

Table 36.12. Editing functions.

<i>Function</i>	<i>Description</i>
again	Repeats the last edit.
copy	Copies the selected text.
cut	Cuts the selected text out of the file.

<code>paste</code>	Pastes the text that was copied or cut prior to choosing the paste option to the current position of the pointer.
<code>accept</code>	Once you have completed editing, you must use this option to save the changes to the file; otherwise, the changes will not be written to the file.
<code>doIt</code>	Evaluates the highlighted text.
<code>printIt</code>	Prints a representation of the result of the evaluation at the current cursor position.
<code>inspectIt</code>	Invokes the Inspector view on the result.
<code>search...</code>	Enables you to search for a specific string.
<code>goto...</code>	Enables you to move to a specific location in the file.
<code>font...</code>	Enables you to change the font of the file.
<code>indent...</code>	Enables you to change the indenting of the file.
<code>save as...</code>	Enables you to save the file under a different name.
<code>print</code>	Prints the file.

To select or highlight text, press the left mouse button over the first character and move the mouse (while pressing the mouse button) to the end of the text you wish to select and then release the mouse button. If you press the left mouse button again, the highlighting is removed and you can select something else.

To scroll through the text, use the scroll bars on the left of the view. By clicking the mouse below or above the thumb, the text scrolls one page for every click. If you press the Shift key at the same time as you click, the text scrolls to the position of the pointer in the scroll bar. This is useful for scrolling rapidly through long documents.

Using the Inspector

The inspector enables you to inspect an object. It consists of two subviews, one showing the names of the object's instance variables and the other showing the value of the selected instance variable. You can start an inspector by using the `inspectIt` function on the edit menu or by sending one of the following messages to an object:

```
anObject inspect
```

or

```
anObject basicInspect
```

The `basicInspect` command will open a general inspector that shows instance variables as they are physically present in the object. The `inspect` command is redefined in some classes to open an inspector showing the logical contents of the object.

Using the Debugger

The debugger is displayed whenever an error occurs in your SmallTalk code. It shows you where the error occurred and how the system got there. The debugger runs in one of three modes: `normal`, `modal`, and `inspecting`.

When in normal mode and an error occurs in a process, which is not the event handler process, the debugger will start up on top of the erroneous process. This blocks all interaction with the affected process and its views. Other views are still active and respond as usual.

When an error occurs in the SmallTalk event handler process, the debugger starts in modal mode. While a modal debugger is active, you cannot interact with any other view.

The inspecting mode can be entered from the ProcessMonitor by the pop-up menu and allows inspection of the state of other processes. But since the debugged process may continue to run, it is only possible to inspect a snapshot of the affected process.

The debugger contains four subviews:

- The Context Walkback List shows the context chain that led to the error.
- The Method Source View shows the method that caused the error.
- The Receiver Inspector allows inspection of the receiver of the selected message.
- The Context Inspector provides information about the arguments and local variables of this context.

The debugger is shown in Figure 36.16.

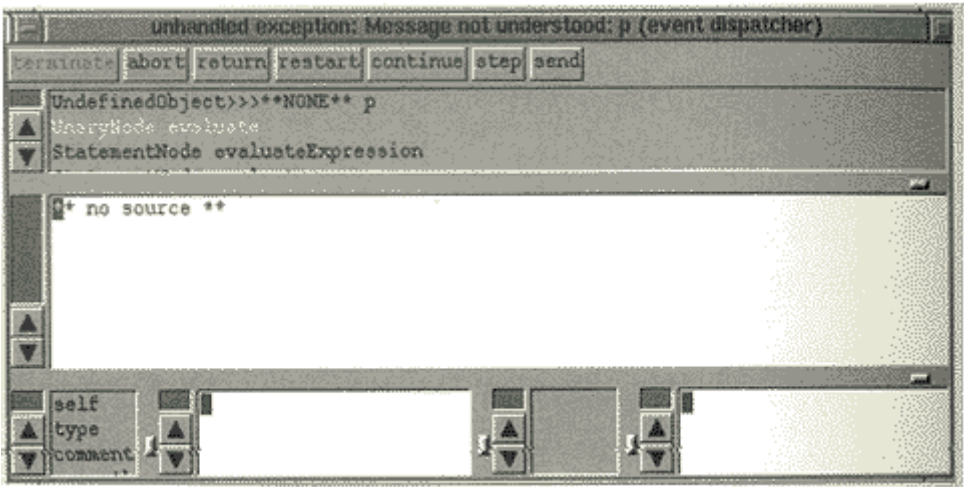


Figure 36.16. *The debugger.*

The functions that are common to each view appear as a set of buttons below the Context Walkback List. These functions are described in Table 36.13.

Table 36.13. The debugger function buttons.

<i>Button</i>	<i>Description</i>
continue	Continues execution
terminate	Terminates the erroneous process
abort	Aborts the current activity if possible
step (single step)	Lets the process continue execution until the next send is executed in the currently selected context
send (single send)	Lets the process continue execution for one message send
return	Continues execution as if the selected context returned
restart	Continues execution by restarting the selected context

The Walkback subview has a pop-up menu with the functions described in Table 36.14.

Table 36.14. Walkback subview pop-up menu function.

<i>Function</i>	<i>Description</i>
exit	Leaves ST/X without saving an image
smalltalk	
show more	Shows 50 more contexts of the Walkback
breakpoints	Not yet available
trace	Not yet available
on/off	
trace step	Not yet available

A minidebugger is entered if an error occurs within the debugger itself. This is a line-by-line debugger that allows limited debugging without the use of a graphical user interface. It is controlled by entering commands in the Xterm window where ST/X was started. If you type ? at the `miniDebugger` prompt, you will get a list of commands that are available for use in this stripped-down debugger.

Summary

This chapter introduced you to the SmallTalk environment that is provided by the SmallTalk/X application. If you are interested in learning how to program using SmallTalk and do not have access to one of the commercial versions of SmallTalk, then SmallTalk/X is perfect for you. Not only does SmallTalk/X come with all the tools and programming aids that were talked about in this chapter, but it also comes with many examples and some fairly complete documentation that will make learning SmallTalk easier for you.

- [Mathematics on Linux](#)
 - [Scilab](#)
 - [Where to Get Scilab](#)
 - [NOTE](#)
 - [More Information on Scilab](#)
 - [Pari](#)
 - [Where to Get Pari](#)
 - [Running Pari](#)
 - [Using LISP-STAT](#)
 - [Where to Get LISP-STAT](#)
 - [Running xlipstat](#)
 - [Where to Get More Information About LISP-STAT](#)
 - [A Last Note](#)
 - [Summary](#)
-

Mathematics on Linux

by Kamran Husain

IN THIS CHAPTER

- Scilab
- Pari
- Using LISP-STAT
- A Last Note

This book has dealt with many issues regarding the tools available for Linux. Now, let's look at some of the mathematics tools for Linux. Specifically, we will work with tools for doing mathematical and statistical applications under Linux. One such tool we will be working with is Scilab, an interactive math and graphics package. Another tool for symbolic math is Pari. For statistical operations using LISP choose LISP-STAT.

Hopefully, this chapter will give you a comfortable alternative to writing applications in languages other than FORTRAN on Linux. (The FORTRAN compiler is called `f77`.)

Scilab

The Scilab application is developed by the Institut National de Recherche de Informatique et en Automatique (INRIA) in France. Although this application is not as formidable as MATLAB, a commercial product with more bells and whistles, Scilab is still powerful enough to provide decent graphics and solutions to math problems.

With Scilab you can do matrix multiplication, plot graphs, and so on. Using its built-in functions, Scilab enables you to write your own functions. With its toolbox, you can build your own signal-processing functions in addition to those provided by Scilab.

Added to all its features, the help file is quite voluminous. If you want to find out how to do a math problem with Scilab, you will probably find it in the docs. Added to the good documentation are sample programs to get you started.

Where to Get Scilab

Now that you are probably interested in Scilab, you will want to know where to get it. Scilab is free via the Internet. The primary site is `ftp.inria.fr`, and the directory for this is in `INRIA/Projects/Meta2/Scilab`. Look for the zipped file with the latest date. Each zipped file is complete in itself. Mirror sites include `sunsite.unc.edu` and `tsx-11.mit.edu`.

The file you are looking for is called `scilab-2.1.1-linux.tar.gz`. In its unzipped form, the file is about 17MB in size. After untarring this file, you will have a directory called `scilab-2.1.1` with all related files and accessories.

After you have installed it, go to the `bin` subdirectory and modify the Scilab shell script file. Replace the assignment of the `SCI` variable with the path to the location of your Scilab files. For example, in my case I set the value to

```
SCI="/home/khusain/scilab-2.1.1"
```

NOTE: If Scilab does not show up in color the first time you invoke it, try
 *customization: -color in your .Xdefaults file. Don't forget to run
 xrdb .Xdefaults to enforce the change.

The prompt for Scilab is `-->`. You will see responses to your commands immediately below where you type in entries.

A healthy example of how to use Scilab would probably be beneficial. Let's see how to declare values:

```
-->x=1.0
```

This sets x equal to 1.0. To declare an array, use square brackets:

```
-->x=[ 1  2  3 ]
```

```
x =
```

```
!  1.  2.  3.  !
```

See Figure 37.1 to see what it looks like on your screen.

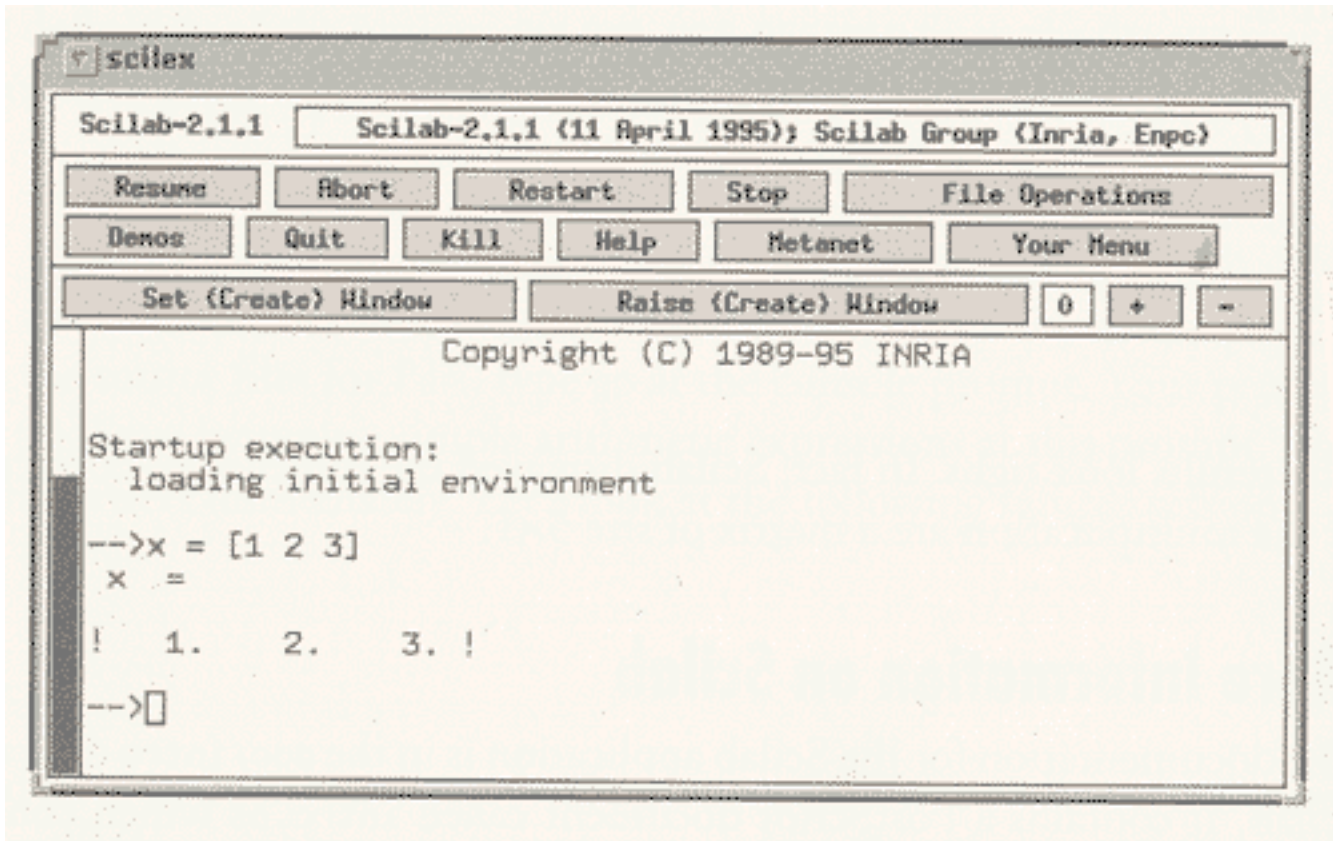


Figure 37.1. *The main screen for Scilab.*

To declare a large array you can use indices of the form [start:end]. Use a semicolon at the end of the line to indicate that you really do not want Scilab to echo the results back to you. So the statement

```
-->x=[1:100];
```

declares `x` as a vector of values from 1 to 100 and does not display the contents of `x` back to you. If you want to give staggered values of `x`, you can use an increment operator in the form [start:increment:stop]. So, this statement declares `x` to contain five odd numbers from 1:

```
-->x=[1:2:10]
```

```
x =
```

```
! 1. 3. 5. 7. 9. !
```

Let's try an example of a simple matrix multiplication problem of $ax=b$. First declare the a matrix, separating all the rows with semicolons. If you do not use semicolons, the values in matrix a will be interpreted as a 25x1 vector instead of a 5x5 matrix.

```
-->a=[ 1 1 0 0 0; 1 1 1 0 0; 0 1 1 1 0; 0 0 1 1 1; 0 0 0 1 1]
```

```
a =
```

```
! 1. 1. 0. 0. 0. !
```

```
! 1. 1. 1. 0. 0. !
```

```
! 0. 1. 1. 1. 0. !
```

```
! 0. 0. 1. 1. 1. !
```

```
! 0. 0. 0. 1. 1. !
```

Then declare X as a vector.

```
-->X=[ 1 3 5 7 9 ]
```



```
X =
```

```
! 1. 3. 5. 7. 9. !
```

To get the dimensions right for the multiplication, you have to use the single quote operator (') to get the transpose of X. Then put the results of the multiplication of a and X transpose into b.

```
-->b= a * X'
```

```
b =
```

```
! 4. !
```

```
! 9. !
```

```
! 15. !
```

```
! 21. !
```

```
! 16. !
```

```
-->
```

The results look right. In fact, Scilab displayed the dimensions correctly too, since the results of the multiplication are a matrix of size 5x1.

More Information on Scilab

The documentation for the Scilab application is in the `doc/intro` directory where you installed Scilab. It contains a PostScript document called `intro.ps` which contains the user's manual titled Introduction to Scilab. Take time to read this manual carefully.

For a list of all the available functions in Scilab, you can look in the `man/LaTeX-doc` directory for a PostScript file called `Docu.ps`. (Yes, the file names are cryptic!) Print this file for a handy paper list of all the available functions. All these functions are still accessible from the help buttons in the front panel.

Pari

The Pari package is useful for doing symbolic mathematical operations. Its primary features include an arbitrary precision calculator, its own programming facilities, and interfaces to C libraries.

Where to Get Pari

To get Pari, use the FTP site `megrez.math.u-bordeaux.fr`, and from the `/pub/pari/unix` directory get the `gplinux.tar.gz` file. The binaries may not work with a later version of Linux because the binaries are built with older versions of shared libraries. If you have a newer version of Linux than the one supported by Pari, either you can edit the sources yourself or wait until the authors of Pari catch up. Sorry.

With the version of Linux on the CD-ROM at the back of this book, you need to compile your own version of Pari. The source files are in the `pari-1.39.03.tar.gz` file. The source tar file unpacks into three directories: `src`, `doc`, and `samples`. You will find the samples very useful indeed.

To compile the sources, run the `Makemakefile` command in the `src` directory. When creating this version, remove the definition of the option `-DULONG_NOT_DEFINED` from the `CFLAGS` macro in the newly created `Makefile`. Then type `make` at the prompt. Be prepared to wait a while for this package to compile.

Running Pari

After you have installed the source files for Pari, type `gp` at the console prompt. Your prompt will be a question mark (`?`). Start by typing simple arithmetic expressions at this prompt. You should be rewarded with answers immediately. Let's look at the following sample session:

```
? 4*8
```

```
%1=32
```

```
? 4/7*5/6
```

```
%2=10/21
```

The answer was returned to us in fractions. To get real numbers, introduce just one real number in the equation. You will then get the answer as a real number. The percent signs identify the returned line numbers.

```
? 4.0/7 * 5/6
```

```
%3=0.476190476190476190476190
```

To set the precision in number of digits, you use the `?\precision` command. The maximum number of digits is 315,623, a large number for just about all users. For a modest precision of 10 digits to the right of the decimal point, use this:

```
?\precision=10
```

```
? 4.0/7*5/6
```

```
%4=0.476190047
```

You can even work with expressions, as shown in the following example:

```
? (x+2)*(x+3)
```

```
%5=x^2+5*x+6
```

You can assign values to variables to get the correct answer from evaluating an expression:

```
? x=3
```

```
%6=3
```

```
? eval(%5)
```

```
%7=60
```

This is not where the power of Pari ends, though. You can factor numbers, solve differential equations, and even factor polynomials. The FTP site for Pari contains a wealth of information and samples. See `megrez.math.u-bordeaux.fr`. Also, the `docs` directory contains samples and the manual to help you get started.

Using LISP-STAT

For statistical computing, consider using LISP-STAT. Written by Luke Tierney at the University of Minnesota, LISP-STAT is a very powerful, interactive, object-oriented LISP package.

Where to Get LISP-STAT

The LISP-STAT package is available from `ftp.stat.umn.edu` in the `/pub/xlispstat` directory. Get the latest tar file version you can--currently, `xlispstat-3-44.tar.gz`. In order to build this file you need the dld library for Linux. This dld library is found in `tsx-11.mit.edu` in the `/pub/linux/binaries/libs` directory as `dld-3.2.5.bin.tar.gz`. Install this dld library in the `/lib` directory first.

Running xlispstat

At the command prompt in an Xterm, type `xlispstat`. You will be presented with a `>` prompt. Type commands at this prompt. For example, to multiply two matrices together, use the following command:

```
> (def a (matrix `(3 3) `(2 5 7 1 2 3 1 1 2)))
```

A

```
> (def b (matrix `(3 1) `(4 5 6)))
```

B

```
> (matmult a b)
```

```
#2A((75.0) (32.0) (21))
```

The variables in LISP-STAT are not case sensitive. Note the single quote (‘) before the list of numbers for the matrix. If you omit the quote, the list will be evaluated and replaced with the result of the evaluation. By leaving the single quote in there, you are forcing the interpreter to leave the list in its place.

Let's try solving a simple set of linear equations using LISP-STAT. The following would be a simple example to solve:

$$3.8x + 7.2y = 16.5$$

$$1.3x - 0.9y = -22.1$$

The following script would set up and solve this linear equation problem:

```
> (def a (matrix `(2 2) `(3.8 7.2 1.3 -0.9)))
```

A

```
> (def b (matrix `(2 1) `(16.5 -22.1)))
```

B

```
> (matmult (inverse a) b)
```

```
#2A( (-11.288732394366198) (8.249608763693271))
```

You can do other math operations on lists of numbers as well. See the following example for calculating the mean of a list of numbers:

```
> (def sm (list 1.1 2.3 4.1 5.7 2.1))
```

```
SM
```

```
> (mean SM)
```

```
3.06
```

There are many plotting functions available for LISP-STAT. For plotting one variable, try the function `plot-function`. For (x,y) pairs of numbers, use the `plot-lines` function. For a function of two variables, use the `spin-function`. For 3-D plots, use the `spin-plot` function.

Plots are not limited to lines. You can do histograms, planar plots, and so on. (See Figure 37.2.) See the help pages for details on specifics of how to generate these plots. Two or more plots can be linked together so that changes in one set of data can be reflected in another. You can add points to a plot using the `add-points` function. For reconfiguring how the points are displayed, you can send commands to the plot windows. Plots can be linked together to enable more than one view of the same data.

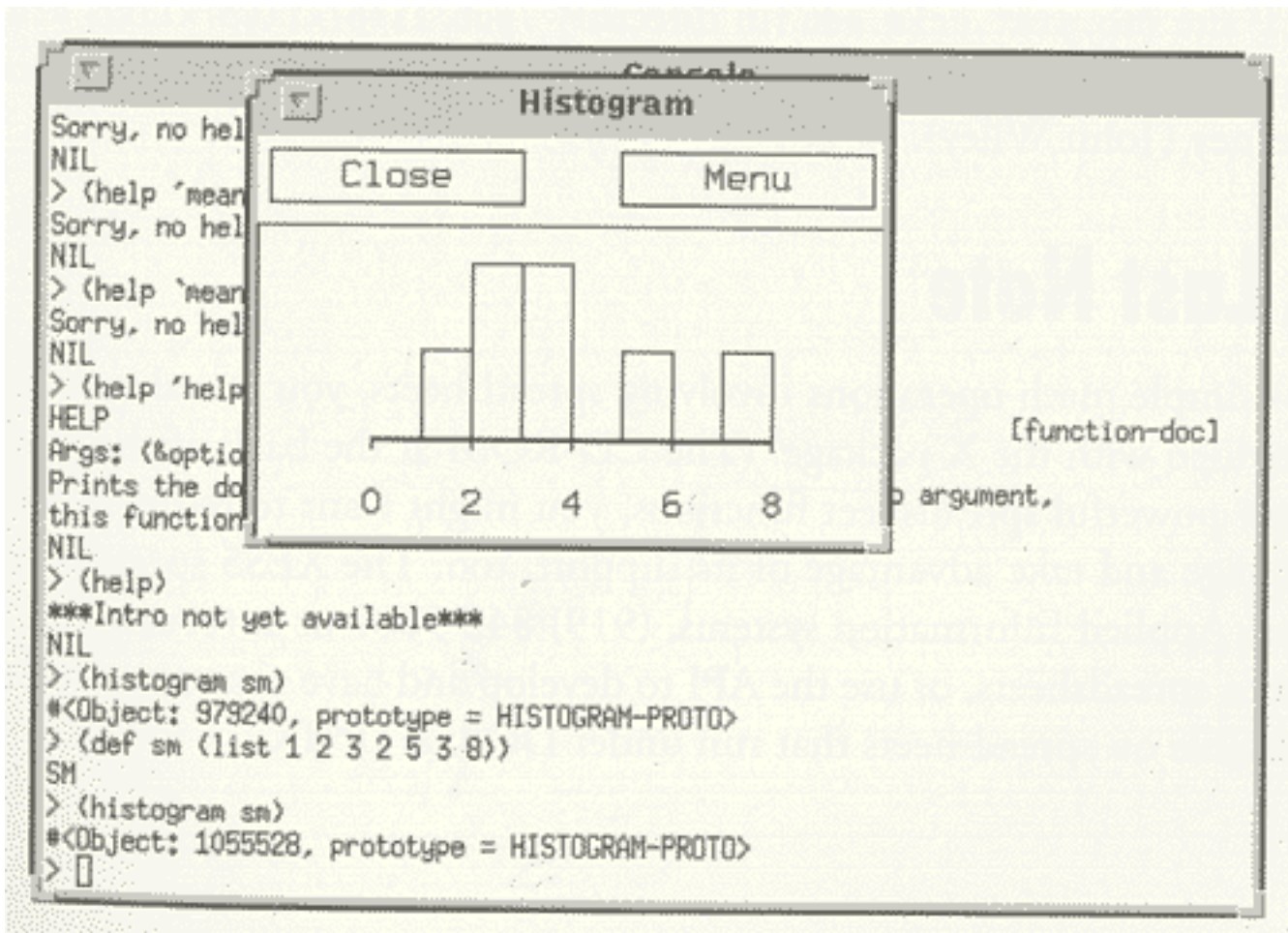
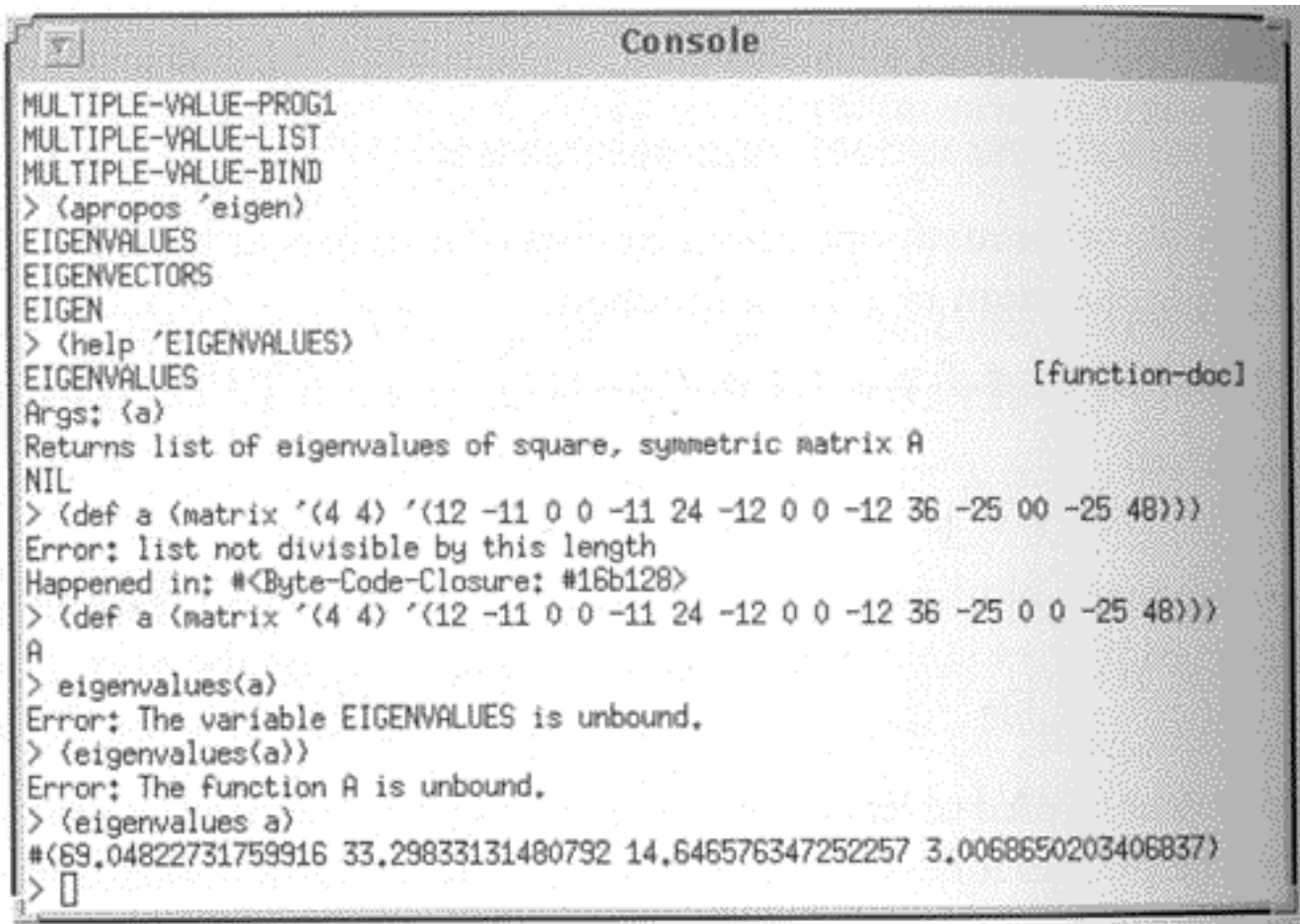


Figure 37.2. A histogram sample.

Each plot is displayed in an X window. You can move the mouse cursor over a point, and it will echo back a value for you.

To get help on this system, use the `help` command. The help documentation for this command should be visible. If nothing shows up, check the environment variables to see if the binaries are in the `PATH`. For example, the command to get help on `EIGENVALUES` and its invocation is shown in Figure 37.3.



```

Console
MULTIPLE-VALUE-PROG1
MULTIPLE-VALUE-LIST
MULTIPLE-VALUE-BIND
> (apropos 'eigen)
EIGENVALUES
EIGENVECTORS
EIGEN
> (help 'EIGENVALUES)
EIGENVALUES                                     [function-doc]
Args: (a)
Returns list of eigenvalues of square, symmetric matrix A
NIL
> (def a (matrix '(4 4) '(12 -11 0 0 -11 24 -12 0 0 -12 36 -25 0 0 -25 48)))
Error: list not divisible by this length
Happened in: #<Byte-Code-Closure: #16b128>
> (def a (matrix '(4 4) '(12 -11 0 0 -11 24 -12 0 0 -12 36 -25 0 0 -25 48)))
A
> eigenvalues(a)
Error: The variable EIGENVALUES is unbound,
> (eigenvalues(a))
Error: The function A is unbound,
> (eigenvalues a)
#(69.04822731759916 33.29833131480792 14.646576347252257 3.0068650203406837)
> 

```

Figure 37.3. *Sample of using EIGENVALUES.*

Where to Get More Information About LISP-STAT

You can get a considerable amount of information from the documents available online at the FTP site `ftp.stat.ucla.edu` (in directory `/pub/lisp/xlisp/xlisp-stat/docs`) or in the Web page at `http://euler.bd.psu.edu`. For more information, read the book *LISP-STAT* by Luke Tierney (John Wiley).

A Last Note

For simple math operations involving spreadsheets, you can always use the `xspread` program provided with the X package. (The CD-ROM at the back of the book has version 2.1.) For more powerful spreadsheet functions, you might want to resort to a commercial spreadsheet package and take advantage of its support, too. The XESS spreadsheet is available for Linux from Applied Information systems, (919) 842-7801 or `info@ais.com`. You can share data between spreadsheets, or use the API to develop and have access to a full suite of math functions available on spreadsheets that run under

DOS or UNIX.

Wolfram Research has released its Mathematica program for Linux. The Mathematica package has extensive numeric and symbolic capabilities, 2-D and 3-D graphics, and a very large library of application programs. With an additional feature called MathLink, you can exchange information between other applications on a network. You can get more information about Mathematica from info@wri.com or <http://www.wri.com>.

Summary

You have several options when it comes to performing mathematical operations or writing such applications under Linux. You can either write the code yourself using C, FORTRAN, or other available languages--or you can use a package. If you are familiar with MATLAB, consider using Scilab. For regular expressions and polynomials, try using Pari. If you are a LISP user or want to do vector operations or statistics, consider using the LISP-STAT package.