THE DEFINITIVE GUIDES TO THE
X WINDOW SYSTEM

VOLUME SIX A

Motif Programming Manual

for Motif 2.1

Open Source Edition

Antony Fountain, Jeremy Huxtable,
Paula Ferguson and Dan Heller

Motif Programming Manual, Open Source Edition

by Antony Fountain, Jeremy Huxtable, Paula Ferguson and Dan Heller

December 2001

Reprinted March 2002 with minor corrections

Copyright © 1991, 1994, 2000, 2001, 2002 O'Reilly & Associates, Inc., Antony Fountain and Jeremy Huxtable. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, v1.0 or later (the latest version is presently available at http://www.opencontent.org/openpub/).

This is an updated version of the Motif Programming Manual, Second Edition, published by O'Reilly & Associates in February 1994. The source files for the Second Edition can be found at http://www.oreilly.com/openbook/motif/. A description of the modifications is contained in the Preface to the Third Edition, which has become the Open Source Edition.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly & Associates, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

Published by:

IST IMPERIAL SOFTWARE TECHNOLOGY

Imperial Software Technology Limited
Kings Court
185 Kings Road
Reading
Berkshire RG1 4EX
Tel: +44 118 958 7055
Fax: +44 118 958 9005
email: sales@ist.co.uk
URL: http://www.ist.co.uk

Contents

Preface . . . . . **xi**

1. Introduction to Motif **1**

Basic User Interface Concepts 2

What Is Motif? 3

Designing User Interfaces 6

2. The Motif Programming Model **11**

Basic X Toolkit Terminology and Concepts 11

The Xm and Xt Libraries 14

Programming With Xt and Motif 16

Summary 39

3. Overview of the Motif Toolkit **41**

The Motif Style 41

Application Controls 45

Application Layout 53

Putting Together a Complete Application 65

Changes in Motif 2.1 85

Summary 95

4. The Main Window **97**

Creating a MainWindow 98

The MenuBar 103
The Command and Message Areas 118
Using Resources 123
Summary 124
Exercises 125

5. Introduction to Dialogs 127
The Purpose of Dialogs 128
The Anatomy of a Dialog 131
Creating Motif Dialogs 133
Dialog Resources 143
Dialog Callback Routines 147
Piercing the Dialog Abstraction 151
Dialog Modality 156
Summary 167

6. Selection Dialogs 169
Types of SelectionDialogs 169
SelectionDialogs 170
PromptDialogs 177
The Command Widget 180
FileSelectionDialogs 181
Summary 193

7. Custom Dialogs 195
Modifying Motif Dialogs 195
Designing New Dialogs 203
Building a Dialog 208
Generalizing the Action Area 221
Using a TopLevelShell for a Dialog 227

Positioning Dialogs 229
Summary 231

8. Manager Widgets 233
Types of Manager Widgets 233
Creating Manager Widgets 235
The BulletinBoard Widget 237
The Form Widget 243
The RowColumn Widget 261
The Frame Widget 271
The PanedWindow Widget 275
Keyboard Traversal 283
Summary 293

9. Containers and IconGadgets 295
Creating a Container 299
Creating IconGadgets 299
Container Resources 299
IconGadget Resources 304
Container Constraints 305
Container Callbacks 310
Container Functions 316
Summary 318
Exercises 318

10. ScrolledWindows and ScrollBars 319
The ScrolledWindow Design Model 319
Creating a ScrolledWindow 323
Working With ScrollBars 330
Implementing True Application-defined Scrolling 342

Working With Keyboard Traversal in ScrolledWindows . 356
Summary 358
Exercises 358

11. The DrawingArea Widget 361
Creating a DrawingArea Widget 362
Using DrawingArea Callback Functions 363
Using Translations on a DrawingArea 371
Using Color in a DrawingArea 378
Summary 383
Exercises 384

12. Labels and Buttons 387
Labels 388
PushButtons 400
ToggleButtons 406
ArrowButtons 421
DrawnButtons 427
Summary 430
Exercise 431

13. The List Widget 433
Creating a List Widget 434
Using ScrolledLists 437
Manipulating Items 439
Positioning the List 452
Navigating the List 453
List Callback Routines 454
Summary 460
Exercises 460

14. The ComboBox Widget 463

Creating a ComboBox 464

ComboBox Resources 467

ComboBox Functions 469

ComboBox Callbacks 471

Summary 473

Exercises 473

15. The SpinBox and SimpleSpinBox Widgets 475

Creating a SimpleSpinBox 477

Creating a SpinBox 484

SpinBox and SimpleSpinBox Resources 488

SpinBox and SimpleSpinBox Callbacks 490

Summary 495

Exercises 496

16. The Scale Widget 497

Creating a Scale Widget 498

Scale Values 500

Scale Orientation and Movement 501

Scale Resources 502

Scale Callbacks 503

Scale Tick Marks 506

Summary 508

17. The Notebook Widget 509

Creating a Notebook 511

Notebook Resources 516

Notebook Constraints 519

Notebook Callbacks 520

Notebook Functions 521
Summary 522

18. Text Widgets 525
Interacting With Text Widgets 527
Text Widget Basics 530
Text Clipboard Functions 551
A Text Editor 556
Text Callbacks 565
Text Widget Internationalization 580
Summary 587
Exercises 587

19. Menus. 589
Menu Types 589
Creating Simple Menus 592
Designing Menu Systems 603
General Menu Creation Techniques 615
Summary 636
Exercises 636

20. Interacting With the Window Manager 637
Interclient Communication 638
Shell Resources 639
VendorShell Resources 647
Handling Window Manager Messages 650
Session Management 655
Customized Protocols 667
Summary 671
Exercises 671

21. The Clipboard 673

Simple Clipboard Copy and Retrieval 675

Copy by Name 683

Clipboard Data Formats 689

The Primary Selection and the Clipboard 692

Implementation Issues 694

Summary 696

22. Drag and Drop 697

Using Drag and Drop 697

The Drag and Drop Model 699

Customizing Built-in Drag and Drop 712

Working With Drag Sources 719

Working With Drop Sites 736

Summary 755

23. The Uniform Transfer Model 757

Overview 758

Exporting the Data 759

Requesting the Data Format 763

Importing the Data 766

Batched Data Transfer 769

An Example 769

Summary 775

24. Render Tables 777

Renditions 778

Render Tables 781

Tab Lists 786

An Example 792

Render Tables and Resource Files 797
Missing Fonts and Renditions 799
Summary 802

25. Compound Strings 803
Internationalized Text Output 803
Creating Compound Strings 805
Manipulating Compound Strings 819
Parse Tables 824
Rendering Compound Strings 839
Summary 841

26. Signal Handling 843
Handling Signals in X11R5 844
Handling Signals in X11R6 855
Summary 859

27. Advanced Dialog Programming 861
Help Systems 861
Working Dialogs 870
Dynamic Message Symbols 887
Summary 892

A. Additional Example Programs 893
A Bitmap Display Utility 893
A Memo Calendar 905

Index 913

Preface

By convention, a preface describes the book itself, while the introduction describes the subject matter. You should read through the preface to get an idea of how the book is organized, the conventions it follows, and so on.

This book describes how to write applications using the Motif toolkit from the Open Software Foundation (OSF). The Motif toolkit is based on the X Toolkit Intrinsics (Xt), which is the standard mechanism on which many of the toolkits written for the X Window System are based. Xt provides a library of user-interface objects called *widgets* and gadgets, which provide a convenient interface for creating and manipulating X windows, colormaps, events, and other cosmetic attributes of the display. In short, widgets can be thought of as building blocks that the programmer uses to construct a complete application.

However, the widgets that Xt provides are generic in nature and impose no user-interface policy whatsoever. That is the job of a user-interface toolkit such as Motif. Motif provides a complete set of widgets designed to implement the application look and feel specified in the *Motif Style Guide* and the *Motif Application Environment Specification*.

The book provides a complete programmer's guide to the Motif toolkit. While the OSF/Motif toolkit is based on Xt, the focus of the book is on Motif itself, not on the Intrinsics. Detailed information about Xt is provided by Volume 4, and references are made to that volume throughout the course of this book. You are not required to have Volume 4 in order to use this book effectively, as the books are not companion volumes, but complementary ones. However, truly robust applications require a depth of knowledge about Xt and Xlib, the layer on which Xt itself is based, that is not addressed in this book alone. We never leave you completely in the dark about Xt or Xlib functions that we use or reference, but you won't learn everything there is to know about them through this particular volume.

This book covers Motif 2.1, which is the latest major release of the Motif toolkit. Motif 2.1 is based on Release 6 of the Xlib and Xt specifications (X11R6). This release of Motif provides many new features, as well as a number of enhancements to existing functionality. All of the changes in Motif 2.1 are summarized in Chapter 3, which provides references to other sections that describe the changes in more detail.

The Plot

There are several plots and subplots in this book and the stories told are intertwined. Our primary goal is to help you learn about the Motif environment from both the programmer's and the user's perspectives. However, we are talking to you as a programmer, not as a user. We treat the user as a third party who is not with us now. In order to create an application for the user, you sometimes have to assume her role, so at times we may ask you to play such a role to help you think about things from the user's perspective rather than the programmer's.

Each chapter begins by discussing the goals that Motif is trying to achieve using a particular widget or gadget. For example, before we describe how to create a FileSelectionDialog, we introduce the object visually and conceptually, discuss its features and drawbacks, and put you in the role of the user. Once you understand what the user is working with, you should have a better perspective on the task of presenting it to her.

The next subplot is that of application design. Many design concepts transcend the graphical user interface (GUI) and are common to all programs that interact with users. You could even interpret this book as a programmer's guide that happens to use Motif as an example. As you read the material, you should stop and think about how you might approach a particular interface method if you were using another toolkit instead of Motif. A wild concept, perhaps, but this approach is the key to better application design and to toolkit independence. If Motif changes in a later release, or if you decide to port your application to another toolkit or even another windowing system, the more generalized your code is, the easier it will be to bring it into a new realm successfully.

The last story we are telling is that of general programming technique. By providing you with examples of good programming habits, styles, and usages, we hope to propagate a programming methodology that has proven to be successful over the years. These techniques have been applied to applications that have been ported to multiple architectures and operating systems. As an added bonus, we have thrown in a number of interesting programming tricks. No, these are not hacks, but conveniences that are particular to C, to UNIX, or even to the X Window System. We don't focus on these things, but they are made available to you in passing, so you should have no problem identifying them when they come up.

This book is intended to be used as a programmer's manual, not a reference manual. Volume 6B, contains reference material for all of the Motif library functions and widget classes. We have tried to identify those features of the toolkit that are most important for general discussion, so we do not discuss every aspect of the Motif toolkit in the body of this book.

Any major software development effort, especially in its early stages, has bugs that prevent certain features from being used and the Motif toolkit is no exception. There are some bugs

in the Motif toolkit that have not yet been worked out, but this does not imply that the toolkit is poorly written or riddled with errors. Throughout the book, we try to alert you to any potential problems you may encounter due to bugs. In some cases, there are things that work in Motif, but they are poorly designed, and we don't recommend that you use them. Again, we provide an explanation of what's going on and sometimes describe an alternative solution. There are also some features, resources, and functions available in the toolkit that are not supported by OSF. OSF reserves the right to change anything not publicly documented, so rather than discuss undocumented features, we simply ignore them.

We should also point out that this book is not intended to solve all your problems or answer all your questions concerning Motif or its toolkit. It is not going to spoon feed you by giving you step-by-step instructions on how to achieve a particular task. You are encouraged, and even expected, to experiment on your own with the example applications or, better yet, with your own programs. We want to provide you with discussion and examples that provoke you into asking questions like, "What would happen if I changed this program to do this?" It would be unrealistic to believe that we could address every problem that might come up. Rather than approaching situations using overly specific examples, we discuss them in a generalized way that should be applicable to many different scenarios.

Assumptions

The basic method for creating simple applications in Motif is conceptually simple and straightforward. Even if you only dabble in C, you can probably understand the concepts well enough to do most things. However, unless you have a strong handle on the C programming language, there is an upper limit to what you will be able to do when you try to create a full-featured, functioning application. After all, the user-interface portion of most applications should make up no more than 30-40% of the total code. The functionality of an application is up to you and is not discussed here. Without a strong background with C, or some other structured programming language, you might have a problem keeping up with the material presented here.

This book also assumes that you are familiar with the concepts and architecture of the X Toolkit Intrinsics, which are presented in Volume 4M, and Volume 5. A basic understanding of the X Window System is also useful. For some advanced topics, the reader may need to consult Volume 1, and Volume 2.

How This Book Is Organized

While this book attempts to serve the widest possible audience, that does not imply that the material is so simple that it is only useful to novice programmers. In fact, this book can be considered an advanced programmer's handbook, since in many places, it assumes a fairly sophisticated knowledge of many features of the X Window System.

Each chapter is organized so that it gets more demanding as you read through it. Each chapter begins with a short introduction to the particular Motif element that is the subject of the chapter. The basic mechanics involved in creating and manipulating the object are addressed next, followed by the resources and other configurable aspects of the object. If there is any advanced material about the object, it is presented at the end of the chapter. Many chapters also include exercises that suggest how the material can be adapted for uses not discussed explicitly in the text.

While the chapters may be read sequentially, it is certainly not required or expected that you do so. As you will soon discover, there are many circular dependencies that justify skipping around between chapters. Since there is no organization that would eliminate this problem, the material is not organized so that you "learn as you go." Instead, we organized the material in a top-down manner, starting with several chapters that provide an introduction to the Motif look and feel, followed by chapters organized on a widget-by-widget basis. The higher-level manager widgets are discussed first, followed by the primitive widgets and gadgets. Advanced material is positioned at the end of the book, since the details are not of paramount importance to the earlier material.

In short, everything is used everywhere. Starting at the beginning, however, means that we won't necessarily assume you know about the material that is referenced in later chapters. On the other hand, the later chapters may make the assumption that you are aware of material in earlier chapters.

The book is broken down into twenty seven chapters and one appendix as follows:

Chapter 1

Introduction to Motif answers the question "Why Motif?" and suggests some of the complexities that the programmer has to master in order to make an application easy to use.

Chapter 2

The Motif Programming Model teaches the fundamentals of Motif by example. It presents a simple "Hello, World" program that shows the structure and style common to all Motif programs. Much of this material is already covered in detail in Volume 4M, so the chapter can be read as a refresher, or a light introduction for those who haven't read the earlier book. The chapter references Volume 4 and Volume 1, to point out areas that the programmer needs to understand before progressing with Motif.

Chapter 3

Overview of the Motif Toolkit explains what is involved in creating a real application. The chapter discusses the arrangement of primitive widgets in an interface, the use of dialog boxes and menus, and the relationship between an application and the window manager. The chapter also describes all of the changes in Release 2.1 of the Motif toolkit. After reading this chapter, the programmer should have a solid overview of

Motif application programming and be able to read the remaining chapters in any order.

Chapter 4

The Main Window describes the Motif MainWindow widget, which can be used to frame many types of applications. The MainWindow is a manager widget that provides a MenuBar, a scrollable work area, and various other optional display and control areas.

Chapter 5

Introduction to Dialogs describes the fundamental concepts that underly all Motif dialogs. It provides a foundation for the more advanced material in the following chapters. In the course of the introduction, this chapter also provides details on Motif's predefined MessageDialog classes.

Chapter 6

Selection Dialogs presents the more complex Motif-supplied dialogs for displaying selectable items, such as lists of files or commands, to the user.

Chapter 7

Custom Dialogs describes how to create new dialog types, either by customizing Motif dialogs or by creating entirely new dialogs.

Chapter 8

Manager Widgets provides detailed descriptions of the various classes of Motif manager widgets. Useful examples explore the various methods of positioning components in Form and RowColumn widgets.

Chapter 9

Containers and IconGadgets describes two components which are new to Motif 2. These were designed to work together in order to provide a more graphical presentation of the front end of the application than the older Main Window. The IconGadget pictorially represents application objects; the Container lays them out in a variety of styles, including Tablular, Grid, and Tree formats. The layout can be changed dynamically: the Container and IconGadget combination approximates to a Model-View-Controller (MVC) system for the Motif widget set.

Chapter 10

ScrolledWindows and ScrollBars describes the ins and outs of scrolling, with particular attention to application-defined scrolling, which is often required when the simple scrolling provided by the ScrolledWindow widget is insufficient.

Chapter 11

The DrawingArea Widget describes the Motif DrawingArea widget, which provides a canvas for interactive drawing. The chapter simply highlights, with numerous code examples, the difficulties that may be encountered when working with this widget, rather

than trying to teach Xlib drawing techniques. Some knowledge of Xlib is assumed; we direct the reader to Volume 1, for additional information.

Chapter 12

Labels and Buttons provides an in-depth look at labels and buttons, the most commonly-used primitive widgets. The chapter discusses the Label, PushButton, ToggleButton, ArrowButton, and DrawnButton widget classes.

Chapter 13

The List Widget describes yet another method for the user to exert control over an application. A List widget displays a group of items from which the user can make a selection.

Chapter 14

The ComboBox Widget describes another component which is new in Motif 2. The ComboBox combines List display with Text input, although the List can be hidden until required. The widget therefore maximizes user convenience using the minimal of screen space.

Chapter 15

The SpinBox and SimpleSpinBox Widgets are also new in Motif 2. Similar in concept to the ComboBox, the widgets allow the user to choose from a set of values, and the current choice is presented through a TextField. The difference is that the user changes the current choice not by selecting from a List, but by rotating through the set of available values using ArrowButtons provided for the purpose.

Chapter 16

The Scale Widget describes how to use the Scale to display a range of values.

Chapter 17

The Notebook Widget describes a component which provides page or tab manager functionality to the Motif 2 toolkit. The programmer adds children to the Notebook, only one of which is visible at any given time. The user can select between pages using Tabs (PushButtons) aligned along the edges of the Notebook, or by selecting the required page number from a SpinBox which the Notebook creates automatically.

Chapter 18

Text Widgets explains how the Text and TextField widgets can be used to provide text entry in an application, from a single data-entry field to a full-fledged text editor. Special attention is paid to problems such as how to mask or convert data input by the user so as to control its format. The chapter also discusses the internationalization features of the widgets provided in Motif 1.2.

Chapter 19

Menus describes the menus provided by the Motif toolkit.The chapter examines how menus are created and presents some generalized menu creation routines.

Chapter 20

Interacting With the Window Manager provides additional information on the relationship between an application and the Motif Window Manager (*mwm*). It discusses the shell widget resources and window manager protocols that can be used to communicate with the window manager. It also discusses various CDE desktop aspects of the window manager interaction.

Chapter 21

The Clipboard describes a way for the application to interact with other applications. Data is placed on the clipboard, where it can be accessed by other windows on the desktop, regardless of the applications with which they are associated.

Chapter 22

Drag and Drop presents the drag and drop mechanism for transferring data that is provided in Motif 1.2. The chapter describes the built-in drag and drop features of the Motif toolkit and provides examples of adding drag and drop functionality to an application.

Chapter 23

The Uniform Transfer Model describes the scheme introduced in Motif 2 which allows the programmer to handle the various data transfer operations supported by Motif (Primary and Secondary Selections, the Clipboard, Drag-and-Drop) using a single programming interface.

Chapter 24

Render Tables describes the Motif 2 mechanisms which control the way in which compound strings are displayed by the toolkit. In Motif 2, strings which appear in widgets can be multi-colored, multi-font, and laid out in a multi-column arrangement. The coloration, font, and tabular information is held separately from the string which is to be drawn in the form of a render table.

Chapter 25

Compound Strings describes Motif's technology for encoding font and directional information in the strings that are used by almost all Motif widgets. It discusses how to use compound strings in an internationalized application.

Chapter 26

Signal Handling presents the problems that can be encountered when mixing UNIX signals with X applications. It explains how signals work and why they can wreak such havoc with X. It presents the new features of X11R6 which are expressly designed to handle this problem.

Chapter 27

Advanced Dialog Programming describes the issues involved in creating multi-stage help systems, using WorkingDialogs that allow the user to interrupt long-running tasks, and dynamically changing the pixmaps displayed in a dialog.

Appendix A

Additional Example Programs provides several additional examples that illustrate techniques not discussed in the body of the book.

Related Documents

The following books on the X Window System are available from O'Reilly & Associates, Inc.:

Volume Zero	*X Protocol Reference Manual*
Volume One	*Xlib Programming Manual*
Volume Two	*Xlib Reference Manual*
Volume Three	*X Window System User's Guide, Motif Edition*
Volume Four	*X Toolkit Intrinsics Programming Manual, Motif Edition*
Volume Five	*X Toolkit Intrinsics Reference Manual*
Volume Six A	*Motif Programming Manual*
Volume Six B	*Motif Reference Manual*
Volume Seven	*XView Programming Manual* with accompanying reference volume.
Volume Eight	*X Window System Administrator's Guide*
PHIGS Programming Manual	
PHIGS Reference Manual	
PEXlib Programming Manual	
PEXlib Reference Manual	
Quick Reference	*The X Window System in a Nutshell*
Programming Supplement for Release 6 of the X Window System	

Conventions Used in This Book

Italic is used for:

- UNIX path names, filenames, program names, user command names, options for user commands, and variable expressions in syntax sections.
- New terms where they are defined.

`Typewriter Font` is used for:

- Anything that would be typed verbatim into code, such as examples of source code and text on the screen.
- Variables, data structures (and fields), symbols (defined constants and bit flags), functions, macros, and a general assortment of anything relating to the C programming language.
- All functions relating to Motif, Xt, and Xlib.
- Names of subroutines in example programs.

`Italic Typewriter Font` is used for:

- Arguments to functions, since they could be typed in code as shown but are arbitrary names that could be changed.

Boldface is used for:

- Names of buttons and menus.

Obtaining Motif

Motif sources can be obtained from a number of locations, although the primary reference site is:

```
http://www.opengroup.org/motif
```

These sources are known as Open Motif, and the use of such sources in applications is restricted to Open Source platforms.

Alternatively, if your hardware vendor is an OSF member, they may be able to provide Motif binaries for your machine. Various independent vendors also provide binaries for some machines.Source licenses must be obtained directly from OSF:

OSF Direct
Open Software Foundation
11 Cambridge Center
Cambridge, MA 02142
USA
+1 617 621-7300
Internet: direct@osf.org

Obtaining the Example Programs

The example programs in this book are available electronically in a number of ways: by FTP, FTPMAIL, BITFTP, and UUCP. The cheapest, fastest, and easiest ways are listed first. If you read from the top down, the first one that works for you is probably the best.

Use FTP if you are directly on the Internet. Use FTPMAIL if you are not on the Internet but can send and receive electronic mail to internet sites (this includes CompuServe users). Use BITFTP if you send electronic mail via BITNET.Use UUCP if none of the above works.

Versions of the example programs for Motif 2.1, Motif 1.2 and Motif 1.1 are available electronically. If you want the Motif 2.1 version, use the filename *examples21.tar.Z*, as shown in the sample sessions below. The filename for the Motif 1.2 version is *examples12.tar.Z*.

FTP

To use FTP, you need a machine with direct access to the Internet. A sample session is shown, with what you should type in boldface.

```
    % ftp ftp.uu.net
    Connected to ftp.uu.net.220 FTP server (Version 6.21 Tue Mar 10 22:09:55 EST
1992) ready.
    Name (ftp.uu.net:paula): anonymous
    331 Guest login ok, send domain style e-mail address as password.
    Password: paula@ora.com (use your user name and host here)
    230 Guest login ok, access restrictions apply.
    ftp> cd /published/oreilly/xbook/motif
    250 CWD command successful.
    ftp> binary (Very important! You must specify binary transfer for compressed files.)
    200 Type set to I.
    ftp> get examples12.tar.Z
    200 PORT command successful.
    150 Opening BINARY mode data connection for examples12.tar.Z
    226 Transfer complete.
    ftp> quit
    221 Goodbye.
    %
```

If the file is a compressed tar archive, extract the files from the archive by typing:

```
    % zcat examples12.tar.Z | tar xf -
```

System V systems require the following *tar* command instead:

```
    % zcat examples12.tar.Z | tar xof -
```

If *zcat* is not available on your system, use separate *uncompress* and *tar* commands.

FTPMAIL

FTPMAIL is a mail server available to anyone who can send electronic mail to and receive it from Internet sites. This includes any company or service provider that allows email connections to the Internet. Here's how you do it.

You send mail to *ftpmail@online.ora.com*. In the message body, give the FTP commands you want to run. The server will run anonymous FTP for you and mail the files back to you. To get a complete help file, send a message with no subject and the single word "help" in the body. The following is an example mail session that should get you the examples. This command sends you a listing of the files in the selected directory, and the requested example files. The listing is useful if there's a later version of the examples you're interested in.

```
% mail ftpmail@online.ora.com
Subject:reply paula@ora.com                (where you want files mailed)
opencd /published/oreilly/xbook/motif
dirmode
binary
uuencode
get examples12.tar.Z
quit
%
```

A signature at the end of the message is acceptable as long as it appears after "quit."

All retrieved files will be split into 60KB chunks and mailed to you. You then remove the mail headers and concatenate them into one file, and then *uudecode* or *atob* it. Once you've got the desired file, follow the directions under FTP to extract the files from the archive.

VMS, DOS, and Mac versions of *uudecode*, *atob*, *uncompress*, and *tar* are available.

BITFTP

BITFTP is a mail server for BITNET users. You send it electronic mail messages requesting files, and it sends you back the files by electronic mail. BITFTP currently serves only users who send it mail from nodes that are directly on BITNET, EARN, or NetNorth. BITFTP is a public service of Princeton University. Here's how it works.

To use BITFTP, send mail containing your ftp commands to *BITFTP@PUCC*. For a complete help file, send HELP as the message body.

The following is the message body you should send to BITFTP:

```
FTP  ftp.uu.net  NETDATA
USER  anonymous
PASS your Internet email address (not your bitnet address)
CD  /published/oreilly/xbook/motif
DIR
BINARYG
ET  examples12.tar.Z
QUIT
```

Once you've got the desired file, follow the directions under FTP to extract the files from the archive. Since you are probably not on a UNIX system, you may need to get versions

of *uudecode*, *uncompress*, *atob*, and *tar* for your system. VMS, DOS, and Mac versions are available. The VMS versions are on *gatekeeper.dec.com* in */archive/pub/VMS*.

Questions about BITFTP can be directed to Melinda Varian, *MAINT@PUCC* on BITNET.

UUCP

UUCP is standard on virtually all UNIX systems, and is available for IBM-compatible PCs and Apple Macintoshes. The examples are available by UUCP via modem from UUNET; UUNET's connect-time charges apply.

You can get the examples from UUNET whether you have an account or not. If you or your company has an account with UUNET, you will have a system with a direct UUCP connection to UUNET. Find that system, and type:

```
uucp uunet\!~/published/oreilly/xbook/motif/examples12.tar.Z
        yourhost\!~/yourname/
```

The backslashes can be omitted if you use the Bourne shell (*sh*) instead of *csh*. The file should appear some time later (up to a day or more) in the directory */usr/spool/uucppublic/yourname*. If you don't have an account but would like one so that you can get electronic mail, then contact UUNET at 703-204-8000.

It's a good idea to get the file */published/oreilly/xbook/motif/ls-lR.Z* as a short test file containing the filenames and sizes of all the files in the directory.

Once you've got the desired file, follow the directions under FTP to extract the files from the archive.

Copyright

The example programs are written by Dan Heller, Paula Ferguson, Antony Fountain, and Jeremy Huxtable for the *Motif Programming Manual*, Copyright 1994 O'Reilly &Associates, Inc. Permission to use, copy, and modify these programs without restriction is hereby granted, as long as this copyright notice appears in each copy of the program source code.

For the purposes of making the book easier to read, the above copyright notice does not appear in the program examples. However, the copyright does exist in the electronic form of the programs available on the Internet.

Compiling the Example Programs

Once you have the examples and you've unpacked the archive as described above, you're ready to compile them. The easiest way is to use *imake*, a program supplied with the X11 distribution that generates proper Makefiles on a wide variety of systems. *imake* uses

configuration files called Imakefiles that are included with the examples. If you have *imake*, you should go to the top-level directory containing the examples, and type:

```
% xmkmf
% make Makefiles
% make
```

The examples all have the same application class for purposes of the app-defaults file. The class name is “Demos” and the app-defaults file (*Demos*) in the main examples directory should be placed in */usr/X11R6/lib/app-defaults/Demos* on a UNIX system. If you can’t write to that directory, or if your normal X11 directory tree is installed elsewhere, you should set the environment variable XAPPLRESDIR to the directory where you installed the examples.

Acknowledgments

Third Edition. The current edition of this book was updated to cover Motif 2.1 by Antony Fountain and Jeremy Huxtable, both of Imperial Software Technology. Jerry originally wrote most of the Motif 2.1 sample programs which appear in the book. He also wrote the utility *Snap*, which was used to recreate all the screen shots for this manual. Originally we intended to include this in the Appendix as a sample application, but space forbade this. For myself, I simply made sure that the examples were non-deprecated. The text, however, is mine, and I accept the blame for everything.

Special thanks go to the people who worked on the production of this book. The final form of this book is the work of the staff at O’Reilly & Associates. I would like to thank all of them for allowing me to take on this project; a special thanks to Paula Ferguson, without whom the manuscript would never have reached the printer. The authors would like to thank all at IST for their patience and support. A special thanks must go to Denise Buckler, John Bishop, Andy Davies, Simon Davies, Ruth Lambert, Andy and Tricia Lovell, Graham Salisbury, and Rob Snell, who all cheerfully assisted in the onerous task of proof reading. Thanks to Alan Sandell for keeping the printer working. A big thanks to Andy Lovell, Neil Smyth, and Derek Lambert for their patience and support when I could have been working on company matters. And last but definitely not least, a very special thank you to my wife Emma for keeping the home fires burning.

Antony J. Fountain

Second Edition. The second edition of this book was updated to cover Motif 1.2,including drag and drop and internationalization, by Paula Ferguson. Dave Brennan, of HaL Computer Systems, took on the unenviable task of learning everything he could about UIL and Mrm, in order to write the UIL programming material for this edition. He did a great job of covering a complex subject.

Adrian Nye deserves recognition for allowing me to work on this project, when I’m sure that he had other projects he would have liked to send my way. I don’t think either one of

us had any idea how involved this update project would become. He also provided editorial support that helped keep me on track in the final stages of the work on the book.

The other writers at O'Reilly &Associates in Cambridge, Valerie Quercia and Linda Mui, provided support that kept me sane while I was working on the book. Their willingness to listen and offer advice is greatly appreciated. Extra gratitude goes to Valerie Quercia for her help with the screen dumps for the book.

David Flanagan deserves credit for always being willing to answer my questions about the technical details of Motif and X. Douglas Rand, Scott Meeks, and David Brooks at OSF answered questions and helped review the new material. Daniel Jahn, of SAS Institute, Inc. , also provided valuable review comments for this edition.

Special thanks go to the people who worked on the production of this book. The final form of this book is the work of the staff at O'Reilly &Associates. The authors would like to thank Chris Reilly for the figures, Donna Woonteiler, Chris Tong, and Ellie Cutler for indexing, Lenny Muellner for tools support, and Stephen Spainhour, Clairemarie Fisher O'Leary, Kismet McDonough, and Eileen Kramer for copy editing and production of the final copy. Thanks also to Donna Woonteiler for her patience in helping me understand the production process.

Finally, I'd like to thank my friends for putting up with me when I kept telling them that I'd be done working non-stop in a month or two. Special thanks to my house mate, Meredith Hunt, who put up with me when I was stressed out and not much fun to live with, and who took care of the cats when I wasn't around. My friends Karen Lewis and Liz Bradley opened their house to me when I needed to escape and be someplace where there are mountains. And thanks to the great people at the Boston Rock Gym, who provided me with a much-needed outlet for climbing the walls.

Despite the efforts of all of these people, the authors alone are responsible for any errors or omissions that remain.

Paula M. Ferguson

First Edition. The first edition of this book took over a year and a half to write and compile from the beginning. But when I look back on the entire effort, and I think about what it takes to do things like this (and other difficult things in life), I realize that what it *really* requires is a state of mind and a mental model that lends itself to seeing the big picture and choosing to do what's necessary to get the job done.

To this, I can only credit one person, Tim O'Reilly, my friend and editor of this book. It's his approach to life, his values, his way of thinking about things, and his talent for expressing them is what has influenced me more than anything else in adopting the kind of mental framework necessary to write a book like this (or to start my company, Z-Code Software, or to do anything I do in life). He never gives me advice when I ask for it, nor does he tell me what to do. Instead, he uses quotes, cites anecdotes, or just describes an

abstract thought that always seems to be appropriate to every situation. In short, he's shown me a way of thinking about things that appreciates the big picture. I take this with me wherever I go, and in whatever I do. Without it, I couldn't have written this book.

Those who worked most closely with me on the project include Irene Jacobson, who dedicated long hours to meticulous editing and support. Her intuition and insistence on proper use of words saved many cuts of Tim O'Reilly's scalpel. David Lewis also gets super-high marks for his excellent feedback, for his technical expertise, and for helping take care of certain Z-Mail ports while I was busy hunched over this computer. More thanks go to the great folks at Z-Code Software, Bart Schaefer and Don Hatch, for not laughing at me when I told people for at least six months that the book would take "just two more weeks now."(I really meant it, too!) Actually, they helped quite a bit with reading nroff'd manuscripts, and by taking care of the business whenever I was at O'Reilly &Associates' offices in "Bahston."

The figures in this book come in two forms: screen dumps and hand-generated figures done by Chris Reilly. What a super job he did--and always on time. And how can I thank Kismet McDonough, Lenny Muellner, Rosanne Wagger, Mike Sierra, Eileen Kramer, and the other production folks at O'Reilly &Associates, who did a wonderful job of copy editing, proofing, page layout, and all the other things that make the difference between a manuscript and a finished book. And that's not all: Ellie Cutler wrote the index. Tony Marotto of Cambridge Computer Associates figured out how to convert our screen dumps into PostScript files and how to scale screen dumps without the moire and plaid patterns you see in many books. He used Jeff Poskanzer's *pmbplus* to convert *xwd* dumps to *gif* format, and then wrote a set of image-processing programs that shift and enhance the tones. Daniel Gilly took on the enormous job of developing the reference appendices when it became clear that I wouldn't have time.

Enthusiastic applause goes to Libby Hanna (do I get a *real* official OSF/Motif decoder ring now!!??), David Brooks, Scott Meeks, Susan Thompson, Carl Scholz, Benjamin Ellsworth, and the entire cast at OSF in Cambridge for their support. And, of course, *everyone* on the motif-talk mailing list.(I wish I could remember all your names!)

People I can't forget: Bill "Rock" Petro, Akkana, Mike Harrigan at NCD for the terminal, Danny Backx at BIM (sorry I didn't get you any review copies!), John Harkin, and certain folks at Sun that I'd love to mention, but I can't because they're into that *OL-thang* and they wouldn't want to be associated with the *M-word*, Jordan Hayes, Paula Ferguson, and Kee Hinckley (just because he's cool).Also thanks to Ralph Swick and Donna Converse at the X Consortium for being somewhat patient with me.

Added thanks to Lynn Vaughn at CNN for keeping me informed about what's going on in the world, since I have no time to look out the window; to Short Attention-Span Theatre, for keeping me amused; and to Yogurt World, for keeping me fed.

This book was written using a Sun workstation, the *vi* editor (for which I guess I ought to thank Bill Joy), SoftQuad's *sqtroff*, X11R4 and various versions of Motif (1.0 through 1.1.3).

For catching and reporting errors that have been fixed in the second printing, I'd like to thank Akkana, Wayne Robertz, Glen Shute, Scott Strool, Trevor Taylor, Peter Wagner, Andrew Wason, Tim Weinrich, and Bill Wohler.

Dan Heller

We'd Like to Hear From You

We have tested and verified all of the information in this book to the best of our ability, but you may find that features have changed (or even that we have made mistakes!). Please let us know about any errors you find, as well as your suggestions for future editions, by writing:

O'Reilly & Associates, Inc.
103 Morris Street, Suite A
Sebastopol, CA 95472
1-800-998-9938 (in the US or Canada)
1-707-829-0515 (international/local)
1-707-829-0104 (FAX)

You can also send us messages electronically. To be put on the mailing list or request a catalog, send email to:

info@ora.com (via the Internet)
uunet!ora!info (via UUCP)

To ask technical questions or comment on the book, send email to:

bookquestions@ora.com (via the Internet)

In this chapter:

- *Basic User Interface Concepts*
- *What Is Motif?*
- *Designing User Interfaces*

1

Introduction to Motif

So many computers, so many operating systems, so many toolkits*.

Developing an application used to be a simple choice, depending upon whether we targeted the application for the Microsoft world, for UNIX, or for the Apple MacIntosh. Each had its own distinct toolkit interface. If you wanted to write your application cross-platform, you had to encapsulate the functionality through a set of common C++ classes, each of which would have a separate internal implementation for each operating environment. Yet for the myriad of UNIX platforms, all you had to do was write Motif.

But now there is Linux, and Java, and GTK+ and Qt, and a host of other considerations. Life never was so complex, and all we want to do is write an application. In many ways, the task of the application programmer is now to write the application with the greatest degree of toolkit independence that can be achieved, whether through a client-server architecture to separate the interface from the back-end processing, or still through encapsulation techniques which hide the underlying toolkit from the higher levels. The toolkit ought in principle to be irrelevant; in practice this is not possible: there has to be a windowing toolkit somewhere at the bottom level, and that toolkit must be appropriate to the target operating system and environment. A emulator or cross-platform common toolkit never quite succeeds in providing the requisite functionality in all levels of detail; for some platforms, if it is not in the native toolkit, it simply isn't ported; this is particularly true of Windows environments, where using anything other than MFC is simply wrong in principle. MFC is *the* native windowing environment for Windows. It just is.

So why Motif? Because it remains what it has long been: the common native windowing toolkit for all the UNIX platforms, fully supported by all the major operating system vendors. It is still the only truly industrial strength toolkit capable of supporting large scale and long term projects. Everything else is tainted: it isn't ready or fully functionally complete, or the functional specification changes in a non-backwards-compatible manner per release, or there are performance issues. Perhaps it doesn't truly port across UNIX systems, or it isn't fully ICCCM compliant with software written in any other toolkit on the desktop, or there are political battles as various groups try to control the specification for

* Infandum, regina, iubes renouare dolorem...

their own purposes. Indeed it may matter very much whose version of the toolkit you have managed to acquire, or if the toolkit is open sourced so you have no idea who is going to stick their untrusted fingers into it at any time. So many problems with each choice you make. With Motif, you know where you are: its stable, its robust, its professionally supported, and it all works.

And yet whatever the toolkit you choose to write your application in, the design goals ought to remain precisely the same. You should be trying to present the application to the user in the most consistent, simple to use, and simple to understand manner of which you are humanly capable. Interfaces consist of basic controls and layout managers, irrespective of the language it is written in. In this respect, Motif, despite its long history, remains as fully capable as any basic toolkit. It is true that Motif 1.2 lacked some of the features which we now expect from a windowing toolkit - the philosophy of design moves on with time - but these issues are addressed in the 2.1 version of the toolkit. The ComboBoxes, SpinBoxes, Tree and Grid layouts are all there. Where Motif differs from other toolkits is the strength of the component inter-operability. The Motif toolkit is not just a collection of controls written in a particular language: everything, but everything, is designed to work with everything else, whether it be navigation between controls, or inter-object data transfer, or the sharing of style resources throughout the control hierarchy. And because it is based on top of the X interface toolkit, Xt, it will work with the vast range of third party components and add-ons which are available to the X world. In many ways, it is precisely these aspects which are stronger in the Motif 2.1 toolkit than ever before.

Much of this chapter can be read as a general introduction to graphical user interface toolkits; the concepts which we present are not specific to Motif, or indeed to any other windowing toolkit. The ideas presented here should be general enough to read in a toolkit independent manner; how you implement those ideas using the specific Motif toolkit is covered in subsequent chapters of this book.

Basic User Interface Concepts

Whether you are the designer of the software or an engineer responsible for implementing someone else’s design, there are some basic principles that will benefit you in your work. Let’s begin with the basics:

- All applications running on a user’s workstation should have a consistent interface design. Programs that deviate from the expected design will almost assuredly confuse the user even if the changes were intended for the user’s benefit. Chances are also high that the user will not want to use the questionable software again.
- Users rely on rote memory; they will remember seemingly complicated interface interaction techniques provided that the functions they perform are useful and are invoked frequently. There is a limit, however, to how much users want to remember. It is important that essential or frequently used functions follow memorable patterns.

- Novice users will probably not want to customize or alter their applications in any way. If they do, the available methods must be as easy and painless as possible.
- Users with more experience most certainly might want to customize the application in all sorts of ways: the greater degree of customization which the application allows, the better.

One of the first things the hard-core X programmer learns is that “the user is always right; if he wants to customize his interface, by God you had better let him.”

This principle is absolutely correct. Unfortunately, many early X applications carry it too far and end up “spineless.” Many such programs actually require the user to make certain customizations in order for the program to be usable or attractive. For some programs, the problem worsens if unreasonable customization settings are given, since there is no sanity-checking for unreasonable configurations.

So far, such customization issues have not become over-problematic because UNIX and X applications are used almost exclusively by technical people who understand the environment and know how to work within it. But it is important to consider users who know absolutely nothing about computers and who don’t want to - they are only using your software because they have to.

The customization issue has partly been addressed in any case by environments like CDE, or the Schemes mechanisms on SGI platforms: users can choose from (and add to) a range of preset styles which will affect all applications on the desktop. Part of the work of the engineer is now to ensure that the application participates in desktop schemes of this kind, so that the user can customize in a general way rather than having to configure the style of every application individually.

What Is Motif?

So, back to Motif. What is it and how can it help you solve your user-interface design goals? To start, Motif is a set of guidelines that specifies how a user interface for graphical computers should *look and feel*. This term describes how an application appears on the screen (the look) and how the user interacts with it (the feel). Look and Feel is not something specific to Motif; all windowing toolkits should present a standardized internally-consistent methodology so that the user is comfortable using the controls which the application presents. Specific toolkits, however, have distinct look and feel, although since some toolkits share a common design philosophy there is a cross-over so that users familiar with one platform are not necessarily naked when presented with an alternative. This will be made clear in the paragraphs which follow. Firstly, let us look at a Motif application.

Figure 1-1 shows a Motif application, used for taking snapshots of windows or capturing areas of the screen.

Figure 1-1 : A Motif Application

The user interacts with the application by typing at the keyboard, and by clicking, selecting, and dragging various graphic elements of the application with the mouse. For example, any application window can be moved on the screen by moving the pointer to the top of the window's frame (the title bar), pressing and holding down a button on the mouse, and dragging the window to a new location. The window can be made larger or smaller by pressing a mouse button on any of the resize corners and dragging.

Most applications sport buttons that can be clicked with the mouse to initiate application actions. Motif uses highlighting and shadowing to make buttons, and other components, look three-dimensional. When a button is clicked on, it actually appears to be pressed in and released.

A row of buttons across the top of most applications forms a *menu bar*. Clicking on any of the titles in the menu bar pops up a menu of additional buttons. Buttons can also be arranged in palettes that are always visible on the screen. When a button is clicked, the application can take immediate action or it can pop up an additional window called a *dialog box*. A dialog box can ask the user for more information or present additional options.

This style of application interaction isn't new to most people, since the Apple MacIntosh popularized it years ago. What is different about Motif is that the graphical user interface specification is designed to be independent of the computer on which the application is running.

Motif was designed by the Open Software Foundation (OSF), a non-profit consortium of companies such as Hewlett-Packard, Digital, IBM, and dozens of other corporations. OSF's charter calls for the development of technologies that will enhance inter-operability between computers from different manufacturers. Targeted technologies range from user interfaces to operating systems.

Part of OSF's charter was to choose an appropriate windowing system environment that would enable the technology to exist on as wide a range of computers as possible. It was decided that the OSF/Motif toolkit should be based on the X Window System, a network-based windowing system that has been implemented for UNIX,VMS, DOS, Macintosh, and other operating systems. X provides an extremely flexible foundation for any kind of graphical user interface.

When used properly, the Motif toolkit enables you to produce completely Motif-compliant applications in a relatively short amount of time. At its heart, though, Motif is a specification rather than an implementation. While most Motif applications are implemented using the Motif toolkit provided by OSF, it would be quite possible for an application implemented in a completely different way to comply with the Motif GUI. The specification is captured in two documents: the *Motif Style Guide*, which defines the external look and feel of applications, and the *Application Environment Specification*, which defines the application programmer's interface (API).*

The Motif specifications don't have a whole lot to say about the overall layout of applications. Instead, they focus mainly on the design of the objects that make up a user interface - the menus, buttons, dialog boxes, text entry, and display areas. There are some general rules, but for the most part, the consistency of the user interface relies on the consistent behavior of the objects used to make it up, rather than their precise arrangement.

The Motif specification is broken down into two basic parts:

- The output model describes what the objects on the screen look like. This model includes the shapes of buttons, the use of three-dimensional effects, the use of cursors and bitmaps, and the positioning of windows and subwindows. Although some recommendations are given concerning the use of fonts and other visual features of the desktops, Motif is flexible in most of these recommendations.
- The input model specifies how the user interacts with the elements on the screen.

The key point of the specification is that consistency should be maintained across all applications. Similar user-interface elements should look and act similarly regardless of the application that contains them.

Motif can be used for virtually any application that interacts with a computer user. Programs as conceptually different as a CAD/CAM package or an electronic mail

* Both books have been published for OSF by Prentice-Hall and are available in most technical bookstores.

application still use the same types of user-interface elements. When the user interface is standardized, the user gets more quickly to the point where he is working with the application, rather than just mastering its mechanics.

Those familiar with Microsoft Windows should have little trouble in using a Motif-based application.This is not a coincidence; its user-interface is based on the same principles as Motif. Motif can be seen as a superset of both MS-Windows and Presentation Manager. Even though the others came first, Motif views them as specific implementations of an abstract specification.

The Motif interface was intentionally modelled after IBM's Common User Access (CUA) specification, which defines the interface for OS/2 and Microsoft Windows. The reason for this is that there is a proven business model for profiting from an "open systems" philosophy. As a result, all of the major operating system vendors support Motif as their native graphical interface environment.

You have two options for making applications Motif-compliant. You can write the entire application yourself, and make sure that all your user-interface features conform to the Motif GUI specifications, or you can use a programming toolkit, which is a more realistic option. A toolkit is a collection of pre-written functions that implement all the features and specifications of a particular GUI.

However, a toolkit cannot write an application for you, nor can it enforce good programming techniques. It isn't going to tell you that there are too many objects on the screen or that your use of colors is outrageous. The job of Motif is solely to provide a consistent appearance and behavior for user-interface controls. So, before we jump into the mechanics of the Motif toolkit, let's take a moment longer with the philosophy of graphical user interfaces.

Designing User Interfaces

The principles behind an effective user interface cannot be captured in the specifications for Motif or any other GUI. Even though the Motif toolkit specifies how to create and use its interface elements, there is still quite a bit left unsaid. As the programmer, you must take the responsibility of using those elements effectively and helping the user to be as productive as possible. You must take care to keep things simple for the beginner and, at the same time, not restrict the more experienced user. This task is perhaps the most difficult one facing the programmer in application design.

There is frequently no right or wrong way to design an interface. Good user-interface design is usually a result of years of practice: you throw something at a user, he plays with it, complains, and throws it back at you. Experience will teach you many lessons, although we hope to guide you in the right direction, so that you can avoid many common mistakes and so that the ones that you do make are less painful.

So, rather than having absolute commandments, we rely on heuristics, or rules of thumb. Here is a rough list to start with:

- Keep the interface as simple as possible.
- Make direct connections to real-world objects or concepts.
- If real-world metaphors are not available, improvise.
- Don't restrict functionality to accommodate simplicity.

This list may sound flippant, but it is precisely what makes designing an interface so frustrating. Keeping an interface as simple as possible relies on various other factors, the most basic of which is intuition. The user is working with your application because he wants to solve a particular problem or accomplish a specific task. He is going to be looking for clues to spark that connection between the user interface and the preconceived task in his mind. Strive to make the use of an application obvious by helping the user form a mental mapping between the application and real-world concepts or objects. For example, a calculator program can use buttons and text areas to graphically represent the keypad and the one-line display on a calculator. Most simple calculators have the common digit and arithmetic operator keys; a graphical display can easily mimic this appearance. Other examples include a programmatic interface to a cassette player, telephone, or FAX machine. All of these could have graphical equivalents to their real-world counterparts.

The reason these seemingly obvious examples are successful interface approaches is because they take advantage of the fact that most people are already familiar with their real-life counterparts. But there is another, less obvious quality inherent in those objects: they are simple. The major problem concerning interface design is that not everything is simple. There isn't always a real-world counterpart to use as a crutch. In the most frustrating cases, the concept itself may be simple, but there may not be an obvious way to present the interaction. Of course, once someone thinks of the obvious solution, it seems odd that it could have been difficult in the first place.

Consider the VCR. Conceptually, a VCR is a simple device, yet statistics used to say that 70% of VCR owners don't know how to program one. How many times have you seen the familiar *12:00-AM* flashing in someone's living room? Researchers say that this situation occurs because most VCRs are poorly designed and are "too feature full." They're half-right; the problem is not that they are too feature full, but that the ways to control those features are too complicated. Reducing the capabilities of a VCR isn't going to make it easier to use; it's just going to make it less useful. The problem with VCRs is that their designers focused too much on functionality and not enough on usability.

So, how do you design an interface for a VCR when there is no other object like it? You improvise. Sure, the VCR is a simple device; everyone understands how one is supposed to work, but few people have actually designed one that is easy to use until recently. Maybe you've heard about the new device that, when connected to your VCR, enables you to have

a complete TV program guide displayed on your screen in the bar-graph layout similar to the nightly newspaper listings. All you have to do is point and click on the program you want to record and that's it - you're done. No more buttons to press, levels of features to browse through, dials to adjust or manuals to read. At last, the right interface has been constructed. None of the machine's features have been removed. It's just that they are now organized in an intuitive way and are accessible in an simple manner.

This method for programming VCRs satisfies the last two heuristics. Functionality has not been reduced, yet simplicity has been heightened because a creative person thought of a new way to approach the interface. The lesson here is that no object should be difficult to use no matter how feature full it is or how complex it may seem. You must rely heavily on your intuition and creativity to produce truly innovative interfaces.

Let's return to computer software and how these principles apply to the user-interface design model. The first heuristic is simplicity, which typically involves fewer, rather than more, user-interface elements on the screen. Buttons, popup menus, colors, and fonts should all be used sparingly in an application. Often, the availability of hundreds of colors and font styles along with the attractiveness of a three-dimensional interface compels many application programmers to feel prompted, and even justified, in using all of the bells and whistles. Unfortunately, overuse of these resources quickly fatigues the user and overloads his ability to recognize useful and important information.

Ironically, the potential drawbacks to simplicity are those that are also found in complexity. By oversimplifying an interface, you may introduce ambiguity. If you reduce the number of elements on your screen or make your iconic representations too simple, you may be providing too little information to the user about what a particular interface element is supposed to do. Under-use of visual cues may make an application look bland and uninteresting.

One of Motif's strengths is the degree of configurability that you can pass on to the end user. Colors, fonts, and a wide variety of other resources can be set specifically by the user. You should be aware, however, that once your application ships, its default state is likely to be the interface most people use, no matter how customizable it may be. While it is true that more sophisticated users may customize their environment, you are ultimately in control of how flexible it is. Also, novice users quickly become experts in a well-designed system, so you must not restrict the user from growth.

Simplicity may not always be the goal of a user interface. In some cases, an application may be intentionally complex. Such applications are only supposed to be used by sophisticated users. For example, consider a 747 aircraft. Obviously, these planes are intended to be flown by experts who have years of experience. In this case, aesthetics is not the goal of the interior design of a cockpit; the goal is that of functionality.

In order to design an effective graphical user interface for an application, you must evaluate the goals of both your particular application and your intended audience. Only with a

complete understanding of these issues will you be able to determine the best interface to use. And remember, an irate customer just might call you for help.

In this chapter:

- *Basic X Toolkit Terminology and Concepts*
- *The Xm and Xt Libraries*
- *Programming With Xt and Motif*
- *Summary*

2

The Motif Programming Model

This chapter teaches the fundamentals of Motif by example. It dissects a simple "Hello, World" program, showing the program structure and style common to all Motif programs. Because much of this material is already covered in detail in Volume 4, *X Toolkit Intrinsics Programming Manual*, this chapter can be used as a refresher or a light introduction for those who haven't read the earlier book. It makes reference to Volume 1, *Xlib Programming Manual*, and Volume 4 to point out areas that the programmer needs to understand (windows, widgets, events, callbacks, resources, translations) before progressing with Motif.

Though we expect most readers of this book to be familiar with the X Toolkit Intrinsics (Xt), this chapter briefly reviews the foundations of Motif in Xt. This review serves a variety of purposes. First, for completeness, we define our terms, so if you are unfamiliar with Xt, you will not be completely at sea if you forge ahead. Second, there are many important aspects of the X Toolkit Intrinsics that we aren't going to cover in this book; this review gives us a chance to direct you to other sources of information about these areas. Third, Motif diverges from Xt in some important ways, and we point out these differences up front. Finally, we point out some of the particular choices you can make when Xt or Motif provides more than one way to accomplish the same task.

If you are unfamiliar with any of the concepts introduced in this chapter, please read the first few chapters of Volume 4. Portions of Volume 1, and Volume 3, *X Window System User's Guide*, may also be appropriate.

Basic X Toolkit Terminology and Concepts

As discussed in Chapter 1, *Introduction to Motif*, the Motif user-interface specification is completely independent of how it is implemented. In other words, you do not have to use the X Window System to implement a Motif-style graphical user interface (GUI). However, to enhance portability and robustness, the Open Software Foundation (OSF) chose to implement the Motif GUI using X as the window system and the X Toolkit Intrinsics as the platform for the Application Programmer's Interface (API).

Xt provides an object-oriented framework for creating reusable, configurable user-interface components called *widgets*. Motif provides widgets for such common user-interface elements as labels, buttons, menus, dialog boxes, scrollbars, and text-entry or display areas. In addition, there are widgets called managers, whose only job is to control the layout of other widgets, so the application doesn't have to worry about details of widget placement when the application is moved or resized.

A widget operates independently of the application, except through prearranged interactions. For example, a button widget knows how to draw itself, how to highlight itself when it is clicked on with the mouse, and how to respond to that mouse click.

The general behavior of a widget, such as a PushButton, is defined as part of the Motif library. Xt defines certain base classes of widgets, whose behavior can be inherited and augmented or modified by other widget classes (subclasses). The base widget classes provide a common foundation for all Xt-based widget sets. A *widget set*, such as Motif's Xm library, defines a complete set of widget classes, sufficient for most user-interface needs. Xt also supports mechanisms for creating new widgets or for modifying existing ones.

Xt also supports lighter-weight objects called *gadgets*, which for the most part look and act just like widgets, but their behavior is actually provided by the manager widget that contains them. For example, a pulldown menu pane can be made up of button gadgets rather than button widgets, with the menu pane doing much of the work that would normally be done by the button widgets.

Most widgets and gadgets inherit characteristics from objects above them in the class hierarchy. For example, the Motif PushButton class inherits the ability to display a label from the Label widget class, which in turn inherits even more basic widget behavior from its own superclasses. See Volume 4, *X Toolkit Intrinsics Programming Manual*, for a complete discussion of Xt's classing mechanisms; see Chapter 3, *Overview of the Motif Toolkit*, for details about the Motif widget class hierarchy.

The object-oriented approach of Xt completely insulates the application programmer from the code inside of widgets. As a programmer, you only have access to functions that create, manage, and destroy widgets, plus certain public widget variables known as *resources*. As a result, the internal implementation of a widget can change without requiring changes to the API. A further benefit of the object-oriented approach is that it forces you to think about an application in a more abstract and generalized fashion, which leads to fewer bugs in the short run and to a better design in the long run.

Creating a widget is referred to as instantiating it. You ask the toolkit for an *instance* of a particular widgetclass, which can be customized by setting its resources. All Motif PushButton widgets have the ability to display a label; an instance of the PushButton widget class actually has a label that can be set with a resource.

Creating widgets is a lot like buying a car: first you choose the model (class) of car you want, then you choose the options you want, and then you drive an actual car off the lot. There may exist many cars exactly like yours, others that are similar, and still others that are completely different. You can create widgets, destroy them, and even change their attributes just as you can buy, sell, or modify a car by painting it, adding a new stereo, and so on.

Widgets are designed so that many of their resources can be modified by the user at run-time. When an application is run, Xt automatically loads data from a number of system and user-specific files. The data from these files is used to build the *resource database*, which is used to configure the widgets in the application. If you want to keep the user from modifying resources, you can set their values when you create the widget. This practice is commonly referred to as *hard-coding* resources.

It is considered good practice to hard-code only those resource values that are essential to program operation and to leave the rest of the resources configurable. Default values for configurable resources are typically specified in an application defaults file, which is more colloquially referred to as the app-defaults file. By convention, this file is stored in the directory */usr/X11R6/lib/app-defaults* and it has the same name as the application with the first letter capitalized.The app-defaults file is loaded into the resource database along with other files that may contain different values set by the system administrator or the user. In the event of a conflict between different settings, a complex set of precedence rules determines the value actually assigned to a resource. See Volume 4, *X Toolkit Intrinsics Programming Manual*, for more information on how to set resources using the various resource files.

Motif widgets are prolific in their use of resources. For each widget class, there are many resources that neither the application nor the user should ever need to change. Some of these resources provide fine control over the three-dimensional appearance of Motif widgets; these resources should not be modified, since that would interfere with the visual consistency of Motif applications. Other resources are used internally by Motif to make one large, complex widget appear to the user in a variety of guises.

The *callback resources* for a widget are a particularly important class of resources that must be set in the application code. A widget that expects to interact with an application provides a callback resource for each type of interaction it supports. An application associates a function with the callback resources in which it is interested; the function is invoked when the user performs certain actions in the widget. For example, a PushButton provides a callback for when the user activates the button.

Note, however, that not every event that occurs in a widget results in a callback to an application function. Widgets are designed to handle many events themselves, with no interaction from the application. All widgets know how to draw themselves, for example. A widget may even provide application-like functionality. For example, a Text widget

typically provides a complete set of editing commands via internal widget functions called *actions*. Actions are mapped to events in a *translation table*. This table can be augmented, selectively overridden, or completely replaced by settings contained in the implementation of a widget class, in application code, or in a user's resource files.

In the basic Xt design, translations are intended to be configurable by the user. However, the purpose of Xt is to provide mechanism, not impose user-interface policy. In Motif, translations are typically not modified by either the user or the application programmer. While it is possible for an application to install event handlers or new translations and actions for a widget, most Motif widgets expect application interaction to occur only through callbacks.

Since the Motif widgets are designed to allow application interaction through callbacks, we don't discuss translations very often in this book. Some of the Motif widgets, particularly buttons when they are used in menus, have undefined behavior when their translations are augmented or overridden. An experienced Xt programmer may feel that Motif's limitations on the configurability of translations violates Xt. But consider that Xt is a library for building toolkits, not a toolkit itself. Motif has the further job of ensuring consistent user-interface behavior across applications.

Whether the goal of consistency is sufficient justification for OSF's implementation is a matter of judgement, but it should at least be taken into account. At any rate, you should be aware of the limitations when configuring Motif widgets. Motif widgets provide callback resources to support their expected behavior. If a widget does not have a callback associated with an event to which you want your application to respond, you should be cautious about adding actions to the widget or modifying its translations.

The Xm and Xt Libraries

A Motif user interface is created using both the Motif Xm library and the Intrinsics' Xt library. Xt provides functions for creating and setting resources on widgets. Xm provides the widgets themselves, plus an array of utility routines and convenience functions for creating groups of widgets that are used collectively as single user-interface components. For example, the Motif MenuBar is not implemented as one particular widget, but as a collection of smaller widgets put together by a convenience function.

An application may also need to make calls to the Xlib layer to render graphics or get events from the window system. In the application itself, rather than in the user interface, you may also be expected to make lower-level system calls into the operating system, file system, or hardware-specific drivers. The application may also be making use of the X11R6 Session Management (SM) and the X11R6 InterClient Exchange (ICE) facilities[*]. Thus, the whole application may have calls to various libraries within the system. Figure 2-1 represents the model for interfacing to these libraries.

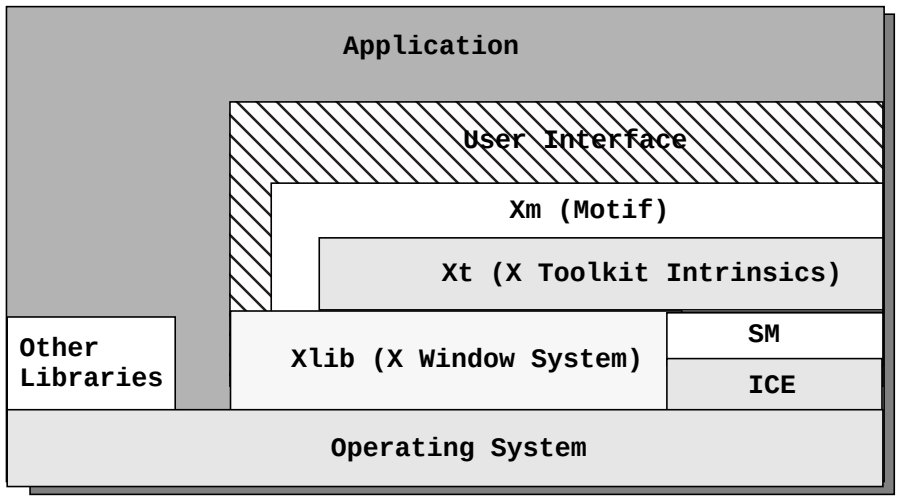


Figure 2-1 User interface library model

As illustrated above, the application itself may interact with all layers of the windowing system, the operating system, and other libraries (math libraries, rpc, database) as needed. On the other hand, the user-interface portion of the application should restrict itself to the Motif, Xt, and Xlib libraries whenever possible. This restriction aids in the portability of the user-interface across multiple computers and operating systems. Since X is a distributed windowing system, once the application runs on a particular computer, it can be displayed on any computer running X - even across a local or wide-area network.

In addition to restricting yourself to using the Motif, Xt, and Xlib libraries, you should try to use the higher-level libraries whenever possible. Focus on using Motif-specific widgets and functions, rather than trying to implement equivalent functionality using Xt or Xlib. Higher-level libraries hide a great number of details that you would otherwise have to handle yourself. By following these guidelines, you can reduce code complexity and size, creating applications that are easier to maintain.

In situations where the Motif library does not provide the functionality you need, you may attempt to borrow widgets from other toolkits or write your own. This technique is possible and made relatively simple because Motif is based on Xt.* For example, an application might make good use of a general-purpose graphing widget.

---

* SM and ICE are fully described in the *Programmer's Supplement for Release 6 of the X Window System.* We will conform to the X11R6 guidelines and use the SessionShell widget class throughout the examples; othewise, Session Management will not form part of this manual, and you are referred to the Supplement for more details.

* While this book discusses certain methods for extending the Motif library, you should refer to Volume 4, *X Toolkit Intrinsics Programming Manual*, for a general discussion of how to build your own widgets.

Whatever libraries you use, be sure to keep your application modular. The first and most important step in the development of an application is its design. You should always identify the parts of the application that are functional and the parts that make up the user interface. Well-designed applications keep the user-interface code separate from the functional code. You should be able to unplug the Motif code and replace it with another user-interface widget set based on Xt merely by writing corresponding code that mirrors the Motif implementation.

Programming With Xt and Motif

The quickest way to understand the basic Motif programming model is to examine a simple application. Example 2-1 is a version of the classic “hello world” program that uses the Motif toolkit.*

Example 2-1 The hello.c program

```
/* hello.c -- initialize the toolkit using an application context
** and a toplevel shell widget, then create a pushbutton that says
** Hello using the varargs interface.
*/

#include <Xm/PushB.h>

main (int argc, char *argv[])
{
    Widget          toplevel, button;
    XtAppContext    app;
    void            button_pushed(Widget, XtPointer, XtPointer);
    XmString        label;
    Arg             args[2];

    XtSetLanguageProc (NULL, NULL, NULL);
    toplevel = XtVaOpenApplication (&app, "Hello", NULL, 0, &argc, argv,
                                    NULL,sessionShellWidgetClass, NULL);
    label = XmStringCreateLocalized ("Push here to say hello");
    XtSetArg(args[0], XmNlabelString, label);
    button = XmCreatePushButton (toplevel, "pushme", args, 1);
    XmStringFree (label);
    XtAddCallback (button, XmNactivateCallback, button_pushed, NULL);
    XtManageChild (button);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

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

* XtVaAppInitialize() is deprecated in X11R6. The SessionShell widget class, and XtVaOpenApplication() are only availble in X11R6.

```
}
```

The output of the program is shown in Figure 2-2.

Figure 2-2 Output of the hello program

You can get the source code for *hello.c* and the rest of the examples in this book via anonymous *ftp* or other methods that are described in the Preface. It is a good idea to compile and run each example as it is presented.

The example programs come with Imakefiles that should make building them easy if you have the *imake* program. This program should already be in */usr/X11R6/bin* on UNIX-based systems that have X11 Release 6 installed. You also need the configuration files for *imake*; they are in */usr/X11R6/lib/config* on most UNIX-based systems. An Imakefile is a system-independent makefile that is used by *imake* to generate a Makefile. This process is necessary because it is impossible to write a Makefile that works on all systems. You invoke *imake* using the *xmkmf* program. Complete instructions for compiling the examples using *imake* are provided in the *README* file included with the source code.

As explained in the Preface, there are versions of the example programs for both Motif 2.1 and Motif 1.2 available electronically. However, all of the example code in this book is designed to work with Motif 2.1 (and X11R6); the programs use functions that are not available in Motif 1.2 (and X11R5). Where we use Motif 2.1 functions, we try to mention how to perform the same tasks using Motif 1.2, usually in a footnote. To use the example programs with Motif 1.2, make the changes we describe. When the necessary changes are significant, we may explain both versions of the program. For a description of the changes that we made to convert the example programs to Motif 2.1, see Changes in Motif 2.1, in Chapter 3.

To compile any of the examples on a UNIX system without using imake, use the following command line:

```
cc -O -o filename filename.c -lXm -lXt -lX11
```

If you want to do debugging, replace -O with -g in this command line. The order of the libraries is important. Xm relies on Xt, and both Xm and Xt rely on Xlib (the `-lX11` link flag specifies Xlib).

Now let's take a look at this program step by step, noting elements of the underlying Xt model and where Motif differs from it.

Header Files

An application that uses the Motif toolkit must include a header file for each widget that it uses. For example, *hello.c* uses a PushButton widget, so we include <*Xm/PushB.h*>.The appropriate header file for each Motif widget class is included on the reference page for the widget in Volume 6B, *Motif Reference Manual*.

If you simply browse through */usr/Motif2.1/include/Xm* (or wherever you have installed your Motif distribution) trying to find the appropriate header file, you will find that each widget class actually has two header files. The one with the name ending in a "P" (e.g. *PushBP.h*) is the widget's private header file and should not normally be included directly by an application. Private header files are generally used only by the code that implements a widget class and its subclasses.

Xt uses public and private header files to hide the details of widget implementation from applications. This technique provides object-oriented encapsulation and data hiding in the C language, which is not designed to support object-oriented programming. (See Volume 4, *X Toolkit Intrinsics Programming Manual*, for additional information on the object-oriented design of widgets.)

For some types of objects, you may see another pair of header files, each containing a capital "G" at the end of their names (for example, *PushBG.h* and *PushBGP.h*). These files are for the gadget version of the object. For the most part, when we talk about widgets, we include gadgets. Later chapters make it clear when to use gadgets and when to use widgets.

A quick examination of the `#include` directives in each of the Motif widget or gadget header files reveals that each of them includes <*Xm/Xm.h*>, the general header file for the Motif library. <*Xm/Xm.h*> in turn includes the following files:

```
#include <X11/Intrinsic.h>
#include <X11/Shell.h>
#include <X11/Xatom.h>
#include <Xm/XmStrDefs.h>
#include <Xm/VirtKeys.h>
```

Therefore, none of these files ever need to be included by your application, as long as you include <*Xm/Xm.h*>. Since <*Xm/Xm.h*> is included by each widget header file, you do not need to include it directly either. If you look closely at the code, you'll see that just about every necessary header file is included the moment you include your widget header file. This method of using header files contrasts with the way other Xt-based toolkits, like the Athena toolkit or the OPEN LOOK Intrinsics Toolkit (OLIT), use header files.

The Motif toolkit provides a new header file, <*Xm/XmAll.h*>, that simply includes all of the public header files.

We recommend that you not duplicate the inclusion of header files. One reason is that if you include only the header files that you need, whoever has to maintain your code can see

which widgets you are dealing with in your source files. Another reason is that duplicating header file is generally bad practice, as you run the risk of redeclaring macros, functions, and variables.

However, it isn't always easy to prevent multiple inclusions. For example, <*Xm/Xm.h*> is included by each widget header file that you include. All of the Motif, Xt and X header files are protected from multiple inclusion using a technique called *ifdef-wrapping*. We recommend that you use this method in your own header files as well. The ifdef-wrapper for <*X11/Intrinsic.h*> is written as follows:

```
#ifndef _XtIntrinsic_h
#define _XtIntrinsic_h
/* Include whatever is necessary for the file... */
#endif /* _XtIntrinsic_h */
```

The wrapper defines `_XtIntrinsic_h` when a file is first included. If the file is ever included again during the course of compiling the same source (*.c*) file, the `#ifdef` prevents anything from being redeclared or redefined.

Of course, the wrapper prevents multiple inclusion only within a single source file; the next source file that gets compiled goes through the same test. If the same files are included, the same macros, data types, and functions are declared again for the benefit of the new file. For this reason, you should never write functions in a header file, since it would be equivalent to having the same function exist in every source file. Function declarations, however, are acceptable and expected.

In addition to the widget header files, you will most likely need other include files specific to your application, such as <*stdio.h*> or <*ctype.h*>.

The order of inclusion is generally not important unless certain types or declarations required by one file are declared in another. In this case, you should include the files in the necessary order. Otherwise, application-specific header files are usually included first, followed by UI-specific header files (with Xt header files, if any, preceding Motif header files), followed by system-specific header files.

Setting the Language Procedure

For Release 5 of the X Window System, the X Toolkit was modified to better support internationalization. An internationalized application retrieves the user's language (called a *locale*) from the environment or a resource file and operates in that language without changes to the binary. An internationalized application must display all of its text in the user's language and accept textual input in that same language. It must also display dates, times, and numbers in the appropriate format for the language environment.

X internationalization is based on the ANSI-C internationalization model. This approach is based on the concept of *localization*, whereby an application uses a library that reads a customizing database at start-up time. This database contains information about the

characteristics of every locale that is supported by the system. When an application establishes its locale by calling `setlocale()`, the library customizes the behavior of various routines based on the locale. See the Third Edition of Volume 1, *Xlib Programming Manual*, for a complete description of the concepts and implementation of X internationalization.

Xt support of internationalization is trivial in most applications; the only additional code needed is a call to `XtSetLanguageProc()` before the toolkit is initialized. `XtSetLanguageProc()` sets the *language procedure* that is used to set the locale of an application. The first argument to the routine specifies an application context, the second argument specifies the language procedure, and the third parameter specifies additional data that is passed to the language procedure when it is called. Since the language procedure is responsible for setting the locale, an Xt application does not call `setlocale()` directly. The language procedure is called by `XtDisplayInitialize()`.

If the second argument to `XtSetLanguageProc()` is `NULL`, the routine registers a default language procedure. Here's the call that we used in Example 2-1 to set the default language procedure:

```
XtSetLanguageProc (NULL, NULL, NULL);
```

The default language procedure sets the locale according to the LANG environment variable, verifies that the current locale is supported, and returns the value of the current locale. For more information about establishing the locale in an Xt application, see Volume 4, *X Toolkit Intrinsics Programming Manual*.

Most of the support for internationalization in Motif is provided by Xlib and Xt. Xlib provides support for internationalized text output, interclient communication, and localization of the resource database, while Xt handles establishing the locale. The Motif Text and TextField widgets have been modified to support internationalized text input and output; see Text Widget Internationalization in Chapter 18, for more information. The Motif routines that work with compound strings and render tables* have also been updated in Motif 2.1. See Chapter 24, *Render Tables*, and Chapter 25, *Compound Strings*, for details on the new API for `XmString` and `XmRenderTable` values.

Initializing the Toolkit

Before an application creates any widgets, it must initialize the toolkit. There are many ways to perform this task, most of which also perform a number of related tasks, such as opening a connection to the X server and loading the resource database. Here's a list of some of the things that are almost always done:

- Open the application's connection to the X server.

* The `XmFontList` is obsolete in Motif 2.0 and later, and is replaced by the `XmRenderTable`.

- Parse the command line for the standard X Toolkit command-line options plus any custom command-line options that have been defined for the application.
- Create the resource database using the app-defaults file, if any, as well as any user, host, and locale-specific resource files.
- Create the application's top-level window, a Shell class widget that handles interaction with the window manager and acts as the parent of all of the other widgets in the application.

There are several functions available to perform toolkit initialization. The one we use throughout is `XtVaOpenApplication()`*, since it performs all of the functions listed above in one convenient call. Here's the call we used in Example 2-1:

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

The widget returned by `XtVaOpenApplication()` is a shell widget. The shell widget acts as the top-level window of the application and handles the application's interaction with the window manager. The SessionShell widget class which we will use for the top level also interacts with the X11R6 Session Management facilities. All of the other widgets created by the application are created as descendents of the shell, of which we'll talk more later in this chapter.

The Application Context

The first argument to `XtVaOpenApplication()` is the address of an application context, which is a structure that Xt uses to manage some internal data associated with an application. Most applications do not manipulate the application context directly. Most often, an application receives an opaque pointer to an application context in the toolkit initialization call and merely passes that pointer to a few other toolkit functions that require it as an argument. The fact that the application context is a public variable, rather than hidden in the toolkit internals, is a forward-looking feature of Xt, designed to support multiple threads of control.

The X11R5 initialization call `XtVaAppInitialize()` is still supported by later versions of the toolkit. Its use is discouraged because the new initialization calls provide a greater degree of upward compatibility with future Xt-based applications.

The Application Class

The second argument to `XtVaOpenApplication()` is a string that defines the *class name* of the application. A class name is used in resource files to specify resource values

* `XtVaAppInitialize`() is considered deprecated in X11R6. `XtVaOpenApplication`() and the SessionShell widget class are only availble in X11R6.

that apply to all instances of an application, a widget, or a resource. (See Volume 3, *X Window System User's Guide*, and Volume 4, *X Toolkit Intrinsics Programming Manual*, for details.) For many applications, the application class is rarely used and the class name is important only because it is also used as the name of the application's app-defaults file.

Whenever a widget is created in Xt, its resources must have certain initial (or default) values. You can either hard-code the values, allow them to default to widget-defined values, or specify the default values in the app-defaults file. These default values are used unless the user has provided his own default settings in another resource file.

By convention, the class name is the same as the name of the application itself, except that the first letter is capitalized.* For example, a program named *draw* would have a class name of *Draw* and an app-defaults filename of */usr/X11R6/lib/app-defaults/Draw*. Note, however, that there is no requirement that an app-defaults file with this name actually be installed.

Exceptions can be made to this convention, as long as you document it. For example, all the example programs in this book have the class name of *Demos*, which allows us to set certain common defaults in a single file. This technique can be useful whenever you have a large collection of independent programs that are part of the same suite of applications.

Command-line Arguments

The third and fourth arguments specify an array of objects that describe the command-line arguments for your program, if any, and the number of arguments in the array. These arguments are unused in most of the examples in this book and are specified as `NULL` and `0`, respectively. The program *xshowbitmap.c* in the Appendix A, *Additional Example Programs*, provides an example of using command-line arguments. See Volume 4, *X Toolkit Intrinsics Programming Manual*, for a more complete discussion of application-specific command-line arguments.

The fifth and sixth arguments contain the value (`argv`) and count (`argc`) of any actual command-line arguments. The initialization call actually removes and acts on any arguments it recognizes, such as the standard X Toolkit command-line options and any options that you have defined in the third argument. After this call, `argv` should contain only the application name and any expected arguments such as filenames. You may want to check the argument count at this point and issue an error message if any spurious arguments are found.

* Some applications follow the convention that if the application's name begins with an "X", the X is silent and so the second letter is capitalized as well. For example, the class name of *xterm* is *XTerm*.

Fallback Resources

The seventh argument is the start of a NULL-terminated list of *fallback resources* for the shell widget created by the initialization call. Fallback resources provide a kind of "belt and suspenders" protection against the possibility that an app-defaults file is not installed. They are ignored if the app-defaults file or any other explicit resource settings are found. When no fallback resources are specified, the seventh argument should be NULL.

It is generally a good idea to provide fallbacks for resources that are essential to the operation of your application. An example of how fallback resources can be used by an application is shown in the following code fragment:

```
String fallbacks[] =
{
    "Demos*background: white",
    "Demos*XmText.foreground: black",
    /* list the rest of the app-defaults resources here... */
    NULL
};
...
toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv, fallbacks,
                                sessionShellWidgetClass, NULL);*
...
```

Fallback resources protect your application against a missing app-defaults file, but they do not guard against one that is modified incorrectly or otherwise corrupted, since they are not used if the app-defaults file is present in any form. A better fallback mechanism would provide protection against these types of problems. Fortunately, there is the function XrmCombineDatabases(), that allows you to provide real fallbacks in case the user or the system administrator misconfigures the app-defaults file.

The Top Level Shell Class

The eighth parameter specifies the type of shell to be used for the top level. It is recommended that this is the sessionShellWidgetclass, which is derived from the applicationShellWidgetClass†. We are not actually using any of the features of the X11R6 SessionShell in the examples, however we will use the SessionShell in order to conform to the recommendations.

Additional Initialization Parameters

The ninth parameter is the start of a NULL-terminated list of resource/value pairs that are applied to the top-level widget returned by XtVaOpenApplication(). If there are no resource settings, which is often the case for this function, you can pass NULL as the ninth parameter. If you do pass any parameters, it should be done just as we describe for

* XtVaAppInitialize() is deprecated in X11R6.

† The ApplicationShell is considered obsolete in X11R6. The SessionShell is only availble in X11R6.

XtVaCreateWidget() later in this chapter. All of the functions whose names begin with XtVa support the same type of varargs-style (variadic) argument lists.

The X11 Release 6 implementation of XtVaOpenApplication() and other varargs functions may not work entirely as expected for some non-ANSI-C compilers due to a bug in the way that Xt declares variadic functions. This problem only arises for some compilers that do not understand function prototypes. The problem is rare since it is compiler-dependent and it only happens on older compilers. It is not a compiler error but an Xt error, since functions are not supposed to mix fixed parameter declarations with variadic declarations. XtVaOpenApplication() mixes these declarations; the first eight parameters are fixed while the ninth through *n*th arguments are variadic. ANSI-C allows, and even requires, this type of specification.

If you experience problems such as segmentation faults or bus errors as a result of using XtVaOpenApplication(), you can try passing an extra NULL parameter after the final NULL. Another option is to use XtOpenApplication(), which is identical to XtVaOpenApplication(), but does not contain a variable argument list of resource/values pairs. Instead, it uses the non-variadic args and num_args method of specifying resource values, which we describe later in this chapter.

Creating Widgets

There is a convenience function for creating every class of widget and gadget supported by the Motif toolkit. For example, to create a PushButton widget, you can use the function XmCreatePushButton(). To create the corresponding gadget, you can use XmCreatePushButtonGadget(). In addition, there are convenience functions for creating *compound objects*. A compound object is a collection of widgets that is treated like a single object. For example, a ScrolledList object is really a List widget inside a ScrolledWindow widget. XmCreateScrolledList() creates the compound object consisting of both widgets.

The convenience functions for creating all of the different types of widgets are described in Volume 6B, *Motif Reference Manual*. In addition to the convenience routines, the Xt Intrinsics also define generic routines which can be used to create arbitrary widget instances, namely XtVaCreateWidget() and XtVaCreateManagedWidget(). These functions allow you to decide whether to create a widget as managed or unmanaged, while the Motif convenience functions always create unmanaged widgets. The Xt routines also allow you to set resources for a widget using the varargs interface, which can often be more convenient than the args and num_args method used by the Motif creation routines.

X nests windows using a parent-child model. A display screen is defined as the root window; every application has a top-level window that is a child of the root window. A top-level window in turn has subwindows, which overlay it but cannot extend beyond its boundaries. If a window extends beyond the boundaries of its parent, it is clipped.

Because every widget has its own X window, widgets follow a similar parent-child model. Whenever a widget is created, it is created as the child of another widget. The shell widget returned by the call to `XtVaOpenApplication()` is the top-level widget of an application. It is usually overlaid with a special class of widget called a *manager widget*, which implements rules for controlling the size and placement of widget children. For example, the Motif RowColumn widget is a manager that allows widgets to be laid out in regular rows and columns, while the Form widget is a manager that allows widgets to be placed at precise positions relative to one another. A manager widget can contain other manager widgets as well as *primitive widgets*, which are used to implement actual user-interface controls. Managers also support gadgets. A gadget is a lighter-weight object that is identical to its corresponding widget in general functionality, but does not have its own window.

In Example 2-1, the button was created as a child of the top-level shell window.This simple application contains only one visible widget, so it does not use a manager. Actually, shells are extremely simple managers. A shell can only have one child; the shell makes itself exactly the same size as the child so the shell remains invisible behind the child. Here's the call we used to create the button:

```
button = XmCreatePushButton (toplevel, "pushme", args, 1);
```

The first argument is the parent of the widget, which must be a manager widget that has already been created. In this example, the parent of the PushButton widget is toplevel, the shell widget returned by the call to `XtVaOpenApplication`().

The second argument is a string that is used as the name of the widget in the resource database. If a user wants to specify the color of the button label for the application, she can use the following specification in a resource file:

```
hello.pushme.foreground: blue
```

The name is different from the variable name that is used to refer to the widget in application code. The following resource specification is not correct:

```
hello.button.foreground: blue
```

The resource name does not need to be identical to the variable name given to the widget inside the program, though to minimize confusion, many programmers make the two names the same. If you want users to be able to configure widget resources, be sure to include the names of the widgets in your documentation.

The remainder of the argument list is an array of resource settings, followed by the number of items in this array. We'll talk about the format of these resource settings in the next section.

Setting and Getting Widget Resources

A widget class defines resources of its own and it inherits resources from its superclasses. The names of the resources provided by each widget class (new and inherited) are documented in the widget reference pages in Volume 6B, *Motif Reference Manual*. The most useful resources are described in detail in the individual chapters on each of the Motif widget classes.

When resources are set in a program, each resource name begins with the prefix `XmN`. These names are mnemonic constants that correspond to actual C strings that have the same name without the `XmN` prefix. For example, the actual resource name associated with `XmNlabelString` is `labelString`. The `XmN` identifies the resource as being Motif-related. Motif also uses the `XmC` prefix to identify resource class symbols. Xt uses the prefix `XtN` for any resources defined by its base widget classes. Motif also provides corresponding `XmN` names for most of these resources.* When you are specifying resources in a resource file or when you are using the `-xrm` option to specify resources on the command line, omit the `XmN` prefix.

The main purpose of the constant definitions for resource names is to allow the C preprocessor to catch spelling errors. If you use the string `width` rather than the constant `XmNwidth`, the program still works. However, if you type `widdth`, the compiler happily compiles the application, but your program won't work and you'll have a difficult time trying to figure out why. Because resource names are strings, there is no way for Xt or Motif to report an error when an unknown resource name is encountered. On the other hand, if you use `XmNwiddth`, then the compiler complains that the token is an undefined variable.

Setting Resources During Widget Creation

The Motif convenience functions, as well as the Xt functions `XtCreateWidget()` and `XtCreateManagedWidget()`, require you to declare resource settings in an array. You pass this array to the function, along with the number of items in the array. By contrast, the varargs-style functions in Xt allow you to specify resources directly in a creation call, as a `NULL`-terminated list of resource/value pairs.

As an example, in the call to `XmCreatePushButton()` in *hello.c*, the only resource set was the string displayed as the PushButton's label, and this was passed to the creation routine in the Arg array *args*. Alternatively, a variable length list of resources could have been set in the same call using the Xt mechanisms, as shown in the following code:

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

* Some toolkits use the `XtN` prefix, even though its resource are not common to all Xt toolkits. If you need access to an Xt-based resource that does not have a corresponding `XmN` constant, you need to include the file *<X11/StringDefs.h>*.

These settings specify that the widget is 200 pixels wide by 50 pixels high, rather than its default size, which would be just big enough to display its label.

When you set resources in the creation call for the widget, those resources can no longer be configured by the user. Such resources are said to be hard-coded. For example, since we've set the width and height of the PushButton in the call to `XtVaCreateManagedWidget()`, a user resource specification of the following form is ignored:

```
*pushme.width: 250
*pushme.height: 100
```

It is recommended that you hard-code only those resource values that are absolutely required by your program. Most widgets have reasonable default values for their resources. If you need to modify the default values, specify the necessary resource values in an app-defaults file, instead of in the application code.

Every resource has a data type that is specified by the widget class defining the resource. When a resource is specified in a resource file, Xt automatically converts the resource value from a string to the appropriate type. However, when you set a resource in your program, you must specify the value as the appropriate type. For example, the Motif PushButton widget expects its label to be a compound string (see Chapter 25, *Compound Strings*), so we create a compound string, use it to specify the resource value, and free it when we were done.

Rather than specifying a value of the appropriate type, you can invoke Xt's resource converters in a varargs list using the keyword `XtVaTypedArg`, followed by four additional parameters: the resource name, the type of value you are providing, the value itself, and the size of the value in bytes. Xt figures out the type of value that is needed and performs the necessary conversion. For example, to specify the background color of the button directly in our program without calling an Xlib routine to allocate a colormap entry, we can use the following code:

```
button = XtVaCreateManagedWidget ("pushme", xmPushButtonWidgetClass, toplevel,
                            XmNlabelString, label,XtVaTypedArg, XmNbackground,
                             XmRString, "red", strlen ("red") + 1, NULL);
```

The data type in this construct is specified using a special symbol called a *representation type*, rather than the C type. An `XmR` prefix identifies the symbol as a representation type. See Volume 4, *X Toolkit Intrinsics Programming Manual*, for more information on resource type conversion and the possible values for representation types. These symbols are defined in the same way as the `XmN` symbols that are used for resource names.

Setting Resources After Widget Creation

After a widget has been created, you can set resources for it using `XtVaSetValues()`. The values set by this function override any values that are set either in the widget creation call or in a resource file. The syntax for using `XtVaSetValues()` is:

```
XtVaSetValues (widget_id, resource-value-list, NULL);
```

The `widget_id` is the value returned from a widget creation call, and *`resource-value-list`* is a `NULL`-terminated list of resource/value pairs.

Some Motif widget classes also provide convenience routines for setting certain resources. For example, `XmToggleButtonSetState()` sets the `XmNset` resource of a ToggleButton. The available convenience functions are described in Volume 6B, *Motif Reference Manual*, and in the chapters on each widget class in this book. A convenience function has direct access to the internal fields in a widget's data structures, so it might have slightly better performance than `XtVaSetValues()`. Functionally, however, the two methods are generally freely interchangeable.

Getting Resource Values

The routine used to get widget resource values is `XtVaGetValues()`. The syntax of this routine is exactly the same as `XtVaSetValues()`, except that the value part of the resource/value pair is the address of a variable that stores the resource value. For example, the following code gets the label string and the width for a Label widget:

```
extern Widget      label;
XmString           str;
Dimension          width;
...
XtVaGetValues (label, XmNlabelString, &str, XmNwidth, &width, NULL);
```

Notice that the value for `XmNlabelString` is an `XmString`, which is a Motif compound string. Almost all of the Motif widget resources that specify textual information use compound strings rather than regular character strings. The `XmNvalue` and `XmNvalueWcs` resources for Text and TextField widgets are the only exceptions to this policy. When you are retrieving a string resource from a widget, make sure that you pass the address of a compound string, not a character string, as in the following incorrect example:

```
extern Widget      label;
char               *buf;
Dimension          width;
...
XtVaGetValues (label, XmNlabelString, &buf, /* do not do this  */ XmNwidth,
                  &width, NULL);
```

If you try to get a compound string resource value with a character string variable, the program still works, but the value of the character string is meaningless. The correct way to handle a compound string resource is to retrieve it with an `XmString` variable and then get the character string from the compound string using `XmStringUnparse()`. See Chapter 25, *Compound Strings*, for more information.

There are some things to be careful about when you are getting resource values from a widget. First, always pass the address of the variable that is being used to store the retrieved value. A value represented by a pointer is not copied into the address space. Instead, the

routine sets the value for the address of the pointer to the position of the internal variable that contains the desired value. If you pass an array, rather than a pointer to the array, the routine cannot move its address. If you pass the address of a pointer, `XtVaGetValues()` is able to reset the pointer to the correct internal value.* For values that are not represented by pointers, such as integers, the value is simply copied. For example, the `width` value is an `int`, so the resource value is copied into the variable.

You should also be careful about changing the value of a variable returned by `XtVaGetValues()`. In the case of a variable that is not a pointer, the value can be changed because the variable contains a copy of the value and does not point to internal data for the widget. However, if the variable is a pointer to a string or a data structure, it does point to internal data for the widget. If you dereference the pointer and change the resulting value, you are changing the internal contents of the widget. This technique should not be used to change the value of a resource. To modify a resource value, you should use `XtVaSetValues()` with a defined resource name, as this routine ensures that the widget redraws and manages itself appropriately.

Motif also provides convenience routines for getting certain resource values from particular widget classes. Most of these functions correspond to the convenience routines for setting resource values. Many of the functions allocate memory for the value that is returned. For example, `XmTextGetString()` allocates space for and returns a pointer to the text in a Text widget. When a convenience function for retrieving a resource value is available, we generally recommend using it.

Using Argument Lists

The Motif convenience functions, and some Xt functions like `XtCreateWidget()` and `XtCreateManagedWidget()`, require you to set resources using a separately-declared array of objects of type `Arg`. You pass this array to the appropriate function along with the number of items in the array.

For example, the following code fragment creates a Label widget using a Motif convenience routine:

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

For all of the Motif convenience routines, the first argument is the parent of the widget being created, the second argument is the widget's name, and the third and fourth

* The Motif toolkit sometimes sets the given address to allocated data, which must be freed when it is no longer needed. This situation occurs when a compound string resource is retrieved from a widget and when the text value of a Text widget is retrieved. These cases are discussed in Chapter 18, *Text Widgets*, and Chapter 25, *Compound Strings*.

arguments are the array of resource specifications and the number of resources in the array. Since the class of the widget being created is reflected in the name of the convenience function, it does not need to be specified as an argument to the routine. For example, `XmCreateLabel()` creates a Label widget, while `XmCreatePushButton()` creates a PushButton widget.

Xt also provides some generic widget creation functions that use the non-variadic argument lists for specifying widget resources. The following code fragment shows the use of `XtCreateWidget()`:

```
Arg      args[5];
int      n = 0;
XtSetArg (args[n], XmNlabelString, label); n++;
label = XtCreateWidget ("label", xmLabelWidgetClass, toplevel, args, n);
XtManageChild (label);
```

With this routine, the name of the widget is the first parameter, the widget class is the second parameter, and the parent is the third parameter. The fourth and fifth parameters specify the resources, as in the Motif convenience routines. Functionally, in this instance the two methods of widget creation are logically identical, and it simply boils down to a question of personal taste. In examples, we will prefer the Motif creation routines, if only because this is a Motif and not an Xt manual.

The argument-list style of setting resources is a touch clumsy and error-prone, since it requires you to declare an array (either locally or statically) and to make sure that it has enough elements. It is a common programming mistake to forget to increase the size of the array when new resource/value pairs are added; this error usually results in a segmentation fault.

In spite of the disadvantages of this method of setting resources, there are still cases where the convenience routines are logically preferred (as opposed to purely stylistic considerations). One such case is when the routine creates several widgets and arranges them in a predefined way consistent with the *Motif Style Guide*. The argument-list style functions also can be useful when you have different resources that should be set depending on run-time constraints. For example, the following code fragment creates a widget whose foreground color is set only if the application knows it is using a color display:

```
extern Widget parent;
Arg           args[5];
Pixel         red;
int           n = 0;
XtSetArg (args[n], XmNlabelString, label); n++;
if (using_color) {
    XtSetArg (args[n], XmNforeground, red); n++;
}
...
widget = XmCreatePushButton (parent, "name", args, n);
```

The non-variadic routines also allow you to pass the exact same set of resources to more than one widget. Since the contents are unchanged, you can reuse the array for as long as it is still available. Be careful of scoping problems, such as using a local variable outside of the function where it is declared. The following code fragment creates a number of widgets that all have the same hard-coded resources:

```
static char *labels[] = { "A Label", "Another Label", "Yet a third" };
XmString     label;
Widget       widget, rc;
Arg          args[3];
int          i, n = 0;

/* Create an unmanaged RowColumn widget parent */
rc = XmCreateRowColumn (parent, "rc", NULL, 0);

/* Create RowColumn's children -- all 50x50 with different labels */
XtSetArg (args[n], XmNwidth, 50); n++;
XtSetArg (args[n], XmNheight, 50); n++;

for (i = 0; i < XtNumber (labels); i++) {
    xm_label = XmStringCreateLocalized (labels[i]);
    XtSetArg (args[n], XmNlabelString, xm_label);
    widget = XmCreateLabel (rc, "label", args, n + 1);
    XtManageChild (widget);
    XmStringFree (xm_label);
}
/* Now that all the children are created, manage RowColumn */
XtManageChild (rc);
```

Each Label widget is created with the same width and height resource settings, while each `XmNlabelString` resource is distinct. All other resource settings for the widgets can be set in a resource file.

To set resources in a resource file, you need to specify the names of the widgets, which in this case are all set to *label*. It is perfectly legal to give the same name to more than one widget. As a result, a resource specification in a resource file that uses a particular name affects all of the widgets with that name, provided that the widget tree matches the resource specification. For example, you could set the foreground color of all of the Labels using the following resource specification:

```
*rc.label.foreground: red
```

Other widgets in the application that have the widget name *label*, but are not children of the widget named *rc*, are not affected by this specification. Obviously, whether you really want to use the same name for a number of widgets is dependent on your application. This technique makes it easier to maintain a consistent interface, but it also limits the extent to which the application can be customized.

We could have used the elements of the `labels` array as widget names, but in this example, these strings contain spaces, which are “illegal” widget names. If you want to

allow the user to specify resources on a per-widget basis, you cannot use spaces or other non-alphanumeric characters, except the hyphen (-) and the underscore(_), in widget names. If per-widget resource specification is not a concern, you can use any widget name you like, including NULL or the null string ("").

Even if a widget has an illegal name, the user can still specify resources for it using the widget class, as in the following example:

```
*rc.XmLabel.foreground: red
```

This resource setting causes each Label widget to have a foreground color of red, regardless of the name of the widget (and provided that the resource value is not hard-coded for the widget). See Volume 4, *X Toolkit Intrinsics Programming Manual*, for a discussion of appropriate widget names and further details on resource specification syntax.

Event Handling for Widgets

Once we have created and configured the widgets for an application, they must be hooked up to application functions via callback resources. Before we can talk about callback resources and callback functions, we need to discuss events and event handling. In one sense, the essence of X programming is the handling of asynchronous events. Events can occur in any order, in any window, as the user moves the pointer, switches between the mouse and the keyboard, moves and resizes windows, and invokes functions available through user interface components. X handles events by dispatching them to the appropriate application and to the separate windows that make up each application.

Xlib provides many low-level functions for handling events. In special cases, which are described later in this book, you may need to dip down to this level to handle events. However, Xt simplifies event handling by having widgets handle many events for you, without any application interaction. For example, widgets know how to redraw themselves, so they respond automatically to Expose events, which are generated when one window is covered up by another and then uncovered. These "widget survival skills" are handled by functions called *methods* deep in the widget internals. Some typical methods redraw the widget, respond to changes in resource settings that result from calls to XtVaSetValues(), and free any allocated storage when the widget is destroyed.

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

Consider the PushButton in *hello.c*. Run the program and note how the widget highlights its border as the pointer moves into it, displays in reverse-video when you click on it, and switches back when you release the button. Watch how the highlighting disappears when you move the pointer out of the widget. Also, notice how pressing the SPACEBAR while

the pointer is in the widget has the same effect as clicking on it. These behaviors are the kinds of things that are captured in the widget's translation table:

```
<Btn1Down>:                          Arm()
<Btn1Down>, <Btn1Up>:                Activate() Disarm()
<Btn1Down>(2+):                      MultiArm()
<Btn1Up>(2+):                        MultiActivate()
<Btn1Up>:                            Activate() Disarm()

<Btn2Down>:                          ProcessDrag()
<Key>osfSelect:                      ArmAndActivate()
<Key>osfActivate:                    PrimitiveParentActivate()
<Key>osfCancel:                      PrimitiveParentCancel()
<Key>osfHelp:                        Help()
~Shift ~Meta ~Alt <Key>Return:       PrimitiveParentActivate()
~Shift ~Meta ~Alt <Key>space:        ArmAndActivate()
<EnterWindow>:                       Enter()
<LeaveWindow>:                       Leave()
```

The translation table contains a list of event translations on the left side, with a set of action functions on the right side. When an event specified on the left occurs, the action routine on the right is invoked. As we just described, moving the pointer in and out of the PushButton causes some visual feedback. The `EnterWindow` and `LeaveWindow` events generated by the pointer motion cause the `Enter()` and `Leave()` actions to be invoked.

As another example, when the first mouse button is pressed down inside the PushButton, the `Arm()` action routine is called. This routine contains the code that displays the button as if it were "pushed in," as opposed to "pushed out." When the mouse button is released, both the `Activate()` and `Disarm()` routines are invoked in that order. Here is where your application actually steps in. If you have provided an appropriate callback function, the `Activate()` action calls it. The `Disarm()` routine causes the button to be redrawn so that it appears "pushed out" again.

Event Specification

In the Xt syntax, events are specified using symbols that are tied fairly closely to pure X hardware events, such as `ButtonPress` or EnterWindow. For example, `<Btn1Down>` specifies a button press for the first mouse button. `KeyPress` events are indicated by symbols called *keysyms*, which are hardware-independent symbols that represent individual keystrokes. Different keyboards may produce different hardware *keycodes* for the same key; the X server uses keysyms as a portable representation, based on the common labels found on the tops of keys.

Motif provides a further level of indirection in the form of *virtual keysyms*, which describe key events in a completely device-independent manner. For example, `osfActivate` indicates that the user invoked an action that Motif considers to be an activating action. An activating action typically corresponds to the RETURN key being pressed or the left mouse

button being clicked. Similarly, osfHelp corresponds to a user request for help, such as the HELP or F1 key being pressed.

Virtual keysyms are supposed to be provided by the vendor of the user's hardware, based on the keys on the keyboard, but some X vendors also provide keysym databases to support multiple keyboards. The X Consortium provides a virtual keysym database in the file */usr/X11R6/lib/XKeysymDB*. This file contains a number of predefined key bindings that OSF has registered with the X Consortium to support actions in the Motif toolkit.

Virtual keysyms can be invoked by physical events, but the Motif toolkit goes one step further and defines them in the form of *virtual bindings*. Here's the translation table for the PushButton widget expressed using virtual bindings:

```
BSelect Press:              Arm()
BSelect Click               Activate() Disarm()
BSelect Release:            Activate() Disarm()
BSelect Press 2+:           MultiArm()
BSelect Release 2+:         MultiActivate() Disarm()
BTranserPress:              ProcessDrag()
KSelect:                    ArmAndActivate()
KHelp:                      Help()
```

Examples of virtual bindings are BSelect, which corresponds to the first mouse button, and KHelp, which is usually the HELP key on the keyboard. The rule of thumb is that any virtual binding beginning with a "B" corresponds to a mouse button event, while any binding beginning with a "K" corresponds to a keyboard event. More than one event can be bound to a single virtual keysym. For example, the *Motif Style Guide* permits F1 to be a help key, so that key is also virtually bound to KHelp.

Virtual bindings can be specified by a system administrator, a user, or an application. One common use of virtual bindings is to reconfigure the operation of the BACKSPACE and DELETE keys. On some keyboards, the BACKSPACE key is in a particularly difficult location for frequent access. Users of this type of keyboard may prefer to use the DELETE key for backspacing. These people may find the default operation of the Motif Text widget annoying, since it does not allow them to backspace using their "normal" backspace key.

Since Xt allows applications and users to override, augment, or replace translation tables, many people familiar with Xt try to specify a new translation for the DELETE key to make it act like a backspace. The translation invokes the action routine that backspaces in a Text widget. However, this approach is limited, in that it only works for a single Text widget. The Text widget has the following translation:

```
<Key>osfBackSpace:   delete-previous-char()
```

The virtual keysym osfBackSpace is bound to delete-previous-char(), which is the backspace action. Rather than changing the translation table to specify that <Key>Delete should invoke this action, a user can redefine the virtual binding of the osfBackSpace keysym. A user can configure his own bindings by specifying the new

virtual keysym bindings in a *.motifbind* file in his home directory. The following virtual binding specifies that the DELETE key is mapped to osfBackSpace:l

```
osfBackSpace:    <Key>Delete
```

As a result of this specification, the DELETE key performs the backspace action in the Text widget, as well as any other widgets in the Motif toolkit that use the osfBackSpace keysym. The advantage of using virtual bindings is that the interface remains consistent and nothing in the toolkit or the application needs to change.

Virtual keysym bindings can also be set in a resource file, using the XmNdefaultVirtualBindings resource. The resource can be specified for all applications or on a per-application basis. To map the DELETE key to osfBackSpace, use the following specification:

```
*defaultVirtualBindings: \
    osfBackSpace: <Key>Delete \n\
    other bindings
```

The only difference between the syntax for the resource specification and for the *.motifbind* file is that the resource specification must have a newline character (\n) between each entry. The complete syntax of Motif virtual bindings is explained in Volume 6B, *Motif Reference Manual*.

Motif a client, *xmbind*, that configures the virtual key bindings for Motif applications. This action is performed by the Motif Window Manager (*mwm*) or any application that uses the Motif toolkit at startup, so you really only need to use *xmbind* if you want to reconfigure the bindings without restarting *mwm* or a Motif application. Motif also provides a function, XmTranslateKey(), to translate a keycode into a virtual keysym. This function allows applications that override the default XtKeyProc to handle Motif's virtual key bindings.

Callbacks

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

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

Figure 2-3 illustrates the event-handling path that results in an application callback being invoked. The widget's translation table registers the widget's interest in a particular type of event. When Xt receives an event that happened in the widget's window, it tests the event against the translation table. If there is no match, the event is thrown away. If there is a match, the event is passed to the widget and an action routine is invoked. The action routine may perform a function internal to the widget, such as changing the widget's appearance by highlighting it. Depending on the design of the widget, the action routine may then pass control to an application callback function. If the action is associated with a callback resource, it checks to see if a callback function has been registered for that resource, and if so, it dispatches the callback.

There are several ways to connect an application function to a callback resource. The most common is to call `XtAddCallback()`, as demonstrated in *hello.c*:

```
void button_pushed(Widget, XtPointer, XtPointer);
...
XtAddCallback (button, XmNactivateCallback, button_pushed, NULL);
```

The first argument specifies the widget for which the callback is installed. The second parameter is the name of the callback resource, while the third is a pointer to the callback function. The fourth argument is referred to as *client data*. If this parameter is specified, its value is passed to the callback function when it is called. Here, the client data is `NULL`.

The client data can be a value of any type that has the same size as an `XtPointer`.An `XtPointer` is usually the same as a `char` pointer; it is typically represented by a 32-bit value. You can pass pointers to variables, data structures, and arrays as client data. You cannot pass actual data structures; the result of passing a data structure is undefined. You can pass variables of type `int` or `char`, but understand that you are passing the data by value, not by reference. If you want to pass a variable so that the callback routine can change its value, you must pass the address of the variable. In this case, you need to make sure that the variable is global, rather than local, since a local variable loses its scope outside of the routine that calls `XtAddCallback()`.

The callback function itself is passed the widget, the client data, if any, and a third argument that is referred to as *call data*. The signature of a callback function can be expressed in one of two ways: using an ANSI-compliant function prototype or using the older style conventions of K&R C. The ANSI-style function declaration is as follows:

```
void button_pushed (Widget widget, XtPointer client_data, XtPointer call_data)
```

In the strictest sense, declaring the types of the parameters to the function is the proper way to handle function declarations and signatures.

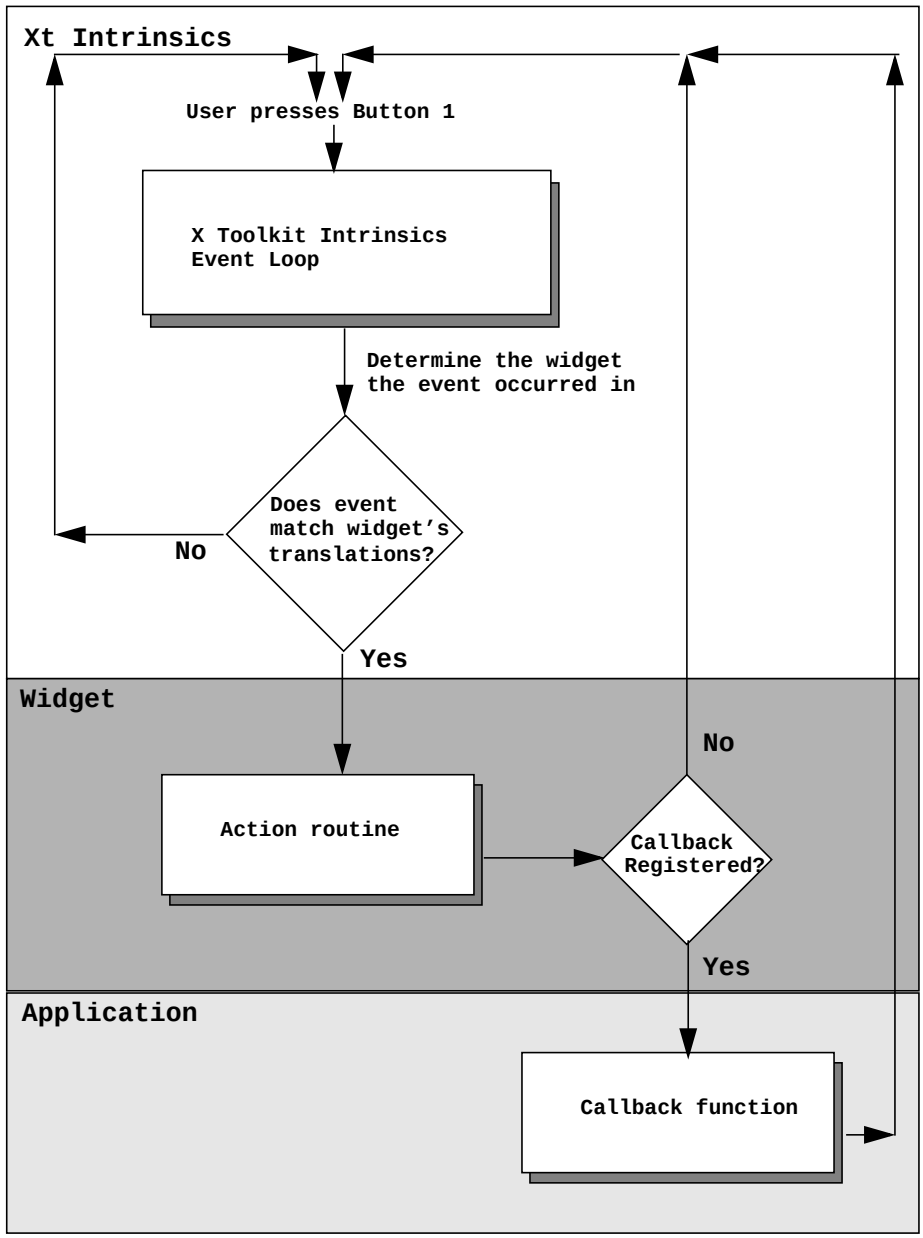


Figure 2-3 Event handling using action routines and callbacks

While this convention is good style and recommended for upwards compatibility, most compilers today still understand the older style conventions:

```
void button_pushed (widget, client_data, call_data)
        Widget         widget;
        XtPointer      client_data;
        XtPointer      call_data;
```

The second style is potentially the more portable method, although it is extremely difficult to think of any operating system vendors whose compiler is not ANSI aware. In the course of the book, we make a habit of declaring `client_data` and `call_data` as `XtPointers`, even though we usually know the actual types of the parameters being passed to the function. Before referencing these parameters, we cast the values to the appropriate types.

The third parameter in a Motif-based callback function is always a structure that contains information specific to the widget class that invoked the callback function, as well as information about the event that triggered the callback. There is a generic callback structure, `XmAnyCallbackStruct`, as well as variations for specific widget classes and callback resources. The `XmAnyCallbackStruct` is defined as follows:

```
typedef struct {
    int          reason;
    XEvent       *event;
} XmAnyCallbackStruct;
```

The callback structure for the Motif PushButton widget class, the `XmPushButtonCallbackStruct`, is defined as follows:

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

We discuss the callback structures for a widget class in this book (see the chapter corresponding to the specific widget type). The callback structures are also documented in the widget reference pages in Volume 6B, *Motif Reference Manual*.

All of the callback structures contain at least the two fields found in `XmAnyCallbackStruct`. The `reason` field always contains a symbolic value that indicates why the callback was called. These values are defined in */usr/Motif2.1/include/Xm/Xm.h* and are usually self-explanatory. For example, when a callback function associated with a PushButton's `XmNactivateCallback` resource is called, the `reason` is `XmCR_ACTIVATE`. The different values for `reason` make it easier to write callback routines that are called by more than one type of widget. By testing the `reason` field, you can determine the appropriate action to take in the callback. Because the widget is always passed to the callback function, you can also find out what widget caused the function to be invoked.

The `event` field contains the actual event that triggered the callback, which can provide a great deal of useful information. See Volume 4, *X Toolkit Intrinsics Programming Manual*, for information on how to interpret the contents of an event. That subject is not discussed at length in this book, although our examples frequently use the events in callback structures to control processing.

The Event Loop

Once all of the widgets for an application have been created and managed and all of the callbacks have been registered, it's time to start the application running. The final two function calls in *hello.c* perform this task:

```
XtRealizeWidget (toplevel);
XtAppMainLoop (app);
```

Realizing a widget creates the actual window for the widget. When you call `XtRealizeWidget()` on the top-level widget of an application (the one returned by the call to `XtVaOpenApplication()`), Xt recursively traverses the hierarchy of widgets in the application and creates a window for each widget. Before this point, the widgets existed only as data structures on the client side of the X connection. After the call, the widgets are fully instantiated, with windows, fonts, and other X server data in place. The first `Expose` event is also generated, which causes the application to be displayed.

The call to `XtAppMainLoop()` turns control of the application over to the X Toolkit Intrinsics. Xt handles the dispatching of events to the appropriate widgets, which in turn pass them to the application via callbacks. The application code is idle until summoned to life by user-generated events.

Summary

We've looked at the skeleton of a simple Motif program. Every application follows more or less the same plan:

1. Initialize the X Toolkit Intrinsics.
2. Create and manage widgets.
3. Configure widgets by setting their resources.
4. Register callbacks to application functions.
5. Realize the widgets and turn control over to Xt's event loop.

How this skeleton is fleshed out in a real application is the subject of the next chapter. Chapter 3, *Overview of the Motif Toolkit*, addresses the role of manager widgets in laying out a user interface, the use of dialog boxes and other popups for transient interactions with the user, the many specialized types of widgets available in Motif, and other essential

concepts. Once you have read that chapter, you should have a sufficient foundation for reading the remaining chapters in any order.

In this chapter:

- *The Motif Style*
- *Application Controls*
- *Application Layout*
- *Putting Together a Complete Application*
- *Changes in Motif 2.1*
- *Summary*

3

Overview of the Motif Toolkit

This chapter helps the reader understand the components of a real Motif application. It discusses how to handle the geometry management of primitive widgets within a manager widget, when to put components into the main window, when to use dialog boxes and menus, and how to relate to the window manager. After reading this chapter, the programmer should have a solid overview of Motif application programming, and she should be able to read the remaining chapters in any order.

In Chapter 2, *The Motif Programming Model*, we talked about the basic structure of an Xt-based program. We described how to initialize the toolkit, create and configure widgets, link them to the application, and turn control over to Xt's main loop. In this chapter, we discuss the widgets in the Motif toolkit and how you can put them together to create an effective user interface for an application.

If you already have a basic understanding of the Motif widgets, you can jump ahead to any of the later chapters in the book that focus on individual widget classes. This chapter provides some insight into the design of the widgets and a general overview of the Motif style and methodology, which you may find useful when developing your own applications.

This chapter also describes all of the new features in Release 2.1 of Motif. If you are familiar with Motif 1.2 but need to get up to speed with Motif 2.1, you should read Changes in Motif 2.1. In this section, we summarize the new features and tell you where to find more information about them.We also describe all the changes made to the example programs in this book to make them up-to-date with Motif 2.1. While Motif 2.1 is backwards-compatible with Motif 1.2, there are a number of functions and resources in Motif 2.1 that replace obsolete functions and resources in Motif 1.2.

The Motif Style

You don't build a house just by nailing together a bunch of boards; you have to design it from the ground up before you really get started. Even with a prefabricated house, where

many of the components have already been built, you need a master plan for putting the pieces together. Similarly, when you are designing a graphical user interface for an application, you have to think about the tasks your application is going to perform. You must envision the interface and then learn to use your tools effectively in order to create what you've envisioned.

The Motif toolkit provides basic components that you can assemble into a graphical user interface. However, without design schematics, the process of assembling the user-interface elements may become ad hoc or inconsistent. Here is where the *Motif Style Guide* comes in. It presents a set of guidelines for how widgets should be assembled and grouped, as well as how they should function and interact with the user.

All Motif programmers should be intimately familiar with the *Style Guide*. While we make recommendations for Motif style from time to time, this book is not a replacement for the *Style Guide*. There are many aspects of Motif style that are not covered in detail here, as they involve the content of an application rather than just the mechanics. On the other hand, the *Motif Style Guide* is not an instructional manual for the Motif toolkit. In fact, many of the objects described in the *Style Guide* are not even widgets, but higher-level, more complex objects that are composed of many widgets.

For example, the *Style Guide* describes an object called a MenuBar, which spans the top of the main window of an application. The MenuBar contains menu titles that, when clicked on, display PulldownMenus. The Motif toolkit does not implement MenuBars or PulldownMenus as distinct widget classes, nor does the *Style Guide* make any recommendations about how menu objects should be implemented. What the *Style Guide* does talk about (albeit somewhat loosely) are the actions that can be taken by an item on a menu: it can invoke an application function, pop up a dialog box containing yet more options and commands, or display a cascading menu (also known as a pullright menu).

The *Style Guide* also makes recommendations about the menus that an application should provide. For example, most applications should have a *File* menu that provides items such as an *Exit* button to exit the application and a *Save* button to save file. It also specifies details of presentation, such as that you should provide an ellipsis (...) as part of the label for a menu item that requires the user to provide more information before action is taken.

How the Motif toolkit goes about supporting, and in some cases enforcing, the guidelines of the *Motif Style Guide* brings up some interesting points, particularly in relation to some of the underlying principles of the X Toolkit Intrinsics. In Xt, a widget is envisioned as a self-contained object that is designed to serve a specific, clearly-defined function. Many of the Motif widgets, such as Labels, PushButtons, ScrollBars, and other common interface objects, are implemented as separate widgets.

In other cases, however, Motif steps outside of the Xt model by creating compound objects out of several widgets and then expecting you to treat them as if they were a single object. For example, Motif provides the ScrolledText and ScrolledList objects, which combine a

Text or List widget with a ScrolledWindow widget, which in turn automatically manages horizontal and vertical ScrollBars.

In another case, the Motif toolkit provides a complex, general-purpose widget that can be configured to appear in several guises. There is no MenuBar widget class and no PulldownMenu widget class. Instead, the RowColumn widget, which also serves as a general-purpose manager widget, has resources that allow it to be configured as either a MenuBar or a PulldownMenu pane. Those familiar with Xt may find this widget design to be a breach of Xt's design goals, though.

In order to allow the programmer to think of ScrolledText objects, MenuBars, and PulldownMenus as distinct objects, the Motif toolkit provides convenience creation functions. These routines make it appear as though you are creating discrete objects when, in fact, you are not. For example, the toolkit functions `XmCreateMenuBar()` and `XmCreateSimplePulldownMenu()` automatically create and configure a RowColumn widget as a MenuBar and a PulldownMenu, respectively. There are also convenience routines for creating various types of predefined dialog boxes, which are actually composed of widgets from four or five separate widget classes.

Convenience routines emphasize the functional side of user-interface objects while hiding their implementation. However, since Motif is a truly object-oriented system, it behoves you to understand what you're really dealing with. For example, if you want to use resource classes to configure all MenuBars to be one color and all PulldownMenus another, you cannot do so because they are not actually distinct widget classes. The class name for both objects is `XmRowColumn`.

In the remainder of this chapter, we look at Motif user-interface objects from the perspective of both the functional object illusion and the actual widget implementation. In the body of the book, we use the Motif convenience routines for creating both compound objects, and simple widgets or gadgets. With the compound objects, we show you how to pierce the veil of Motif's convenience functions and work directly with the underlying widgets when necessary.Figure 3-1 shows the entire class hierarchy of the Motif widget set.

We begin by taking a closer look at the Motif user-interface components with which the user typically interacts. Then we examine how the manager widget classes are used to arrange the more visible application controls. And finally, we explore the use of all of these objects to create functional windows and dialogs that make up a real application.

.

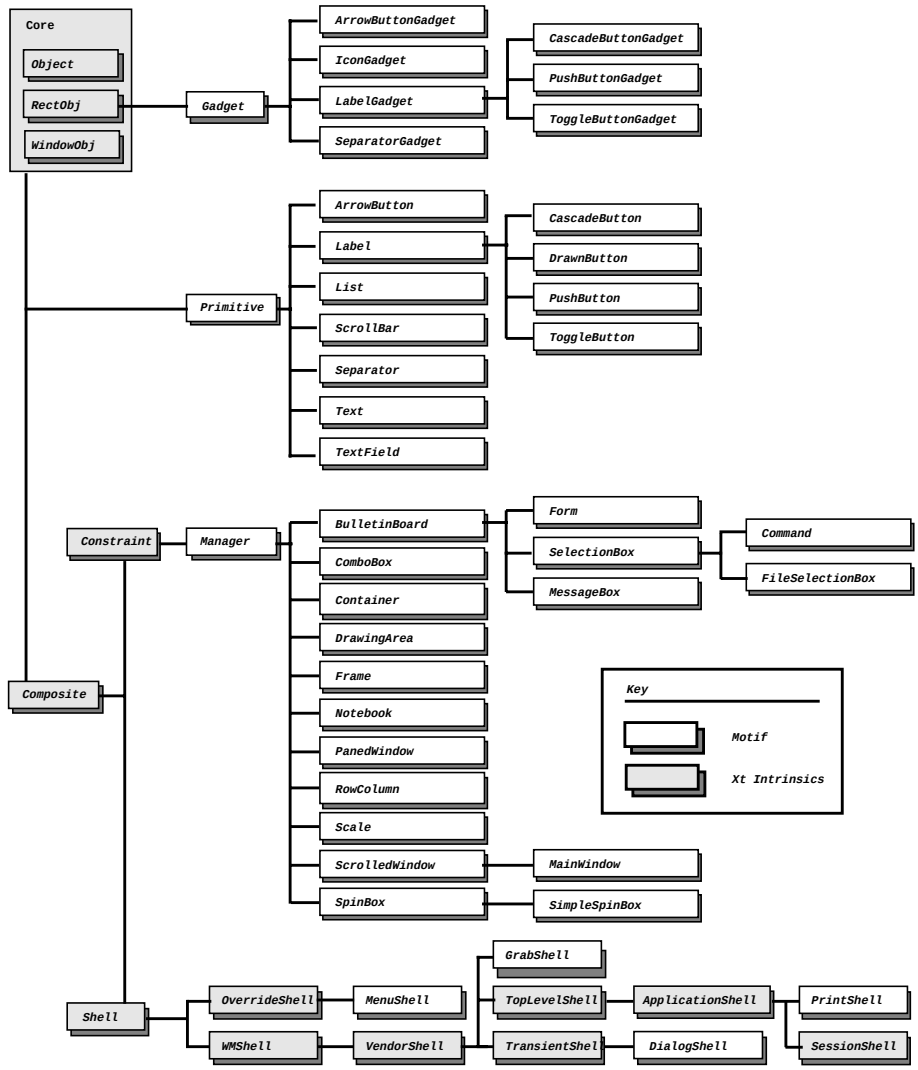


Figure 3-1 Class Hierarchy of the Motif widget set

Application Controls

In many ways, application controls are the heart of a graphical user interface. Rather than controlling an application by typing commands, the user is presented with choices using graphical elements. The user no longer needs to remember the syntax of commands, since her choices are presented to her as she goes along. As we've discussed, some of Motif's application controls (such as menus) are compound objects assembled by convenience routines. Others are simple, single-purpose widgets that you can create directly.

The widgets in this latter group are collectively referred to as *primitive* widgets -- not because they are simple, but because they are designed to work alone. The contrast is not between primitive and sophisticated widgets, but between primitive and manager widgets. Some of the primitive Motif widget classes have corresponding gadget classes. The following sections describe the different types of primitive application controls available in the Motif toolkit.

The compound objects in the Motif toolkit are composed of primitive widgets and gadgets. Because an understanding of these objects relies on an understanding of the primitive widgets, as well as the Motif manager and shell widgets, we are going to postpone discussing compound objects until later in the chapter.

The Primitive Widget Class

The Primitive widget class is a superclass for all of the Motif primitive widgets. This widget class is a metaclass; it serves only to define certain common behavior used by all its subclasses, so one never instantiates a widget directly from the Primitive class. This statement is somewhat like saying that hammer is a class of object, but that you never really have a generic hammer. You can only have a specific type of hammer, like a claw hammer, a ball peen hammer, or a sledge hammer.*

Just as all hammers have particular characteristics that qualify them as hammers, the Primitive widget class provides its subclasses with common resources such as window border attributes, highlighting, and help with keyboard traversal (so the user can avoid the mouse and navigate through the controls in a window using the keyboard). The actual widget classes that you use are subclassed from the Primitive class, as shown in Figure 3-2.

* A claw hammer has the prongs in the back behind the hammer-head that allow you to pull nails out of a wall; a ball peen hammer has a round corner where the claw would be otherwise be; a sledge hammer is the large, heavyweight hammer used to drive thick nails through concrete or to destroy things.

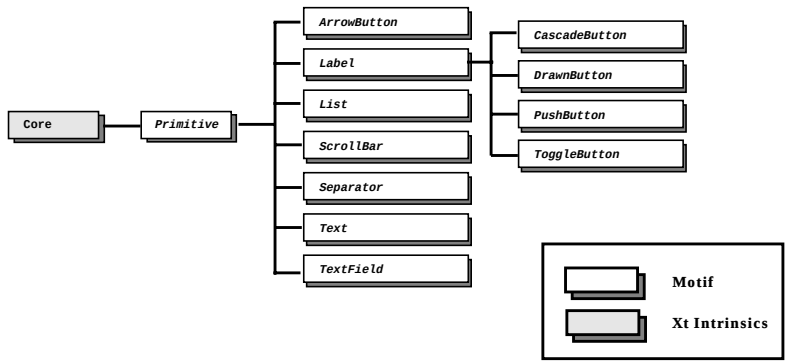


Figure 3-2 The Primitive widget class hierarchy

The Primitive class itself inherits even more basic widget behavior from the Xt-defined Core widget class, which establishes the basic nature of "widgetness." The Core class provides widgets with the capability to have windows and background colors, as well as translations, actions, and so on. You could actually use a simple Core widget as an instance and define your own translations and action routines, although this technique is not used frequently. Complete details are provided in Volume 4.

The Label Class

The Label widget provides a visual label either as text or as an image in the form of a `Pixmap`.The text of a Label is an `XmString`, or compound string, not a character string (`char *`).A compound string can be oriented from left-to-right or right-to-left and it can also contain multiple lines and multiple fonts. Chapter 25, *Compound Strings*, discusses functions that manipulate compound strings, as well as functions that convert between character strings and compound strings.

The Label widget does not provide any callback routines, since it does not have any specified behavior. Using Xt, you could install event translations and action routines to make a Label respond to user input, but the Label widget is not intended to be used this way. It is only meant to be used to display labels or other visual aids. In Motif, instances of Label and all of its subclasses are automatically registered as drag sources for drag and drop operations by the toolkit*.

Label widgets are described in detail in Chapter 12, *Labels and Buttons*. Figure 3-3 displays a single Label widget with multiple lines and multiple fonts.

* In fact, in Motif 2.1, drag and drop for a Label, LabelGadget, or Scale may be disabled by default if the resource `XmNenableUnselectableDrag` is False. See the section on XmDisplay in Volume 6B for more details.

Figure 3-3 A Label with multiple lines and fonts

The PushButton Class

The PushButton widget supports the same visual display capabilities as a Label, since it is subclassed from Label. In addition, the PushButton provides resources for the programmer to install callback routines that are called when the user arms, activates, or disarms the button. The PushButton also displays a shadow border that changes in appearance to indicate when the pointer is in the widget and when it has been activated.

When a PushButton is not selected, it appears to project out towards the user. When the pointer moves into the button, its border is highlighted. When the user actually selects the button by pressing the first mouse button on it, the button appears to be pushed in and is said to be armed. The user activates a PushButton by releasing the mouse button while the button is armed. PushButton widgets are also covered in detail in Chapter 12, *Labels and Buttons*. Figure 3-4 shows some examples of PushButtons.

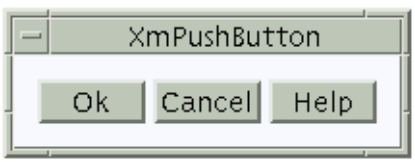


Figure 3-4 PushButton widgets

The DrawnButton Class

The DrawnButton widget is similar to a PushButton in its functionality and its three-dimensional appearance. However, the DrawnButton is used when an application wants to draw the text or image directly into the widget's window, rather than have the widget handle the drawing. If the image is dynamic and changes frequently during the course of an application, you may want to handle the drawing yourself. The DrawnButton provides additional callback resources that are called when the button is resized or exposed and additional ways to draw an outlined border. The DrawnButton widget is discussed in Chapter 12, *Labels and Buttons*. Figure 3-5 shows some DrawnButtons.

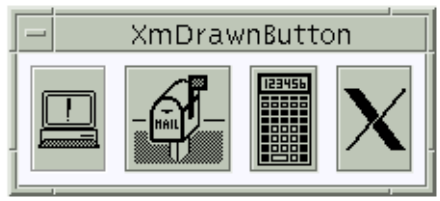


Figure 3-5 DrawnButton widgets

The ToggleButton Class

The ToggleButton widget displays text or graphics like a Label widget, but it has an additional indicator graphic (a square, diamond, and additionally in Motif 2.1, a circle or check mark shape) to the side of the label. The indicator shows the state of the ToggleButton: in Motif 1.2 this could be simply on or off; in Motif 2.1 a toggle can exist in a third *indeterminate* state. When the ToggleButton is on, the indicator is colored and appears to be pushed in. When the button is off, the indicator appears to project outward. In the indeterminate state, the toggle is half colored, half uncolored. The ToggleButton provides an additional resource for specifying a callback routine that is called when the user changes the state of the ToggleButton.

One common use of ToggleButtons is to set the application state. In this case, the callback routines typically set simple `Boolean` variables internal to the application. ToggleButtons can also be arranged in two different kinds of groups. In one configuration, known as a RadioBox, only one button in the group of buttons can be chosen at a time. The other configuration, a CheckBox, allows the user to select any number of buttons. When ToggleButtons are grouped as a RadioBox, the indicators are by default diamond-shaped; otherwise, they default to a square-shaped appearance. ToggleButton widgets are described in detail in Chapter 12, *Labels and Buttons*. Figure 3-6 shows the two different ways that ToggleButtons can be grouped.

CheckBox **RadioBox**

Figure 3-6 ToggleButton widgets

The CascadeButton Class

The CascadeButton widget is a special kind of button that is used to popup menus. A CascadeButton can only be used as a child of a RowColumn widget, such as: in a MenuBar as the title of a PulldownMenu, in a PulldownMenu pane as an item that has a cascading menu associated with it, or as the button in an OptionMenu. The menu that is posted by a CascadeButton is not a part of the widget itself; the menu is associated with the button through a resource. A CascadeButton merely provides the label and other visual aids that support the appearance that a menu can pop up from the object. Even though the CascadeButton widget class is subclassed from Label and could inherit all of its functionality, Motif imposes restrictions on the labels that a CascadeButton can display. CascadeButton labels cannot contain multiple lines or multiple fonts. Because CascadeButtons are typically used in menus, they do not display border shadows like other buttons. They do have similar highlighting behavior when selected, however. CascadeButton widgets are explained in both Chapter 4, *The Main Window*, and Chapter 20, *Interacting With the Window Manager*.

The ArrowButton Class

Despite the similarity in its name, the ArrowButton widget is not subclassed from Label like the other button widgets. Like the remaining widgets described in this section, it is subclassed directly from the Primitive widget class. The ArrowButton widget contains an image of an arrow pointing in one of four directions: up, down, left, or right. When the user selects this widget, the ArrowButton provides visual feedback giving the illusion that the button is pressed in and invokes a callback routine that an application can use to perform application-specific positioning.

In most respects, an ArrowButton can be considered identical to a PushButton, as it is easy enough to provide an arrow pixmap for a PushButton. Since directional arrows are a common user-interface element, the ArrowButton is provided as a separate widget class for simplicity. ArrowButton widgets are covered in detail in Chapter 12, *Labels and Buttons*. Figure 3-7 shows the four variations of the ArrowButton widget.

Figure 3-7 ArrowButton widgets

The List Class

The List widget provides a mechanism for the programmer to make a list of text items available to the user for selection. The user selects items from a List using the mouse or the keyboard. The List widget allows you to specify whether the user can select a single item or multiple items. While List is a Primitive widget, it is typically created as part of a ScrolledList compound object using a Motif convenience function. The advantage of the ScrolledList object is that it provides a ScrollBar when the List grows bigger than the size of its visible area. Instances of the List widget are automatically registered as drag sources for drag and drop operations by the toolkit. We explore the List widget in detail in Chapter 13.

Figure 3-8 shows a List widget in context with other interface elements.

Figure 3-8 A List widget in an application dialog

The ScrollBar Class

The ScrollBar widget is one of the more intuitive user-interface elements in the Motif toolkit. ScrollBars are almost always used as children of a ScrolledWindow widget. When the contents of a window are larger than the viewing area, a ScrollBar allows the user to scroll the window to view the entire contents.

ScrollBars can be oriented vertically or horizontally. The ScrollBar also provides a number of callback resources that allow you to control its operation. ScrollBar widgets are discussed in Chapter 10, *ScrolledWindows and ScrollBars*. Figure 3-9 shows both vertical and horizontal ScrollBars.

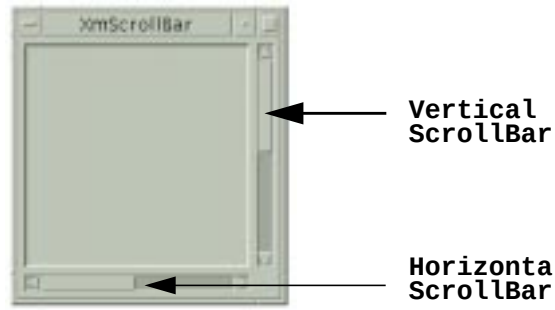


Figure 3-9 ScrollBars

The Separator Class

The Separator widget is used as a visual aid to separate adjacent items in a display. A Separator appears as a line between the objects it is separating; it can be oriented vertically or horizontally. Separators can be used in menus to separate menu items, in dialog boxes to separate discrete areas of control, and at various points in an interface for purely aesthetic reasons.

The Text and TextField Classes

The Text widget is a complete text editor contained in a widget. The Text widget provides resources to configure the editing style of the widget, as well as callback resources that allow text verification. The widget can be configured as a multiline text entry area or as a single-line data entry field. The TextField widget class is available as a somewhat lighter-weight text entry area. The TextField widget is limited to a single-line, but in all other respects there is little difference between the two classes. Instances of the Text and TextField widgets are automatically registered as drag sources and drop sites for drag and drop operations by the toolkit.

The Text and TextField widgets can be used in many different ways to support the text entry requirements of an application. The two widgets are described in detail in Chapter 18, *Text Widgets*.Figure 3-10 shows an application that uses various forms of the Text widget.

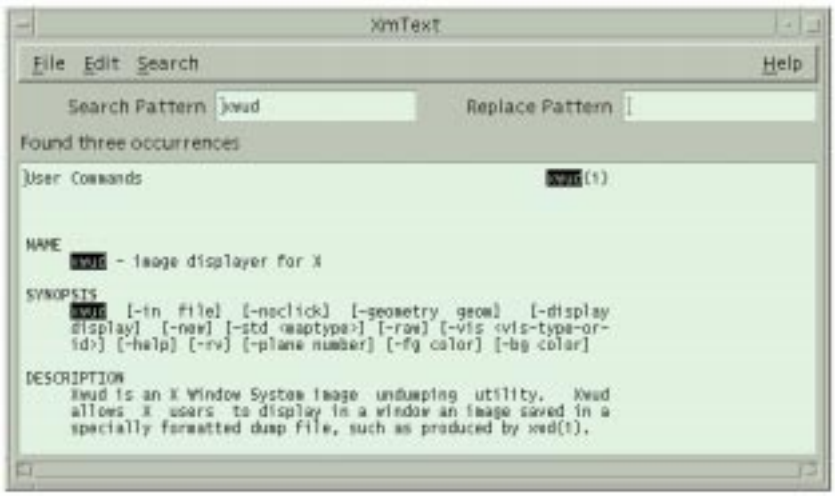


Figure 3-10 Text Widgets

Gadgets

Another set of application controls is provided in the form of gadgets. There are gadgets that are equivalent to many of the primitive widgets: ArrowButtonGadgets, SeparatorGadgets, PushButtonGadgets, CascadeButtonGadgets, ToggleButtonGadgets, LabelGadgets, and in Motif 2.1, IconGadgets. The IconGadget is similar to a LabelGadget, except that it can display a label and an image simultaneously. The appearance and behavior of the gadgets are mostly identical to that of the corresponding widgets*. A further understanding of how gadgets work depends on an understanding of the manager widgets that support them, so we are going to return to this topic later in the chapter.

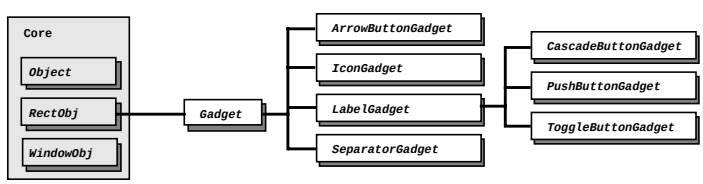


Figure 3-11 The Gadget class hierarchy

* The IconGadget is exceptional: there is no widget equivalent to this gadget class.

The Gadget class is a superclass for all of the Motif gadgets. Like Primitive, this class is a metaclass that is never instantiated. However, gadgets are not widgets. The Gadget class is subclassed from the RectObj class, not from the Core widget class. Figure 3-11 shows the class hierarchy for gadgets.

Application Layout

While the controls are the most obvious part of a graphical user interface, these elements alone do not make an effective interface. A random arrangement of buttons or a collection of nested menus can make an application as obscure and as difficult to use as one with a command-line interface. The arrangement of the controls in an application makes all the difference.

To help you lay out your application, Motif provides you with a set of manager widgets. You can think of manager widgets as boxes in which you can put things. These boxes, however, can grow or shrink as necessary to provide the best fit possible for the items that they contain. You can place boxes inside of other boxes, whether or not they contain other items. By using different size boxes, you can organize things in many different ways.

Manager widgets are so named because they manage the size and position of other widgets. The relationship between a manager widget and the widgets that it manages is commonly referred to as the *parent-child* model. The manager acts as the parent, and the other widgets are its children.

Unlike primitive widgets, such as PushButtons, ScrollBars, and Labels, whose usefulness depends on their visual appearance and interaction with the user, manager widgets provide no visual feedback and have few callback routines that react to user input. Manager widgets have two basic purposes: they manage the sizes and positions of their children, and they provide support for gadgets. Like other widgets, manager widgets have windows, they can receive events, and they can be manipulated directly with Motif and Xt functions. You can draw directly into the window of a manager widget, look for events in the widget, and specify resources for it.

There are many manager widget classes, each of which is tuned for a particular kind of widget layout. A manager widget can manage other manager widgets, as well as primitive widgets like Labels and PushButtons. In fact, the layout of an application is typically a kind of tree structure. As discussed in Chapter 2, the top of the tree is always a shell widget like that returned by `XtVaAppInitialize()`. Shell widgets are composite widgets that can only have a single managed child. This child is usually a general-purpose manager widget. This manager contains other managers and the primitive widgets that compose the user interface for a window in an application.

Figure 3-12 shows all of the different manager and primitive widgets that make up the displayed dialog box.

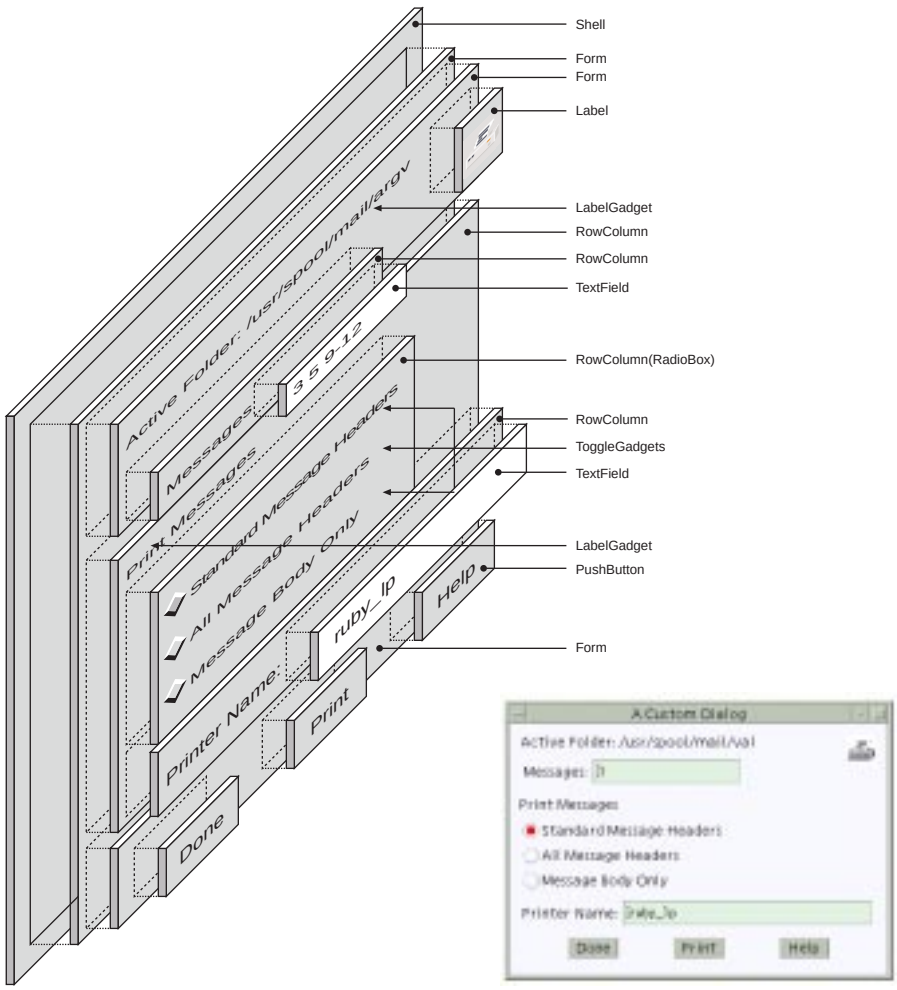


Figure 3-12 The layout of a dialog box

The parent-child relationships between the widgets in this dialog box are illustrated in the tree structure shown in Figure 3-13.

Although the dialog box is composed of many different components, it appears to the user as a single, conceptually focused user-interface object.

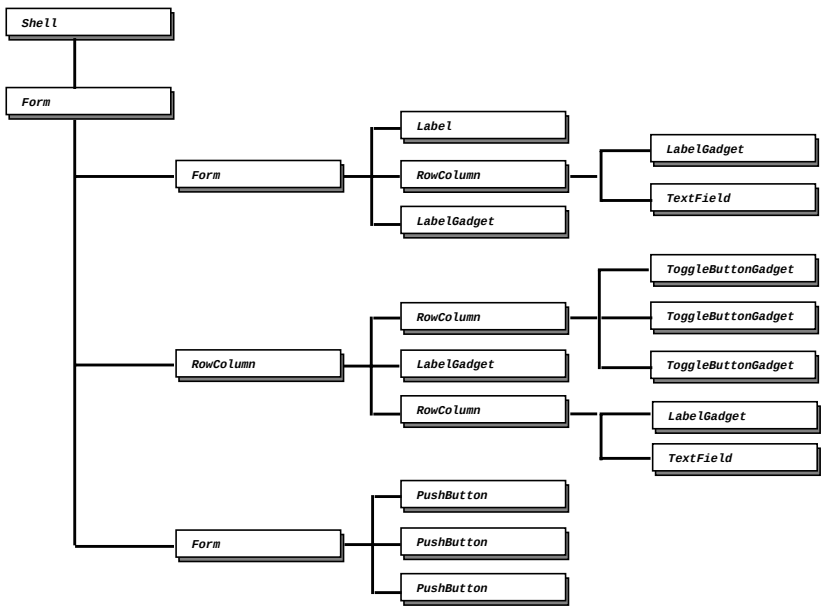


Figure 3-13 Parent-child relationships between widgets

The Manager Widget Class

As with the Primitive widget class and the Gadget class, the Manager widget class is a superclass for all of the Motif manager widgets. The Manager class is another metaclass. You never create an instance of a Manager widget; you create an instance of one of its subclasses. The actual widget classes that you use are shown in Figure 3-14.

Manager is subclassed from the Xt Constraint class, which in turn is subclassed from the Xt Composite class. The Composite widget class defines the basic characteristics of widgets that are able to manage the size and position of other widgets. Xt uses the general term *composite widget* for any widget with this capability. The Constraint class adds the capability to provide additional resources for the widgets that are being managed. These resources constrain the position of the widgets. They can be thought of as hints about how the widgets should be laid out.

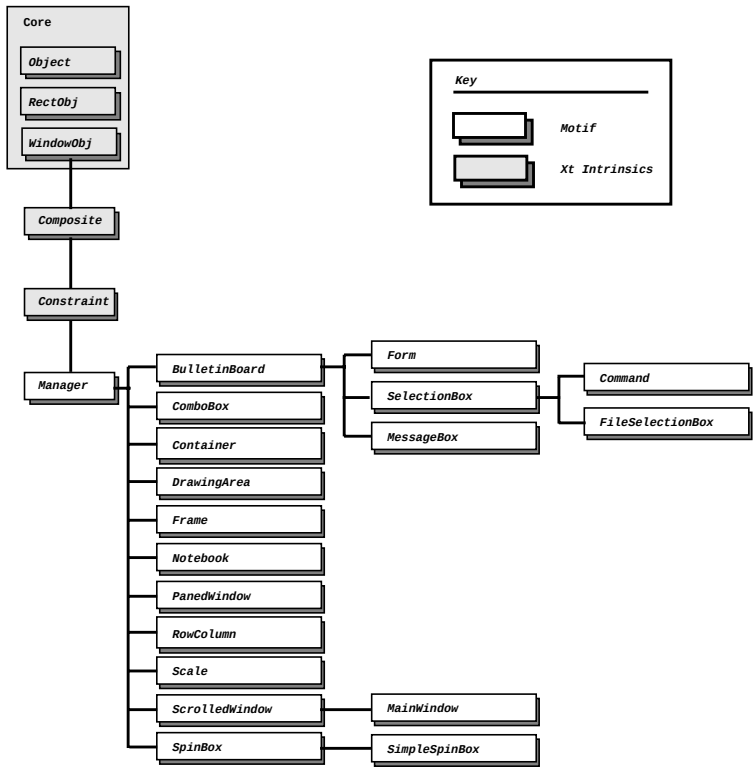


Figure 3-14 Class hierarchy of the Manager widget classes

Motif provides a number of general-purpose manager widgets that allow the programmer to manage the size and arrangement of an arbitrary number of children. In some ways, the art of Motif programming is the design of effective widget layouts, using these particular manager widgets. Motif also provides some narrowly-focused manager widgets, such as certain dialog classes, that can almost be treated as if they were single user-interface components. These widgets create and manage their children with minimal help from an application.We sometimes refer to these widgets as compound objects, since they include both a manager widget and one or more children. This section describes the different manager widgets briefly; a more detailed description of the widgets is given in Chapter 8, *Manager Widgets*.

The DrawingArea Class

The DrawingArea widget provides an area in which an application can display graphics. Callback routines can be used to notify the application when expose and resize

events take place and when there is input from the keyboard or mouse. The DrawingArea can also be used to manage the geometry layout for child widgets, but its functionality in this area is quite limited.The DrawingArea is discussed in detail in Chapter 11, *The DrawingArea Widget*.

The ScrolledWindow Class

The ScrolledWindow widget provides a viewport for data such as text or graphics. If the data that is being viewed is larger than the ScrolledWindow, ScrollBars allow the user to view the entire contents of the window interactively. The ScrolledWindow is discussed in Chapter 10, *ScrolledWindows and ScrollBars*.

The MainWindow Class

The MainWindow widget acts as the standard layout manager for the main window of an application. It is specifically tuned to pay attention to the existence of a MenuBar, a command area, a message area, a work region, and ScrollBars, although all of these areas are optional. The MainWindow is discussed in Chapter 4, *The Main Window*.

The RowColumn Class

The RowColumn widget is perhaps the most widely used and robust of all of the manager widgets. As its name suggests, the widget lays out its children in rows and columns. The RowColumn widget is used by many different parts of the toolkit to implement compound objects like MenuBars, PulldownMenus, CheckBoxes, and RadioBoxes. The general purpose RowColumn is discussed in Chapter 8, *Manager Widgets*.

The Frame Class

The Frame widget provides a three-dimensional border for a widget that does not normally have a border. It can also be used to enhance the style of the border for a widget that already has a border. In Motif, a Frame widget can have two children: a work area and a title. The work area child can be a manager widget that contains many other children. The Frame is discussed in Chapter 8, *Manager Widgets*.

The PanedWindow Class

The PanedWindow widget manages its children in a vertically (and, in Motif 2.1, a horizontally) tiled format. Its width always matches the widest widget in its list of managed children; the widget forces all of its children to stretch to the same width as that widget. Each pane in a PanedWindow contains a child widget; every pane has an associated sash (or grip) that allows the user to change the height of the pane interactively. Resizing a pane with the grip can cause the widgets in other panes to change size. The PanedWindow is discussed in Chapter 8, *Manager Widgets*.

The BulletinBoard Class

The BulletinBoard widget does not impose much of a layout policy for the widgets that it manages. The widget acts like a real bulletin board, in that an application pins a widget on the bulletin board, and it sticks where it is placed. The BulletinBoard does impose margins and has a resource that controls whether or not its children can overlap. However, when a BulletinBoard is resized, it does not move or resize its children based on its new size. The BulletinBoard is useful mostly for the layout of dialog boxes and other windows that are rarely resized. The predefined Motif dialog widget classes use BulletinBoard widgets for this reason. The BulletinBoard is discussed in Chapter 8, *Manager Widgets.*

The Form Class

The Form widget provides a great deal of control over the placement and sizing of the widgets it manages. A Form can lay out its children in a grid-like manner or it can allow its children to link themselves to one another in a chain-like fashion. Form uses constraint resources to specify how children are resized and positioned relative to each other and the Form as a whole. The Form is discussed in Chapter 8, *Manager Widgets*.

The Scale Class

The Scale widget is a slider object that is somewhat similar in appearance and functionality to a ScrollBar. A Scale is typically used to provide feedback to the user about the value of a state variable in an application. This widget class is not intended to be used as a general manager. The Scale creates and manages its own widgets, which are needed to construct the Scale object. The only children that you can add to a Scale widget are Label widgets that represent tick marks, although in Motif 2.1 there are convenience routines to automatically place tick marks along the Scale. The Scale is discussed in Chapter 16, *The Scale Widget*.

The following Manager widget classes are additionally available in Motif 2.1:*

The Container Class

The Container class is a complex constraint widget which can lay out IconGadget children in three styles: in a tree arrangement, with a tabular data style, and in a free floating format based upon the x, y specifications for each child. The Container class allows for a more object-oriented approach to the front end of an application than the older MainWindow, in that the IconGadget children can pictorially represent application objects of some kind with the Container providing the layout and selection mechanisms. The Container is discussed in Chapter 9, *Containers and IconGadgets*. Figure 3-15

* Available from Motif 2.0 onwards.

shows the Container configured to display in a tree arrangement with additional tabular data.

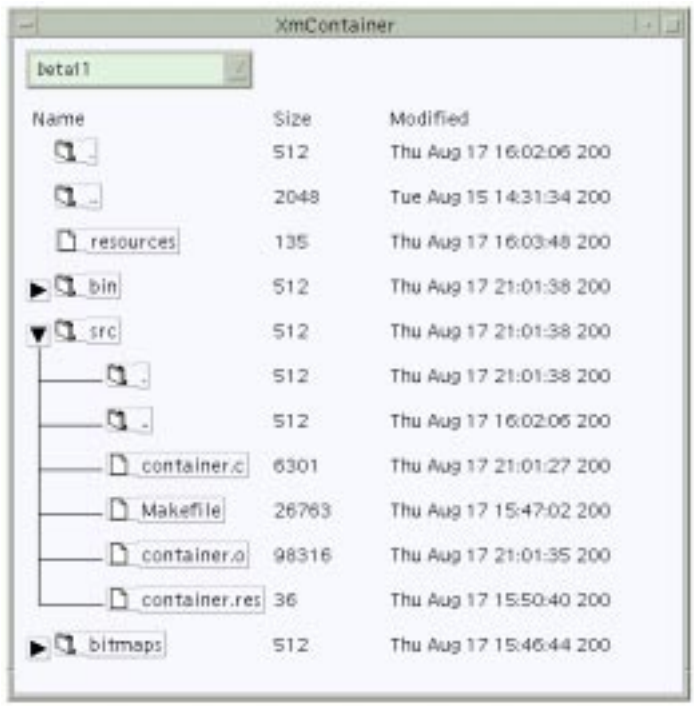


Figure 3-15 A Container widget with IconGadget children

The SpinBox Class

The SpinBox class allows the user to input data by selecting from, and rotating through, a set of values. Text widget children are added to the SpinBox, whereupon the range or set of values associated with each text is specified through constraint resources. The SpinBox automatically adds extra ArrowButtons which are used for rotating through the values of the text widget child which currently has the input focus. The programmer however has to supply the Text widgets underneath the SpinBox. For convenience, the SimpleSpinBox subclass is provided which encapsulates the most frequent use of this type of arrangement: it comes with a single built-in Text child. The SpinBox is discussed in Chapter 15, *The SpinBox and SimpleSpinBox Widgets*. Figure 3-16 shows a SpinBox containing three Label and Text children, and a SimpleSpinBox. The SimpleSpinBox is not meant to be used as a general purpose manager.

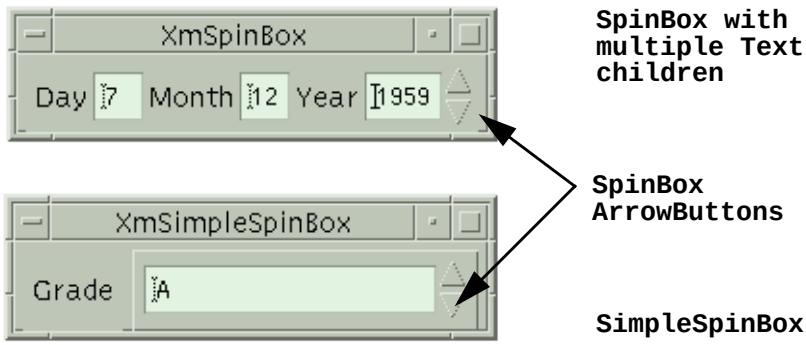


Figure 3-16 SpinBox and SimpleSpinBox widgets

The ComboBox Class

The ComboBox class combines textual input with list selection. The widget presents itself to the user as a Text widget with an ArrowButton to the side. The user can either type directly into the Text widget, or press the ArrowButton, when a list of items from which to choose is popped up immediately under the Text. Whether in fact the Text widget is directly editable, and whether the list of available options is permanently visible (as opposed to being displayed on user request by pressing the ArrowButtons) is controllable through resources when the ComboBox is created. This widget class is not intended to be used as a general manager. The ComboBox is discussed in Chapter 14, *The ComboBox Widget*. Sample ComboBoxes are shown in Figure 3-17.

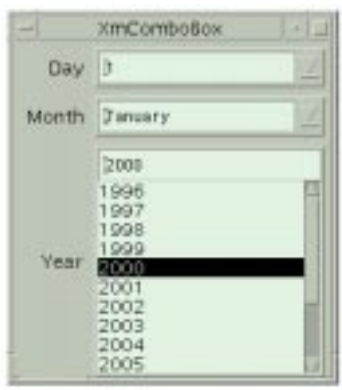


Figure 3-17 ComboBoxes with other widgets

The Notebook Class

The Notebook class lays out its children as though they are pages in a book. That is, only one child is currently visible at any given time, and they all occupy a single area on the screen; the user can chose from the available pages either by selecting from Tabs which can be associated with a child, or by activating the Page Scroller, which is typically a SpinBox. To complete the analogy, resources are provided to control the general book-like characteristics of the Notebook in terms of its binding and overlapping page appearance. The Notebook is a constraint widget: you add children, and then specify the role which each child is to perform. Typically, a Form or other manager is added to represent some page, and optionally PushButtons can be added and associated with a page in order to represent Tab inserts along the edges of the Notebook pages. The Notebook is discussed in Chapter 17, *The Notebook Widget*. Figure 3-18 shows a Notebook with Tabs inserted on the edge.

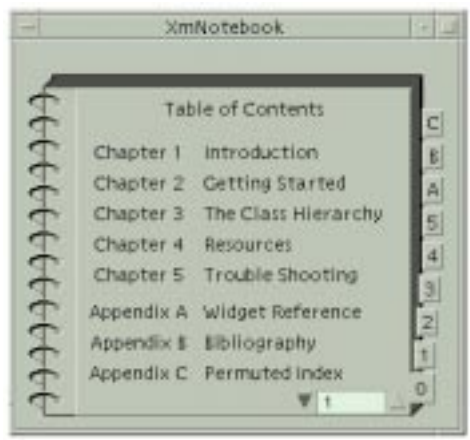


Figure 3-18 The Notebook widget

Geometry Management

The process by which a manager widget controls the layout of its children is known as *geometry management*. A child widget is always placed within the boundaries of its parent. A child cannot move or resize itself without requesting permission from its parent, which can deny the request. The manager, acting as the parent, can even force the child into an arbitrary size or position. However, like any good parent, a manager widget should be fair at all times and not deny reasonable requests made by its children. As you might expect, geometry management can be quite complex in an application with several levels of managers.

As an example, consider adding a new item to a List widget. In order to display the new item, the List widget must grow vertically, so it requests a new size from its manager parent. If that parent can accommodate the larger size, or it has another mechanism for satisfying the request, such as ScrollBars, it can approve the request. However, if the manager itself must grow to honor the List widget's request, it has to negotiate with its own parent. This chain reaction may go all the way up to the shell widget, in which case the shell must communicate with the window manager about the new size. If the window manager and the shell agree to the new size, the acknowledgement filters back down through the widget tree to the List widget, which can now grow to its requested size. If any of the composite widgets in the hierarchy refuse to resize, the List widget's request is either denied or only partially fulfilled.

Most of the time, this type of interaction completes successfully, as there are rarely disputes among children about resizing negotiations or positional boundaries. Children usually go where their managers put them and make very few requests of their own. One exception is a RowColumn widget that is acting as a MenuBar, since it must be situated at the top of the window, and it must span the window horizontally. ScrollBars are another possible exception, since they are typically positioned at the edges of ScrolledWindow widgets.

So, how do children request geometry changes from their parents? The answer to this question is rather complicated, since the X Toolkit Intrinsics supports a large selection of functions that enable two-way communication about geometry management. For example, a child can use `XtMakeGeometryRequest()` to request permission to be made a specific size or to be placed in a particular location. A parent can use a function like `XtQueryGeometry()` to give a child the opportunity to announce its preferred geometry.

Some of these functions and methods are described in Chapter 1, but a detailed treatment of custom geometry management techniques is beyond the scope of this book. These functions are mostly used by the internals of composite and constraint widgets. See Volume 4, for a more detailed discussion of geometry management techniques.

In the Motif toolkit, geometry management cannot work without cooperation. The easiest way for a child to cooperate with its parents and siblings is simply to comply with whatever layout policy is supported by its manager widget parent. A child should not try to force itself into a size or a position that is not supported by its parent. Each of the manager widget classes described above is designed to support a specific layout style. For example, the RowColumn widget lays out its children in rows and columns, the Form widget allows its children to specify positions relative to other widgets within the Form, and the PanedWindow widget lets its children specify their desired maximum and minimum heights.

Manager widgets use constraint resources to support their layout policies. Constraint resources are defined by Xt's Constraint widget class, which is a superclass for the Manager widget class and thus all of the Motif manager widgets. Unlike other resources,

constraint resources apply to the *children* of a manager widget, not to the manager itself. Examples of constraint resources include maximum and minimum heights, relative sizes and positions, specific positional constraints, and even absolute x, y coordinates. While these examples deal exclusively with size and position, constraint resources can be used for any arbitrary information that needs to be kept on a per-child basis.

Here's how constraint resources work. When a manager needs to size or position its children, it deals only with the children that are managed; unmanaged children are ignored in geometry management negotiations. For each managed child, the manager examines the child's constraint resources. Depending on the constraints that are specified, the manager either enforces the geometry changes or negotiates with its own parent to see if it can comply with the changes. This process uses an extra internal data structure for each child. The data structure stores the constraints that are used by the widget's parent to aid it in geometry management.

Gadget Management

In addition to handling geometry management, manager widgets are responsible for their gadget children. In order to understand how managers support gadgets, we need to define more clearly what a gadget is. Every widget has its own X window, which simplifies many aspects of programming, since each widget can take responsibility for repainting itself, selecting its own events, and in general being as self-sufficient as possible. Historically, however, windows have been perceived as heavyweight objects. The concern is that system performance will be degraded if an application uses too many windows. Since an application with a graphical user interface frequently uses hundreds of widgets, or perhaps even thousands for a very large program, the performance issue is an important one.

Gadgets, or windowless widgets, were originally developed as a part of Motif. They were added to Xt as of X11 Release 4. Motif provides gadget versions of many common primitive widgets, such as PushButtons and Labels. Like widgets, gadgets can be created using either Motif convenience functions or `XtCreateManagedWidget()`. While the widget and gadget versions of an object are functionally very similar, there are some small but important differences.

Because a gadget does not have its own window, it is entirely dependent on its parent, a manager widget, for its basic functionality. For example, the manager must handle redrawing the gadget on exposure, highlighting it as a result of keyboard traversal, and notifying it of event activity. Without a window, a gadget has no control over window-based attributes normally associated with a widget. For this reason, gadgets can only be used in managers that support them. How closely a gadget emulates its widget counterpart is largely dependent on the capabilities of the manager widget parent.

In Motif 1.2, the Manager class limits the colors that can be used by gadgets. A gadget uses the same background, foreground, and shadow colors as its manager widget parent. These

restrictions are not inherent in the Xt Composite widget class or in Xt-based gadgets; they are specific to the Motif 1.2 Manager and Gadget classes. It is possible to write a Composite widget that allows its gadget children to specify their own background colors. Such a widget would have to paint the area of its window occupied by the gadget with the specified color to give the user the impression that the gadget is indeed a separately-colored widget. Indeed, gadgets in Motif 2.1 have been redesigned with precisely this extra functionality.

Although gadgets were originally developed to improve performance, it is no longer necessary to automatically use them if you are looking for performance improvements in an application with many widgets. In both X11 Release 4 and Release 5, windows have become substantially lighter-weight objects than they were when gadgets were first developed. If anything, gadgets are worse than widgets at this point from a performance perspective because the Motif managers take a very simplistic approach to the way they handle events for gadgets. A manager tracks all events, even `MotionNotify`, whether or not its gadgets have expressed interest in the events. As a result, gadgets typically generate a great deal of network traffic. X terminal users are especially likely to notice a network performance drop. There are some other complications that surround the use of gadgets, which we discuss when they come up in the course of this book.

Keyboard Traversal

Keyboard traversal is a mechanism that allows a user to navigate through the components in a user interface using only the keyboard. The *Motif Style Guide* specifies that all applications must support keyboard traversal for all application functionality. Support of keyboard traversal is important because not every display provides a mouse or other pointing device. For some applications, such as data entry, using keyboard traversal is more convenient than using a pointing device. All of the Motif widgets support keyboard-based navigation.

Keyboard traversal is based on the concept of a *tab group*. A tab group is a group of widgets that are related for the purpose of keyboard traversal. For example, all the items in a menu are considered a tab group, since they are grouped together and perform related functions.

At any given time, only one component on a display can be “listening” to the keyboard for keyboard events. The widget that is listening to the keyboard is said to have the keyboard focus, or input focus. The widget that has the input focus identifies itself by displaying a location cursor. The location cursor is often a highlighted border that surrounds the widget. A user can move the input focus to another widget using the mouse or the keyboard.

The user can move the keyboard focus between items in the same tab group using the arrow keys. When the user finds the item that she wants, she can activate it with the RETURN key or the SPACEBAR. If the user wants to move from one tab group to another, she uses the TAB key. (In a multiline Text widget, CTRL-TAB is used because otherwise there would be no way to insert a tab character.) To traverse the tab groups in reverse, the SHIFT

key is used with the TAB key. Keyboard traversal wraps from the last item to the first item, both within a tab group and between tab groups.

Although keyboard traversal is not completely controlled by manager widgets, they do play a pivotal role in implementing it. A manager widget is typically initialized as a tab group; its primitive widget children are members of the tab group. The Text and List widgets are exceptions to this rule. These widgets are set up as their own tab groups, so that keyboard traversal can be used to move among the text in a Text widget or the items in a List widget. Within a tab group, there is no sense of a manager-within-manager structure. The widget hierarchy is flattened out so that it appears to the user that all of the controls in a window are at the same level.

Keyboard traversal only works if each widget in an interface cooperates. If a PushButton has the keyboard focus and the user presses the TAB key, the internals of the PushButton widget are responsible for directing the focus to the next tab group. Manager widgets play a key role in keyboard traversal because they are responsible for the keyboard events that take place within gadgets. If an event occurs within a PushButton gadget, its manager parent is responsible for directing the input focus to the next tab group.

Although the whole process of keyboard traversal may seem complex and difficult, it is automated by the Motif toolkit and does not require application intervention. However, the toolkit does provide mechanisms that allow you to control keyboard navigation. There are resources that allow you to specify widgets that are tab groups, widgets that are in tab groups, and widgets that do not participate in keyboard navigation. There are also functions that allow you to specify explicitly the direction of keyboard traversal. Fortunately, such fine-tuning is rarely necessary.

Putting Together a Complete Application

Managers and primitive widgets provide the basic tools with which you can build a graphical user interface from the ground up. Motif also provides several components that address the large-scale organization of an application. The specialized MainWindow manager widget is intended to be used as the organizing frame for an application. Motif also provides different types of menus and dialog boxes that can be used to organize application functionality.

Since an application is always used in conjuction with a window manager, we need to discuss the role played by the window manager. In the course of this discussion, we also need to take a closer look at shell widgets, since they provide the communication link between an application and the window manager.

Both pixmaps and colors play an important role in a graphical user interface. Motif provides routines that cache pixmaps so that they can be reused throughout an application. The three-dimensional appearance of Motif components is implemented using a variety of

color resources. It is important to understand these resources so that the 3D shadows are an effective part of the user interface.

The Main Window

Every application is different. A word processor, paint program, or spreadsheet typically has a single main work area, with controls taking on a peripheral role, perhaps in PulldownMenus. More sophisticated programs, on the other hand, may have several main work areas. For example, an electronic mail program may have a work area in which the user reviews and selects from a list of incoming messages, another where she reads and responds to messages, and yet another where she issues commands to organize, delete, or otherwise affect groups of messages. Still other applications, such as data-entry programs, don't really have a separate work area. The work area is really just a collection of controls, such as CheckBoxes and text entry areas, that are filled in by the user.

It is quite conceivable that an application could provide multiple windows for performing different tasks. For example, an order entry program might use one window for looking up a customer record, another for checking stock on hand, and yet another for entering the current order. Motif allows for the creation of multiple top-level application windows, as well as transient dialog boxes that ask for additional information or confirmation before carrying out a command.

Nonetheless, every application has at least one main window. The main window is the most visible window in an application. It is the first window the user sees and also the place where the user interacts with most application functionality. No matter how small or large an application may be, there needs to be a focal point that ties it all together. As a program grows more complex, the main window may grow more abstract and perform fewer functions, but it always exists. In a sophisticated application, the main window is transformed into a hub where the user starts, finishes, and returns again and again as she goes from one function to the next.

The *Motif Style Guide* suggests a particular layout for the main window. Applications should use this layout unless they have a compelling reason not to. The recommended layout is shown in Figure 3-19.

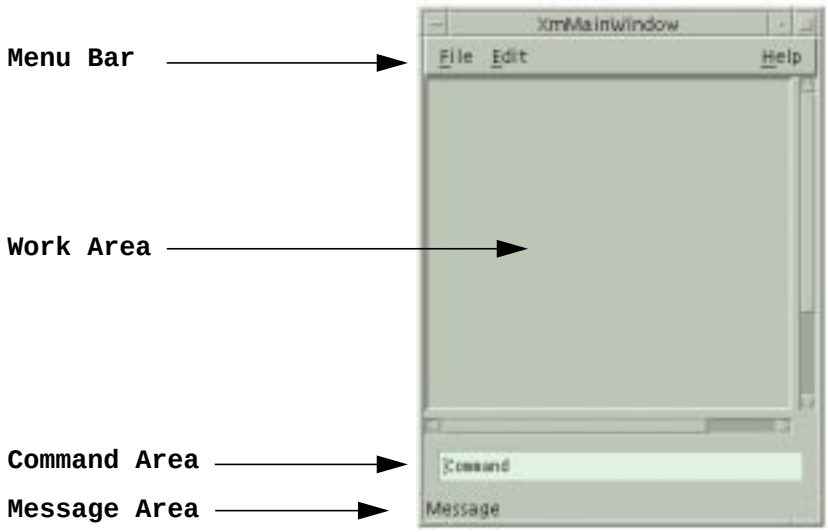


Figure 3-19 Recommended layout for MainWindow widget

A main window should have a menu bar across the top, with the work area immediately below it. The work area usually contains the main interface object of the application. For example, a paint or draw application might provide a DrawingArea widget as a canvas, an electronic mail application might provide a ScrolledList of message summaries from which the user can make selections, and a Text editor might place a Text widget in the work area. An application work area might require a custom widget or a non-widget-based X window instead.

The work area can have both horizontal and vertical scrollbars allowing the user to view its entire contents if they are too large to be displayed all at once. The main window can also contain an optional *command area* below the work area, where the user can enter typed commands. This area is most helpful for porting character-based applications to a Motif GUI, but it can be useful for other applications as well. At the bottom of the main window is an optional *message area*. This area should be used for status and informational messages only, not for error messages or any other type of message that requires a response from the user.

While it is possible to construct your own main window, the Motif toolkit provides the special-purpose MainWindow widget, which supports the recommended style. All of the elements in the MainWindow are optional, so an application can use it to display just the areas that it requires. The MainWindow widget is described in detail in Chapter 4.

Menus

Motif supports three different styles of menus. PulldownMenus that are displayed from the MenuBar in a MainWindow are the most common type of menu. A PulldownMenu is displayed when the user selects a CascadeButton in the MenuBar. The menu pane is displayed below the CascadeButton. Figure 3-20 shows a typical MenuBar and PulldownMenu.

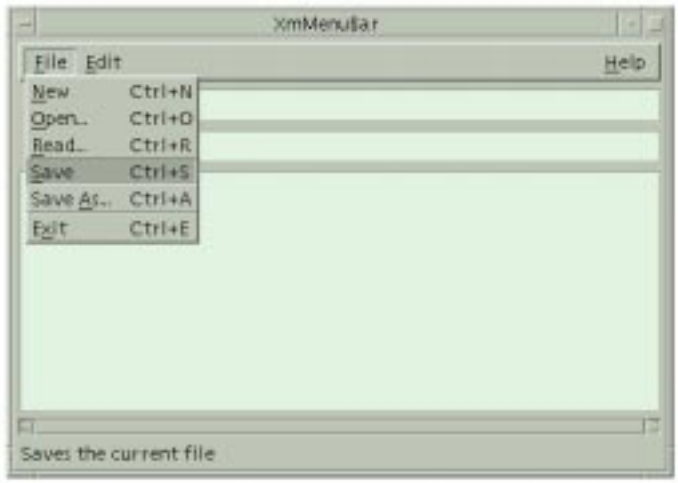


Figure 3-20 A MenuBar and an associated PulldownMenu

An item in a PulldownMenu can have a *cascading menu* associated with it. The cascading menu is displayed to the right of the menu item as shown in Figure 3-21, so these menus are sometimes referred to as *pullright menus*.

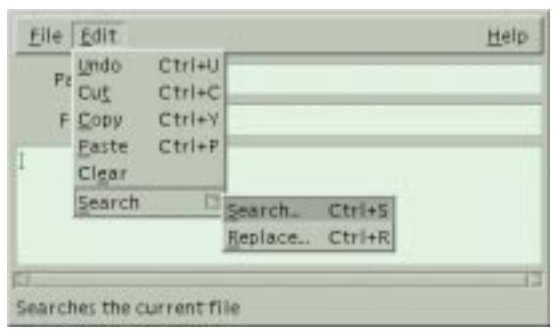


Figure 3-21 A cascading menu

MenuBars, PulldownMenus, and cascading menus are all created in a similar way. Motif provides convenience functions that create specially configured RowColumn widgets for these menu objects. The RowColumn widget is then populated with PushButtons,

CascadeButtons, ToggleButtons, and Separators, or their gadget equivalents. In the case of a MenuBar, all of the children must be CascadeButtons, since each button brings up a separate menu. In a PulldownMenu pane, most of the items are PushButtons or ToggleButtons, although Separators can be used for clarity. If an item posts a cascading menu, it must be a CascadeButton. The additional menu is created separately, populated with its own buttons, and attached to the CascadeButton.

Motif also supports a construct called an OptionMenu. An OptionMenu is another specially-configured RowColumn widget, but in this case the behavior is quite different. An OptionMenu is typically used to prompt the user to choose a value. The RowColumn widget displays a Label and a CascadeButton that shows the current value. When the user clicks on the button, a menu that contains the rest of the choices is popped up directly on top of the CascadeButton. Choosing an item from the menu modifies the label of the CascadeButton so that it shows the currently-selected item. Figure 3-22 shows an OptionMenu, both before and after it is popped up.

Additionally, Motif provides PopupMenus. Unlike the other types of menus, a PopupMenu is not attached to a visible interface element. A PopupMenu can be popped up at any arbitrary location in an application, usually as a result of the user pressing the third mouse button. PopupMenus are meant to provide shortcuts to application functionality, so an application can use different PopupMenus in different contexts and for different components in an interface.

Figure 3-22 An OptionMenu

A menu can be torn off from the component that posted it. A menu is normally only displayed for as long as it takes the user to make a selection. Once the selection is made, the menu is closed. When a menu is torn off, it remains posted in its own window. Now the user can make as many selections from the menu as she would like without having to repost the menu each time. For more information on tear-off menu functionality, as well as the different types of Motif menus, see Chapter 19, *Menus*.

The Window Manager

To the user, the MainWindow looks like the top-level window of an application. In window-system talk, a top-level window resides at the top of the window hierarchy for an application. Its parent is the *root window*, which is what the user perceives as the background behind all the windows on the desktop. In the Xt-world, however, things are a little different. Behind every visible top-level application window is a special kind of widget known as a shell widget.

Every window that can be placed independently on the screen, including top-level windows and dialog boxes, has a shell widget as its parent. The user does not see the shell because it is obscured by all of the other widgets in the window. A shell widget can only contain one managed child widget; the shell does not perform any geometry management except to shrink-wrap itself around this child. The child is typically a manager widget, such as a MainWindow, that is responsible for managing the layout of the primitive components, such as Labels, Text widgets, ScrollBars, and PushButtons. The items that the user actually sees and interacts with are descendants of the shell widget because they are contained within its boundaries.

Aside from managing its single child, the main job of the shell is to communicate with the *window manager* on behalf of the application. Without the shell, the application has no idea what else is happening on the desktop. It is very important for you to understand that the window manager is a separate application from your own. The visual and physical interaction between an application and the window manager is usually so close that most users cannot tell the difference between the two, but the distinction is important from a programming perspective.

To get an idea of the relationship between the window manager and an application, let’s compare it with the way a bed is built and how it fits into a room. A bed is made up of a frame, a mattress, and as many accessories as you want to pile on top of it. The main window is the mattress; the sheets, pillows, blankets, and stuffed animals you throw on it represent the user-interface controls inside the main window. The whole lot sits on top of the bed frame, which is the shell widget. When you push a bed around the room, you’re really pushing the bed’s frame. The rest just happens to go along with it. The same is true for windows on the screen. The user never moves an application window, she moves the shell widget using the window manager frame. The application just happens to move with it.

You may have to stretch your imagination a little to visualize a bed resizing itself with its frame, but this is precisely what happens when the user resizes an application. It is the window manager that the user interacts with during a resizing operation. The window manager only informs the application about the new size when the user is done resizing. The window manager tells the shell, the shell communicates the new size to its child, and the change filters down to the rest of the widgets in the application.

The window manager frame is composed of *window decorations* that the window manager places on all top-level windows. These controls allow the user to interactively move a window, resize it, cause it to redraw itself, or even to close it. Figure 3-23 shows the standard Motif window manager (*mwm*) decorations. For information on how to use *mwm*, see Motif Volume 3.

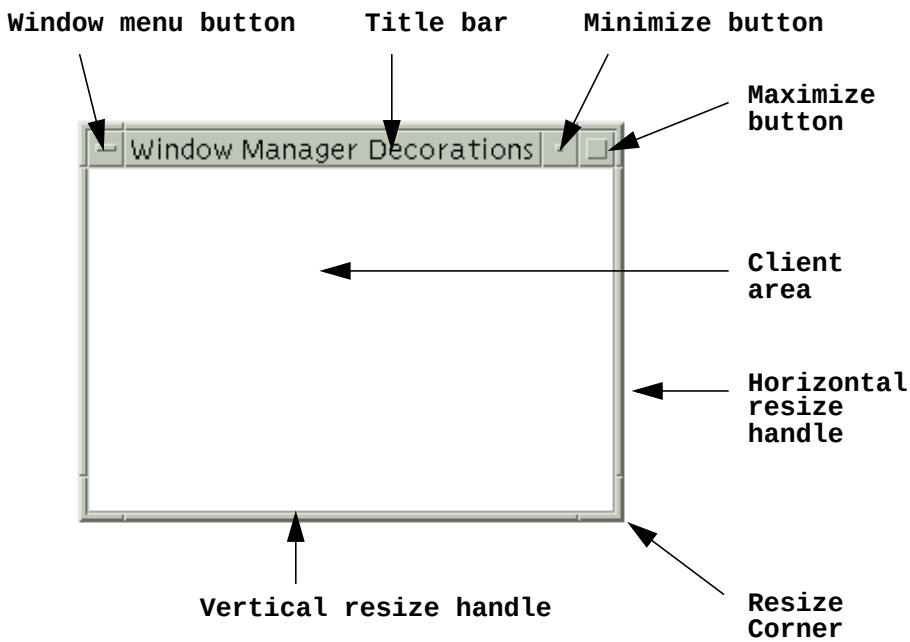


Figure 3-23 Motif window manager decorations

The *window menu* displays a list of window manager functions that allow the user to move, resize, and exit the application. An application does not have access to the menu itself or the items within it; similarly, it cannot get handles to the minimize and maximize buttons. These objects belong to the window manager and act independently from an application.

Motif provides *window manager protocols* that allow menu items like these to affect an application. An application can also interact with the window manager using many of the same types of protocols. You can specify which of the items in the window menu you want to appear, whether or not there are resize handles on the window frame, and whether or not you want to allow the user to iconify the window. However, the user is expecting all of the applications on her desktop to interact consistently with the window manager. This expectation is magnified by the fact that the user has probably set quite a few resources for the window manager. Since unexpected interference from an application rarely makes

users happy, you should leave the window manager alone. A technical discussion of the window manager can be found in Chapter 20, *Interacting With the Window Manager*.

As we pointed out earlier, it is possible for an application to have more than one independent window. In addition to the main window, there may be one or more dialog boxes, as well as popup windows, and even independent application windows that co-exist with the main window. Each of these cases requires different handling by the window manager, and as a result, there are several different classes of shell widgets. Figure 3-24 shows the class hierarchy of the different types of shell widgets available in the Motif toolkit. The Shell widget class is another metaclass that specifies resources and behaviors inherited by all of its subclasses.

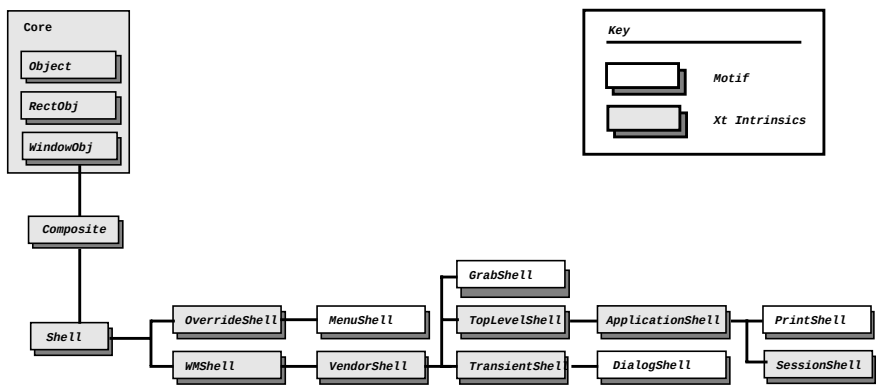


Figure 3-24 The Shell widget class hierarchy

Shells for Menus

In some cases, an application needs to put up a temporary window that is completely free of window manager interaction. Menus are one such a case. When a user pops up a menu, she typically wants to make a choice immediately, and she wants that choice to take precedence over any other window system activity. The window manager does not need to be involved either to decorate or to position the menu, as it is entirely up to the application.

As its name suggests, the OverrideShell widget class is provided for windows that bypass the window manager. OverrideShells are like futons; you can place them on the floor without using a bed-frame (and without being tasteless). It doesn't make much sense to use an OverrideShell as the main window for an application, except possibly for a screen-locking application. The purpose of this type of application is to prevent other applications from appearing on the screen while the computer is left unattended. Because the window manager is unaware of the OverrideShell, it does not provide window manager controls, and it does not interpret window manager accelerators and other methods for bypassing the lock.

The OverrideShell is a generic Xt-based widgetclass, so the Motif toolkit provides the MenuShell to service the special interface needs required by the *Motif Style Guide*. The MenuShell's translation table is set to support keyboard traversal, its `XmNfocusPolicy` is set to `XmPOINTER`, and its `XmNallowShellResize` resource is set to `True`. The MenuShell also makes sure that its child is a RowColumn widget. There is little more to be said about MenuShells, but for an in-depth discussion on the various types of menus you can use in Motif, see Chapter 19, *Menus*.

Shells for Window Manager Communication

Shell widgets must communicate with the window manager to negotiate screen real estate and a wide variety of other properties. The information that is exchanged is defined by the X Consortium's *Inter-Client Communications Conventions Manual* (ICCCM). The WMShell widget class implements ICCCM-compliant behavior as a standard part of the X Toolkit Intrinsics, so that it is available to all vendors providing Xt-based widget sets and window managers. This shell widget is what allows Motif applications to work correctly with virtually any ICCCM-compliant window manager. In our analogy, a WMShell is a simple, wire bed-frame that doesn't have any special attributes, like wheels or rollers.

The VendorShell widget class is subclassed from the WMShell class; it allows vendors, such as OSF, to define attributes that are specific to their own window managers. In our analogy, this widget class is like having a bed frame that has attached cabinets, shelves above the headboard, or nice wheels that glide on the carpet. The Motif VendorShell is aware of special features of *mwm*. The widget does not actually add any functionality to the window manager, but it is designed for applications that wish to interact with it. For example, all the attributes of window manager decorations can be modified or controlled through resources specific to the VendorShell.

WMShells and VendorShells are never instantiated directly by an application, but the features they provide are available to an application. For example, the Motif VendorShell allows an application to specify the items in the window menu and to control what happens when the user closes the window from the window menu. Chapter 20, *Interacting With the Window Manager*, discusses window manager interactions in more detail.

Shells for Dialogs

You can think of dialog boxes as an application's *secondary windows*. Since dialogs are not meant to remain on the screen for very long, they do not need all of the decorations that are typically provided by the window manager. However, dialogs are not completely independent like menus, so they do need to be controlled by the window manager. For example, if an application is iconified, its dialog boxes are typically iconified as well. Dialog boxes are usually implemented in Xt using TransientShells.

The DialogShell is a Motif-defined widget class subclassed from the TransientShell and VendorShell classes. Motif functions for creating dialog boxes tend to hide the shell widget

side of the dialog. When you make a call like `XmCreateMessageDialog()`, you are actually creating a MessageBox widget as a child of a DialogShell widget. See Chapter 5, *Introduction to Dialogs*, for details on Motif dialogs.

Shells for Application Windows

When you initialize the X Toolkit with a call such as `XtOpenApplication()`, you are automatically returned a SessionShell widget to use as the top-level widget in your application*. If an application uses additional top-level windows, they are typically TopLevelShells. The differences between these two classes are subtle and deal mostly with how resources are specified in a resource file. In Chapter 7, *Custom Dialogs*, we explore some ways in which TopLevelShells can be used as primary windows apart from the main window.

Dialogs

Some applications can get all their work done in one main window. Others may require multiple windows, so Motif allows an application to have multiple top-level windows. However, even applications without this level of complexity need to display transient windows called dialog boxes. Motif provides two main types of dialog boxes: message dialogs and selection dialogs. Message dialogs are designed to allow an application to communicate with the user, while selection dialogs prompt the user to enter different types of information. It is also possible to create custom dialogs for specialized application functionality.

Message Dialogs

Message dialogs simply communicate some kind of message to the user and include buttons that allow the user to respond to the message. For example, a menu item to delete a file might issue a dialog with the message, “Are you sure?” with PushButtons labelled *Yes*, *No*, and *Cancel*.

The Motif MessageBox widget that is used to create message dialogs actually comes in seven different guises. The different styles are meant to be used for different types of messages; some of the styles also display a symbol defined by the *Motif Style Guide*. Motif provides convenience routines for creating all of the different styles, so they are often referred to as if they are distinct widget classes.

ErrorDialog

The ErrorDialog shows a “do not enter” symbol along with a message that the user has made an error. For example, she may have pressed a PushButton at the wrong time,

* The ApplicationShell, `XtAppInitialize`() and `XtVaAppInitialize`() are considered deprecated in X11R6.

made an invalid selection in a List widget, or entered an unknown filename for a Text widget.

InformationDialog

The InformationDialog displays an "i" along with an informational message. These dialogs are usually displayed in response to a request for help.

MessageDialog

The MessageDialog does not display a symbol by default, although a symbol can be specified using the `XmNsymbolPixmap` resource. These dialogs can be used to display any kind of message.

QuestionDialog

The QuestionDialog shows a question mark symbol with a question that the user needs to answer. Questions are typically of the yes/no form, so the possible answers typically include *Yes* and *No*. A QuestionDialog should not be used for a question that requires an answer in the form of text or a selection from a list of some kind.

TemplateDialog

Motif provides a TemplateDialog to allow an application to create a custom dialog. By default, the TemplateDialog does not display a symbol or a message, but these items can be added to the dialog.

WarningDialog

The WarningDialog displays an exclamation mark along with a message that warns the user about a particular situation. These dialogs are commonly used to make sure that the user wants to do something destructive, like delete a file or exit an application without saving data.

WorkingDialog

The WorkingDialog displays an hourglass with a message indicating that the application is busy processing a lengthy computation or anything else that requires the user to wait.

Figure 3-25 shows a typical QuestionDialog in an application. For more information on message dialogs, see Chapter 5, *Introduction to Dialogs*.

Figure 3-25 A QuestionDialog

Selection Dialogs

Selection dialogs are meant to provide the user with a list of choices of some sort. Motif provides different styles of selection dialogs for different purposes. For example, a SelectionDialog presents a ScrolledList containing an arbitrary list of choices that can be selected with the mouse. The dialog also contains TextField widget that can be used to type in a choice which may or may not also be on the list. Figure 3-26 shows a SelectionDialog.

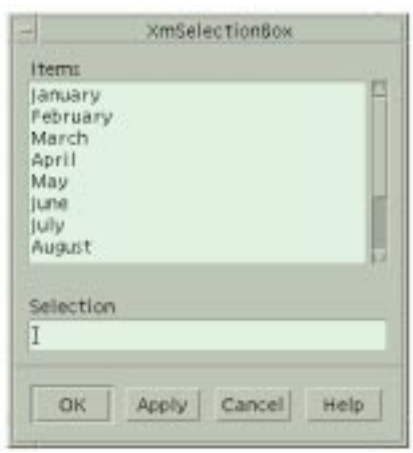


Figure 3-26 A SelectionDialog

The PromptDialog, as shown in Figure 3-27 is useful for prompting the user to enter some information.

Figure 3-27 A PromptDialog

The FileSelectionDialog is a more complex cousin to the SelectionDialog. It is used to select a file in the directory structure. A FileSelectionDialog is shown in Figure 3-28.

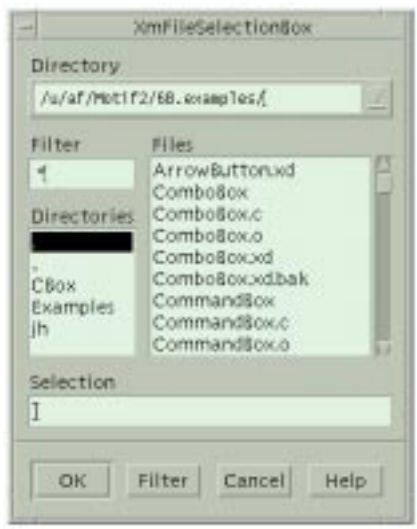


Figure 3-28 A FileSelectionDialog

The CommandDialog is an extension of the PromptDialog in that items input to the text entry field are stored in a ScrolledList. The intent is for the user to provide the application with commands; the list region contains a history of the commands that have already been

typed. The user can select an item in the history list to reissue a previous command. Figure 3-29 shows an example of a CommandDialog.

Figure 3-29 A CommandDialog

For detailed information about all of the different Motif selection dialogs, see Chapter 6, *Selection Dialogs*.

Custom Dialogs

There are many types of functionality that are not covered by the standard Motif dialog types. Fortunately, it is fairly easy to create your own dialogs. If you need to create a custom dialog, there are some guidelines in the *Motif Style Guide* that you should follow. At the highest level, all dialogs are broken down into two major components: the *control area* (or work area) and the *action area*. These areas are conceptual regions that may be represented by multiple widgets.

In a message dialog, the control area is used only to display messages, but as you can see from the selection dialogs, this area can be used to provide a variety of control elements. For example, the SelectionDialog uses a List widget and a TextField widget. It is also common for a custom dialog to display an array of PushButtons or ToggleButtons. A communications program might have a setup dialog that allows the user to set parameters such as baud rate, parity, start and stop bits, and so on, using an array of ToggleButtons. The controls in the control area provide information that is used by the application once an action area button is pressed.

Figure 3-30 shows a custom dialog with a control area that contains many items. Chapter 7, *Custom Dialogs*, discusses how to build customized dialogs, which may require the direct creation of widgets in the control area. Motif dialogs, on the other hand, do not require you

to create any of the objects in the control area. The widgets displayed in that part of the dialog are always predefined and automatically created.

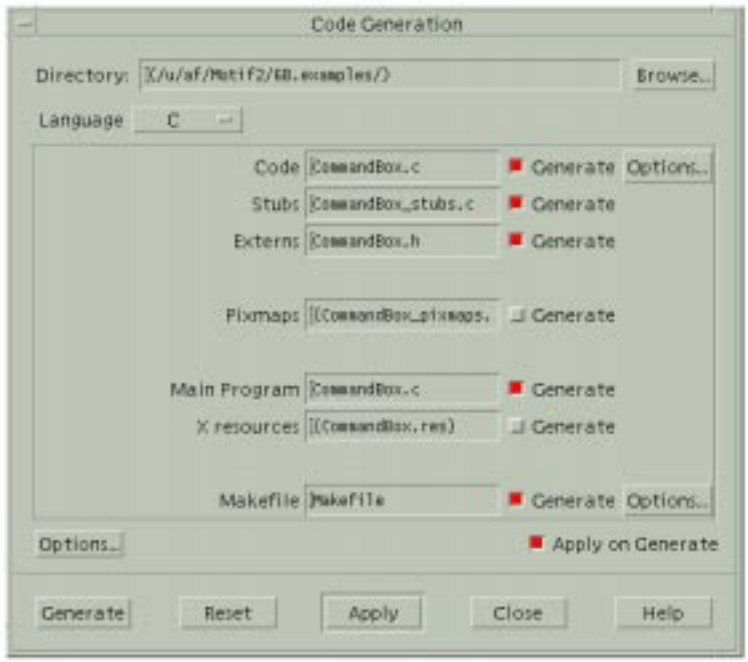


Figure 3-30 A custom dialog

Dialog Modality

One important concept to be aware of when it comes to dialogs is *modality*. In general, GUI-based programs are expected to be modeless. What this ultimately means is that the user, not the application, should be in control. The user should be able to choose from an array of application functions at any time, rather than stepping through them in a prearranged sequence, under the application's control.

Of course, there are limits to modelessness. Sometimes one thing has to happen before another. Often, sequencing can be taken care of simply by nesting graphical user interface elements. For example, faced with the main window, the user may have only a choice of menu titles; once she pulls down the file menu, she may have a choice of opening, closing, saving, renaming, or printing the contents of a file. At some point, though, she goes far enough down a particular path that her choices need to be constrained.

With respect to dialogs, modality allows a dialog box to acquire input before the user can go back to working with the application. For example, if the user asks to load a file, she may need to specify a filename in a dialog before she can edit the file. A modal dialog requires an answer immediately, by disallowing input to any other part of the application until it is either satisfied or cancelled. There may be other cases, though, where dialogs are

modeless. They can be left up on the screen without an immediate response, while the user interacts with the main application window or another dialog.

Pixmaps

In this section, we are going to take a closer look at how Motif supports graphic images. The Motif Label widget and all of its subclasses can display pixmaps as their labels. The MessageBox provides the `XmNsymbolPixmap` resource for specifying the image that is displayed in a dialog.

The Motif toolkit provides a number of routines for manipulating pixmaps. `XmGetPixmapByDepth()` and `XmGetPixmap()` both create a pixmap and cache it, so that it can be reused by an application. `XmGetPixmapByDepth()` provides a way to specify the depth of the pixmap that is created. `XmGetPixmap()` always creates a pixmap that has the same depth as the screen on which image is created. The caching mechanism provided by these routines is on a per-client basis; different processes cannot share pixmaps.

Whenever a new pixmap is created using one of these functions, the toolkit retains a handle to the pixmap in case another call is made requesting the same image. If this occurs, the function returns the exact same pixmap that was returned to the original requester and increments an internal reference counter. In order to keep a clean house, whenever you retrieve a pixmap using either `XmGetPixmap()` or `XmGetPixmapByDepth()`, you should call `XmDestroyPixmap()` when you no longer need the image. This function decrements the reference count for the pixmap. If the reference count reaches zero, `XmDestroyPixmap()` actually calls `XDestroyPixmap()` to discard the pixmap.

`XmGetPixmapByDepth()` takes the following form:

```
Pixmap XmGetPixmapByDepth( Screen    *screen
                           char      image_name,
                           Pixel     foreground,
                           Pixel     background,
                           int       depth)
```

The *image_name* can either be a filename or the name of an image registered using `XmInstallImage()`, which we are going to describe shortly. The background and foreground colors and the depth of the pixmap are specified by the corresponding parameters.

`XmGetPixmap()` takes the same form as `XmGetPixmapByDepth()`, minus the *depth* parameter. `XmGetPixmap()` creates a pixmap that has the same depth as the given *screen*, so you cannot rely on `XmGetPixmap()` to create a single-plane pixmap.* In

* The terms single-bit and single-plane are interchangeable; they imply a pixmap with only two colors: `0` and `1`. While the term *bitmap* usually refers to a single-plane pixmap, this is not necessarily true outside of the X social culture.

Motif, you can use XmGetPixmapByDepth() to create a bitmap; you can also use an Xlib routine, XCreateBitmapFromData().

Whenever XmGetPixmapByDepth() or XmGetPixmap() is called, it looks in the cache for a previously-created pixmap that matches the given name, colors, and depth. If the routine finds a match, it returns the cached pixmap and increments the reference count for the image. Since the pixmaps are cached, two separate parts of an application could have a handle to the same pixmap.

The *image_name* parameter is the key to where the routines get the data for the pixmap. As we just mentioned, this parameter can either be a filename or a symbolic name previously registered using XmInstallImage(). Both XmGetPixmap() and XmGetPixmapByDepth() use the following algorithm to determine what pixmap to return or create:

1. Look in the pixmap cache for an image that has the same *screen*, *image_name*, *foreground*, *background*, and *depth* as the specified image. If there is a match, return the pixmap.
2. If there is no match in the pixmap cache, look in the image cache for an image that matches the specified *image_name*. If there is a match, use the image to create the pixmap that is returned.
3. Otherwise, interpret the *image_name* as a filename, read the pixmap data directly out of that file, and create the pixmap.

The first step is fairly straightforward. The second step checks the image cache that is used internally by the Motif toolkit. Motif defines a number of images that you can use in an application. Table 3-1 lists the image names predefined by the toolkit.

Table 3-1 Predefined Image Names in the Motif Toolkit

Image Name	Description
background	Solid background tile
25_foreground	A 25% foreground, 75% background tile
50_foreground	A 50% foreground, 50% background tile
75_foreground	A 75% foreground, 25% background tile
vertical_tile	Vertical lines tile (Motif 1.2.3 onwards)
horizontal_tile	Horizontal lines tile (Motif 1.2.3 onwards)
horizontal	As horizontal_tile (Motif 1.2.2 backwards compatibility)
vertical	As horizontal_tile (Motif 1.2.2 backwards compatibility)
slant_left	Left slanting lines tile
slant_right	Right slanting lines tile
menu_cascade	A rightwards pointing arrow (Motif 2.1)
menu_cascade_rtol	A leftwards pointing arrow (Motif 2.1)

Table 3-1 Predefined Image Names in the Motif Toolkit (continued)

Image Name	Description
menu_checkmark	A tick mark (Motif 2.1)
menu_dash	A horizontal line (Motif 2.1)
collapsed	A rightwards pointing filled arrow (Motif 2.1)
collapsed_rtol	A leftwards pointing filled arrow (Motif 2.1)
expanded	A filled arrow pointing downwards (Motif 2.1)

Motif also installs a number of images at run-time to support dialog images and other random pixmaps. None of these image names are publicly available. You can install your own images by predefining them and loading them into the image cache using `XmInstallImage()`, which takes the following form:

```
Boolean XmInstallImage (XImage image, char *image_name)
```

The *image* parameter is a pointer to an `XImage` data structure that has been previously created or, more commonly, statically initialized by the application. It is possible to create an image dynamically from an existing window or pixmap using `XGetImage()`, but this is not the way the function is typically used.

If you attempt to install an image using an *image_name* that matches one already in the cache, the function returns `False` and the image is not installed. Otherwise, the function returns `True`. You can uninstall an image by calling `XmUninstallImage()`. Once the image is uninstalled, it cannot be referenced by name any more and a new image may be installed with the same name. The `XImage` structure is not copied by `XmInstallImage()`, so if the image pointer you pass has been allocated using `XCreateImage()` or `XGetImage()`, you must not free the data until after you call `XmUninstallImage()`.

If `XmGetPixmap()` or `XmGetPixmapByDepth()` finds a match in the image cache, it creates the pixmap based on the image data, not on the image itself. As a result, the pixmap that is created is not affected by the image being uninstalled by `XmUninstallImage()`.

If the pixmap retrieval routines do not find a match in the image cache, the pixmap is loaded from a file. If *image_name* starts with a slash character (/), it is taken as a full pathname. Otherwise, the routines look for the file using a search path. On POSIX systems, the environment variable `XBMLANGPATH` can be set to specify a desired directory in which to search for bitmap files. If this variable is not set, the pathname used is based on the values of the `XAPPLRESDIR`, `HOME`, and `LANG` environment variables. See the reference page in Volume 6B, for complete details on the search path that is used.

When `XmGetPixmap()` or `XmGetPixmapByDepth()` looks in the pixmap cache for a image name, the pathname must match completely for the routine to return a cached image. The file *xlogo64* will not match a previously-loaded pixmap that has the name */usr/X11R6/include/bitmaps/xlogo64*. If you do not need to worry about using different pixmaps for

different environments, we recommended that you always specify a full pathname to these routines to be assured that you get the desired file.

Color

Color plays an important role in a graphical user interface. It appeals to the senses, so it can provide an aesthetic quality, while at the same time it can be used to convey information to the user. However, for all the power of color, it is frequently abused by applications. A color combination that appeals to some people may offend others. The safest bet with color is to avoid hard-coding any use of color in your application and provide enough flexibility so that the user can configure colors in a resource file or interactively using the application. Of course, many applications are based on the use of color, so this sweeping generalization only applies to those parts of an application that are not dependent on color. In any case, you should be wary when providing information or state purely through the use of color: a color-blind user may not notice the differences; color-blindness is not a trivial or rare issue.

The Motif widget set provides a number of widget resources that specify colors. All of the Motif widgets use the `XmNforeground` and `XmNbackground` resources. Although every widget class makes different use of the `XmNbackground` and `XmNforeground` resources, text is typically rendered in the foreground color and everything else is shown using the background color. Some widgets provide additional color resources for particular aspects of their appearance. For example, ToggleButtons use the `XmNselectColor` resource for the square/diamond selection indicator, PushButtons use `XmNarmColor` as their background when they are armed, and ScrollBars use `XmNtroughColor` to set the color of the area behind the slider and directional arrows. In Motif 2.1, gadgets can also be colored in much the same way that their widget equivalents can; in Motif 1.2, however, their colors are inherited from their Manager parent.

The `XmNborderColor` resource is another resource that can be specified for any widget, as it is defined by the Core widget class. Since Motif widgets typically have a border width of `0`, this resource is rarely used. The `XmNhighlightColor` resource specifies the color of the highlighting rectangle that is displayed around the interface component that has the keyboard focus. This resource is defined by the Gadget, Manager, and Primitive metaclasses, so it can be specified for any Motif component.

Perhaps the most troublesome of all the color resources are `XmNtopShadowColor` and `XmNbottomShadowColor`. These are the colors that give Motif widgets their 3D appearance on a color display. If set inappropriately, these colors can ruin the aesthetics of an interface. These resources are set automatically by the toolkit based on the background color of the object, so the colors are not normally a problem. If the background color of a PushButton is blue when it is created, the toolkit automatically calculates the `XmNtopShadowColor` to be a slightly lighter shade of blue and the `XmNbottomShadowColor` to be a slightly darker shade.

The problems arise if you want to change the background color of a widget dynamically because the toolkit does not automatically change the shadow colors for you. So if you change the XmNbackground of the PushButton to red, the top and bottom shadow colors remain the different shades of blue. In Motif 1.2, note that the shadow resources are only used by widgets, not gadgets: if you dynamically change the background color of a manager widget, it automatically recalculates the top and bottom shadow colors and redisplays its gadgets correctly. Many consider the fact that this process is not automated for widgets to be a design flaw in the Motif toolkit.

If you need to change the background color of a widget dynamically, you can recalculate the shadow colors and set the resources yourself. You can use the XmChangeColor() routine, which takes the following form:

```
void XmChangeColor (Widget widget, Pixel background)
```

This routine changes all the foreground color, shadow colors, and select color for the specified *widget* based on the *background* color. The select color only applies to ToggleButtons (XmNselectColor) and PushButtons (XmNarmColor).

The routine XmGetColors() can be used to query the colors which Motif calculates. XmGetColors() takes the following form:

```
void XmGetColors( Screen      *screen,
                  Colormap    colormap,
                  Pixel       bg,
                  Pixel       *fg,
                  Pixel       *top_shadow,
                  Pixel       *bottom_shadow,
                  Pixel       *select)
```

This routine takes a colormap and a background color and calculates and returns an appropriate foreground color, top and bottom shadow colors, and select color. Once you have the colors, you could specify the appropriate resources for the widget.

A basic problem behind setting and getting colors for widgets is that what you get for a given pixel value depends on the colormap. A pixel is simply an index value into an array of color definitions (a colormap). The problem with colormaps is that you never know what colormap is associated with any particular widget.

By calling XtVaSetValues() using the type-converting resource, XtVaTypedArg, we defer the problem to the toolkit and its string-to-color type converter. The toolkit allocates the color out of the colormap already owned by the toolkit and sets the background color accordingly. Then we can get the actual pixel value and the colormap using XtVaGetValues(). We pass the colormap and the background pixel value to XmGetColors() to calculate the rest of the colors. Once we have obtained all of the colors, we can set them using XtVaSetValues().

The Label widget and its subclasses cannot display text using more than one color. However, you can create a multi-plane pixmap and render various strings directly into it using `XDrawString()`. You can use multiple colors by changing the foreground color in the `GC` using `XSetForeground()` or `XChangeGC()`. Once you have the pixmap, you can use it to set the `XmNlabelPixmap` resource for the widget.

The text of the entries in a List widget is rendered using the widget's `XmNforeground` color. You cannot change the color of individual items in a List widget. The `XmNbackground` of the List affects all areas of the widget not associated with the entries themselves. The text in a Text widget or a TextField widget is also displayed using the `XmNforeground` color; there is no way to display text using different colors in these widgets. When a List widget or Text widget is the direct child of a ScrolledWindow, the ScrollBars automatically match the background color of the List or Text widget.

Changes in Motif 2.1

Release 2.1 of the Motif toolkit introduces a number of new features, as well as many enhancements to existing functionality. This section summarizes all of the changes in Motif 2.1 and refers you to other sections in the book for more detailed information on specific changes. We also describe the changes that we made to the example programs in the book to make them accurate with respect to Motif 2.1.

General Toolkit Changes

Gadget Resources

Gadgets can now be painted independently, and no longer directly inherit their color appearance from the Manager parent. Foreground, background, top and bottom shadow, and highlight colors are now included in the gadget cache. Similarly cached are the top and bottom shadow pixmaps, and the highlight pixmap.

Traits

A Trait is an encapsulation of a piece of logical widget behavior. It defines a set of methods for querying and setting this behavior, whatever it may be. Different widget classes may share in common the behavior, even though their class inheritance graphs are only vaguely related. To be more concrete, if we consider a ComboBox and a Text widget, the class hierarchy for the ComboBox does not derive through a Text class directly, and yet considered logically, because the ComboBox and the Text widget both have a value which is a string, there is sufficient in common such that we could define methods to read or write the value irrespective of which widget instance we are actually dealing with. Such methods already exist in Motif 2.0, and are known as a Trait.

Traits are named, and there is a standard routine for querying a widget to determine whether it supports a given trait. And thus there are two ways of setting the value of a text widget: we can use the older `XmTextSetString()` functional interface, or we can fetch the `XmQTaccessTextual` trait from the widget concerned, then use the `setValue()` routine of the trait. The beauty of the second method is that it will also work for other widgets in the Motif set which are logically also Text-like in some of their behavior.

However, Traits are really the domain of the widget author, to provide consistency in behavior between logically related widget classes. Mention of particular Traits will be made if and when necessary, otherwise you are referred to the Widget Writer's Guide in the official documentation.

Renditions and RenderTables

The `XmFontList` data type and associated functions are now considered deprecated. In Motif 1.2, the appearance of compound strings depended upon a small number of widget attributes, of which the `XmNfontList` resource is the most important. The mechanisms for inheriting compound string appearance characteristics relied solely upon default `XmFontList` values derived usually from the containing VendorShell or BulletinBoard. In Motif 2.0, there is the new entity called the `XmRendition`, which is a named (tagged) object that consists of a complete set of appearance resources, including coloration, font, underline and strike-through settings. An `XmRendition` is a shareable object which is independently reference counted. An `XmRenderTable` is simply a set of `XmRendition` objects; compound strings are rendered by comparing tags associated with components in the string against tagged `XmRendition` objects in the `XmRenderTable`. The means whereby a widget inherits compound string rendering information is now rationalized through the Trait mechanisms: a parent widget may choose to implement a Trait which provides default render table data to its descendants. The BulletinBoard, VendorShell, and MenuShell classes implement such a Trait.

The appearance of a compound string can now be specified through a whole group of attributes that can be manipulated as a single set. Compound strings may now be multi-colored as a result.

An `XmRendition` object is a pseudo-widget: although not true widget classes, Renditions and RenderTables may be specified in resource files, as well as in code. Widget classes which used to support the `XmNfontList` now also support an `XmNrenderTable` resource. For backwards compatibility, the `XmNfontList` resource continues to persist, although it is internally implemented through the new `XmRenderTable` type.

It is not necessary to precisely specify all attributes for each and every Rendition within a RenderTable: attributes may be given the value `XmAS_IS`, which simply means that the value of the attribute is inherited from Renditions which are placed earlier in the RenderTable.

Renditions and RenderTables are discussed at length in Chapter 24.

TabLists

In Motif 1.2, creating tabular or multi-columnar data within a widget could usually only be performed through some code by the programmer which required careful calculations based upon the size of the current font. Motif 2.0 introduces the notion of an XmTabList, which is a set of XmTab objects. An XmTab describes a logical offset across a widget: it consists of a floating point quantity, a unit in which the quantity is expressed (inches, font units, millimetres, and so forth), and an offset model, which specifies whether the Tab value is counted in terms of absolute distance across the widget, or relative to a previous XmTab object in the XmTabList.

The new XmRendition object contains an XmTabList attribute. The creation of a multi-column list can now be achieved by embedding tab component separators within the compound strings of the list: each tab separator marks the beginning of a new column entry, where that column appears on the screen depends on the XmNtabList attribute of the Rendition used to render that portion of the compound string. Tabs and TabLists are covered as part of the discussion in Chapter 25, *Compound Strings*.

Compound Strings

Compound strings have been re-modelled to use the new XmRendition object. In order to do this, new XmString component types have been defined.

The compound string segments XmSTRING_COMPONENT_RENDITION_BEGIN and XmSTRING_COMPONENT_RENDITION_END can be embedded into a compound string in order to associate portions of the string with particular Rendition specifications.

To enable tabular layout of compound strings, the new XmSTRING_COMPONENT_TAB segment is defined, and this marks a column boundary within the string. How this is rendered will depend upon the value of the XmTabList attribute associated with the current Rendition in force.

Additionally, the compound string segments XmSTRING_COMPONENT_LAYOUT_PUSH and XmSTRING_COMPONENT_LAYOUT_POP can be used to embed layout direction specifications into the string.

XmStringComponentCreate() has been augmented to create the new component types.

Compound Strings are discussed in Chapter 25.

Parse Mappings and Parse Tables

Strings and compound strings can be dynamically manipulated by new table-driven parsing routines. An XmParseMapping represents an entry in the table, an XmParseTable is the table itself. Each entry in the table specifies a transformation: what to compare against in

the original input string, what to replace any matching occurrence with, and so forth. The XmParseMapping object can either perform simple substitutions by supplying fixed substitution patterns, or it can specify further substitution routines which dynamically modify the input depending on circumstances.

Typically, parse tables and their constituent parse mapping objects are used by passing them as parameters to the XmStringParseText(), XmStringUnparse(), and XmStringGenerate() functions.

Essentially, parse tables are simply filters which provide programmatic control over the way in which strings are converted into compound strings, or vice versa.

Parse Mappings are discussed in Chapter 25, *Compound Strings*.

Layout Direction

In Motif 1.2, although compound strings could be reversed by suitable setting of the XmNstringDirection resource, the layout of components in which they were rendered could not. The new Motif 2.0 XmNlayoutDirection resource rectifies the issue: it is possible to reverse the layout of a ComboBox, for example, so that the constituent arrow button is drawn to the left of the text. This could be performed at user request either for reasons of Internationalization or handedness. Layout direction resources are added to both the Manager and Primitive base classes: all Motif widgets therefore inherit the control.

Uniform Transfer Model

In Motif 1.2, different styles of communication between widgets required separate code to implement. Thus the codes to implement data transfer through the ClipBoard, to the primary or secondary selection, and through Drag and Drop would not necessarily share much in common in terms of the functions required to achieve the desired effect. In Motif 2.0, the disparate communication interfaces have been subsumed into a common Uniform Transfer Model.

Under the Model, two new callbacks are added to the system: an XmNconvertCallback, and an XmNdestinationCallback. The convert callback is associated with the source of the data, and is both responsible for exporting the data in the format required by the destination, and in furnishing a list of formats in which the source is prepared to export that data. The destination callback communicates with the source in order to determine the best format in which to receive the data, and it arranges for the data to be handled appropriately when it arrives by setting up a transfer procedure to perform the task. The simplest destination callback could in fact request data in a fixed format from the source without bothering to request the list of supported forms.

The programmer is not required to implement convert and destination callbacks for all the various types of data transfer which Motif supports. Widgets have mechanisms which utilize the Trait system in order to effect default data transference. A programmer only

needs to write convert or destination callbacks where the data is to be transferred in a manner which differs from the built-in target formats.

The Uniform Transfer Model is discussed in Chapter 23, *The Uniform Transfer Model*.

Automatic Popup Support

In the past, in order to popup a context sensitive menu, it was necessary to write event handler code to intercept ButtonPress events, followed by the appropriate code to pick and display the relevant menu. In Motif 2.0, the RowColumn widget has been enhanced to provide auto-popup behavior, and the decision making process of selecting the relevant menu to display has been encapsulated in a new callback, the `XmNpopupHandlerCallback`, built into the Manager and Primitive classes. Now it is only necessary to provide the callback, filling in an appropriate field of the callback data, in order to specify the required menu: the housekeeping tasks of event interception and menu display are built-in.

Specific Widget Changes

Motif 2.1 introduces a number of new widget classes, as well as including new resources for classes previously defined.

VendorShell

The VendorShell has the new resources `XmNbuttonRenderTable`, `XmNlabelRenderTable`, and `XmNtextRenderTable`. These supersede the deprecated `XmNbuttonFontList`, `XmNlabelFontList`, `XmNtextFontList` resources respectively.

For finer control over the X input contexts which are created in Internationalized applications, the resource `XmNinputMethod` is provided: the value `XmPER_SHELL` creates one input context per shell hierarchy, the value `XmPER_WIDGET` creates one for each widget which requests one.

VendorShell also supports the `XmNlayoutDirection` resource. The widget does not use this resource itself, but maintains and supplies the resource as a default for whichever descendant in the widget hierarchy lacks an explicit value.

In Motif 2.0, `XmNshellUnitType` is considered deprecated: it is replaced by the `XmNunitType` resource. This also acts as a default value for widget descendants requiring resolution information.

ArrowButton

The XmNdetailShadowThickness resource allows the programmer to specify the shadow thickness inside the triangle of the ArrowButton. The ArrowButtonGadget also supports the resource.

BulletinBoard

The BulletinBoard has the new resources XmNbuttonRenderTable, XmNlabelRenderTable, and XmNtextRenderTable which superseded the deprecated XmNbuttonFontList, XmNlabelFontList, XmNtextFontList resources respectively.

ComboBox

ComboBox is a new widget as of Motif 2.0, combining direct textual input with the convenience of list selection.

Container

Container is a new widget in Motif 2.0. It organises IconGadget children in a variety of layout styles, including a Tree format.

Display

The XmDisplay object has suffered a number of changes in order to interface Motif with a CDE desktop. Most of the resources alter the appearance of Toggles, and the shadowing on Buttons, and are described fully in Volume 6B.

The most important of the new resources are the XmNnoFontCallback and XmNnoRenditionCallback lists. Whenever an attempt is made to render a compound string, if font or rendition information is found to be absent, a callback can be supplied by the programmer which can attempt to find an alternative. This is a significant improvement over Motif 1.2, where the system itself would decide on an appropriate default font without recourse to any intelligent intervention.

DrawingArea

DrawingArea now supports the new XmNconvertCallback and XmNdestinationCallback resources associated with the Uniform Transfer Model. The DrawingArea itself does not define any export target formats.

FileSelectionBox

In Motif 2.0 and later, the search pattern and base directory path can be displayed in separate text fields, as opposed to being concatenated together and displayed in a single field. The resource XmNpathMode controls whether this new feature is enabled.

Gadget

The appearance resources XmNbackground, XmNbackgroundPixmap, XmNbottomShadowColor, XmNbottomShadowPixmap, XmNhighlightPixmap, XmNtopShadowPixmap are added so that Gadgets no longer strictly inherit their colors from the Manager parent.

As for the Manager and Primitive base classes, Gadget also supports XmNlayoutDirection to control the order in which components of the object are laid out.

GrabShell

A new widget in Motif 2.0. GrabShell is a shell widget which grabs the pointer and keyboard when it is mapped. It therefore directs focus to its child, and is used by the ComboBox to implement its popup list.

IconGadget

New in Motif 2.0, the IconGadget can display both textual and pixmap information simultaneously. The gadget is closely associated with the Container. Each IconGadget supposedly represents pictorially some application object of some kind, and the Container organises the layout and selection of the given objects. Extra "detail" data can be associated with an IconGadget, and the Container can display this extra information in a tabular format.

Label

The XmNfontList resource is deprecated, and is superseded by the XmNrenderTable resource. Similarly for LabelGadget.

List

The List supports keyboard matching of items in Motif 2.0 and later. If the resource XmNmatchBehavior is enabled, characters typed are compared with the first character of each item, and the new currently selected item is reset accordingly. The color of the selected item itself can now be specified through the XmNselectColor resource.

The set of selected positions can be manipulated through the new XmNselectedPositions, XmNselectedPositionCount resources.

The way in which the user selects items in the list is controllable through the XmNselectionMode resource. In XmNORMAL_MODE, navigating the list using the keyboard can select the item under the location cursor. In XmADD_MODE, navigating through the list has no side effects with respect to the selected item set.

The List supports the XmNdestinationCallback in order to make the widget partake in the Uniform Transfer Model.

MainWindow

From Motif 2.0, the routine XmMainWindowSetAreas() is marked as deprecated. The programmer should set the XmNcommandWindow, XmNmenuBar, XmNmessageWindow, XmNworkWindow, XmNhorizontalScrollBar, XmNverticalScrollBar resources directly using the standard Xt mechanisms.

Manager

New support for automatic popup menu control is provided through the Motif 2.0 XmNpopupMenuHandlerCallback.

The Motif 2.0 XmNlayoutDirection resource facilitates automatic layout control.

MenuShell

The XmNbuttonFontList and XmNlabelFontList resources are deprecated, and are superseded by the XmNbuttonRenderTable and XmNlabelRenderTable resources. Similarly deprecated is the XmNdefaultFontList resource, although there is no replacement XmNdefaultRenderTable resource.

Notebook

Notebook is a new widget in Motif 2.0. It simply lays out its children as though they are pages in a book.

PanedWindow

As of Motif 2.0, the PanedWindow now officially supports a horizontal configuration: set the XmNorientation resource to XmHORIZONTAL or XmVERTICAL to taste.

Primitive

The XmNlayoutDirection, XmNconvertCallback resources are added to this base class.

To support automatic context-sensitive menus, the XmNpopupHandlerCallback has been added to the system.

PrintShell

The PrintShell interfaces with the X11R6 X Print (Xp) extensions. A widget hierarchy can be printed by creating that hierarchy underneath a PrintShell, followed by appropriate code to invoke the printing. Printing can be either synchronous, or asynchronous, and the programmer can decide, by setting appropriate widget resources, whether the output is to

consist of the contents of the widgets concerned, or whether it is more of a screen snapshot of the widgets themselves.

RowColumn

A new resource, `XmNtearOffTitle`, allows the programmer to specify a title for a tear-off menu.

Scale

As of Motif 2.0, the Scale widget supports automatic tick marks. The function `XmScaleSetTicks`() evenly spaces marks of various sizes along the edge.

The Scale can be configured as to whether it responds to user input through the new `XmNeditable` resource: for a read-only scale, set the resource to false.

Arrows can be placed at either or both ends of the Scale through the `XmNshowArrows` resource, and the general appearance of the slider is configurable through `XmNsliderMark`: this can be configured to appear in various etched rectangle arrangements, as a circle, or as a thumb mark.

The size of the slider is configurable through the `XmNsliderSize` resource. This resource is undocumented by the official channels, and thus there is no official guidance to its usage.

The color of the slider is also tunable: it can either be based upon the foreground or background of the Scale, or upon the existing trough color. The `XmNsliderVisual` resource controls this aspect of behavior.

The Scale can behave as a thermometer, with the slider anchored at one end rather than floating in the middle. `XmNslidingMode` is the resource required to configure this setting.

Lastly, as of Motif 2.0, the `XmNfontList` resource is deprecated, and replaced with the newer `XmNrenderTable` resource. The Scale also supports the `XmNconvertCallback` list in order to participate in the Uniform Transfer Model.

Screen

The XmScreen object has been enhanced to provide a greater control over the way in which Motif allocates colors. The `XmNcolorAllocationProc` resource allows the programmer to specify a procedure to perform the allocation. The default is the standard `XAllocColor`() routine.

Similarly, the algorithm by which Motif calculates default foreground, background, and shadow colors is also now tunable through the `XmNcolorCalculationProc` resource.

The allocation of pixmaps can be controlled through the `XmNbitmapConversionModel` resource: by default (`XmMATCH_DEPTH`) pixmaps are created such that the depth matches

the widget for which they are allocated. Setting the value to XmMATCH_DYNAMIC converts loaded bitmap files to a pixmap depth of 1.

Also on the subject of pixmaps, the XmNinsensitiveStipplePixmap resource provides a stipple to use when making widgets appear insensitive. This is mostly used internally by the Gadget utilities.

Motif as of version 2.0 supports the notion of color objects: the XmNuseColorObject resource enables the feature, such that if a color is dynamically altered, all widgets which reference the color are changed as a side effect. Clearly, this resource is part of the CDE enhancements to Motif: it allows the desktop to change the whole style of color of an application without having to modify the entire widget hierarchy.

ScrollBar

Much of the enhancements associated with the Scale are in fact related to the ScrollBar: XmNeditable, XmNshowArrows, XmNsliderMark, XmNsliderVisual, XmNslidingMode are all newly supported as of Motif 2.0.

The resource XmNsnapBackMultiple controls the behavior of the ScrollBar if the user drags the mouse outside the bounds of the widget. It specifies a distance, which if exceeded, causes the ScrollBar to snap back to its pre-drag settings.

ScrolledWindow

As of Motif 2.0, the ScrolledWIndow (and derived classes) supports automatic drag through the resource XmNautoDragModel.

SpinBox and SimpleSpinBox

Two new widget classes, the first available as of Motif 2.0, the second from Motif 2.1, which allows the user to rotate through a range of values. SpinBox is the general purpose manager, into which any number of Text components are added. It rotates the values associated with the Text component which currently has the focus. SimpleSpinBox is a pre-packaged unit that contains a single built-in Text component. The range of values associated with any Text is specified through constraint resources. Rotation of the values is achieved by pressing on an ArrowButton which the SpinBox components automatically add for the purpose.

Text and TextField

The number of lines within the Text is now available through the XmNtotalLines resource, added as of Motif 2.1.

In both widget classes, the XmNfontList resource is obsolete, replaced with the XmNrenderTable resource, and the XmNdestinationCallback is added in order to interface with the Uniform Transfer Model.

ToggleButton and ToggleButtonGadget

The Toggle widgets have been reworked in order to provide consistency of appearance under the CDE environment.

The resource XmNdetailShadowThickness controls the thickness of the shadow on the Toggle indicator.

In Motif 2.0 and later, a Toggle may be in one of three states: set, unset, and indeterminate. By default, the Toggle holds two states, unless the resource XmNtoggleMode is set to XmTOGGLE_INDETERMINATE, which enables the third state. The resource XmNindeterminateInsensitivePixmap and XmNindeterminatePixmap are pixmaps displayed when the toggle is in the third indeterminate state.

In Motif 1.2, the resource XmNindicatorOn is a Boolean value; in Motif 2.0 and later, this becomes an enumerated type, and specifies not just whether the indicator is visible, but also its appearance: a check box, shadowed box, check (tick) mark, cross, and so on become available. This blurs the distinction with the resource XmNindicatorType, which is extended to include XmONE_OF_MANY_ROUND, XmONE_OF_MANY_DIAMOND, indicating a round or diamond shaped indicator.

The resource XmNset also changes type from Boolean to an enumeration. The valid range is now XmUNSET, XmSET, and XmINDETERMINATE.

Lastly, an XmNunselectColor is added from Motif 2.0 onwards to complement the XmNselectColor resource.

Changes to the Example Programs

All of the example programs in this book have been updated to Motif 2.1 and X11R6. For example, calls to manipulate compound strings and font lists have been replaced with calls to handle the new render table type.

Changes involving new Motif 2.1 functions and resources are described in detail when each example is presented.

Summary

The Motif widget set gives you a great deal of flexibility in designing an application. But with this flexibility can come indecision, or even confusion, about the most effective way to use these objects. If you want to give a user a set of exclusive choices, should you use a PulldownMenu, a dialog box that contains ToggleButtons arranged in a CheckBox, or a List widget? There is no right answer--or perhaps it is better to say that the right answer depends on the nature of the choices and the flow of control in your application.

Designing an effective user-interface is an art. Only experience and experimentation can teach you the most effective way to organize an application. What we can do in this book is teach you how to use each widget class and give you a sense of the tradeoffs involved in using different widgets. In this chapter, we've given you a broad overview of the Motif toolkit. Subsequent chapters delve into each widget class in detail. You should be able to read the chapters in any order, as the needs of your application dictate.

In this chapter:

- *Creating a MainWindow*
- *The MenuBar*
- *The Command and Message Areas*
- *Using Resources*
- *Summary*
- *Exercises*

4

The Main Window

This chapter describes the Motif MainWindow widget, which can be used to frame many types of applications. The MainWindow is a manager widget that provides a menu bar, a scrollable work area, and various other optional display and control areas.

As discussed in Chapter 3, *Overview of the Motif Toolkit*, the main window of an application is the most visible and the most used of all the windows in an application. It is the focal point of the user's interactions with the program, and it is typically the place where the application provides most of its visual feedback.To encourage consistency across the desktop, the *Motif Style Guide* suggests a generic main window layout, which can vary from application to application, but is generally followed by most Motif applications. Such a layout is shown in Figure 4-1. As described in Chapter 3, a main window can provide a menubar, a work area, horizontal and vertical scrollbars, a command area, and a message area.

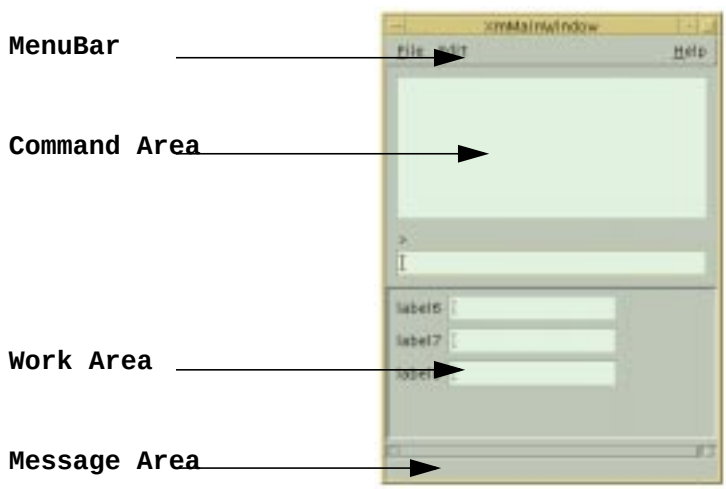


Figure 4-1 The main window of a Motif program

In an effort to facilitate the task of building a main window, the Motif toolkit provides the MainWindow widget. This widget supports the different areas of the generic main window

layout. However, the MainWindow widget is not the only way to handle the layout of the main window of your application. You are not required to use the MainWindow widget and you should not feel that you need to follow the Motif specifications to the letter. While the *Style Guide* strongly recommends using the main window layout, many applications simply do not fit the standard GUI design model. For example, a clock application, a terminal emulator, a calculator, and a host of other desktop applications do not follow the Motif specifications in this regard, but they can still have Motif elements within them and can still be regarded as Motif-compliant. If you already have an application in mind, chances are you already know whether or not the main window layout is suited to the application; if you are in doubt, your best bet is to comply with the *Motif Style Guide*.

Before we start discussing the MainWindow widget, you should realize that this widget class does not create any of the widgets it manages. It merely facilitates managing the widgets in a way that is consistent with the *Style Guide*. In order to discuss the MainWindow widget, we are going to have to discuss a number of other widget classes and use them in examples. As a beginning chapter in a large book on Motif programming, this may seem like a bit much to handle, especially if you are completely unfamiliar with the Motif toolkit. We encourage you to branch off into other chapters whenever you find it necessary to do so. However, it is not our intention to explain these other widgets ahead of time, nor is it our assumption that you already understand them. The lack of an understanding of the other widgets should not interfere with our goal of describing the MainWindow widget and how it fits into the design of an application.

Creating a MainWindow

The MainWindow widget class is defined in *<Xm/MainW.h>*, which must be included whenever you create a MainWindow widget. As mentioned in Chapter 2, *The Motif Programming Model*, you should probably use a SessionShell or TopLevelShell widget as the parent of a MainWindow*. If the MainWindow is being used as the main application window, the SessionShell returned by `XtOpenApplication()`† (or another similar toolkit initialization function) is typically used as the parent. The function `XmCreateMainWindow()` can be used to create an instance of a MainWindow widget, as shown in the following code fragment‡:

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

main (int argc, char *argv[])
{
    Widget         app_shell, main_w;
```

* The ApplicationShell is considered deprecated in X11R6.

† `XtAppInitialize`(), `XtVaAppInitialize`() are now considered deprecated in X11R6.

‡ `XtVaAppInitialize`() is considered deprecated in X11R6.

```
    XtAppContext   app_context;
    /* Resources for the MainWindow */
    Arg            resource_values[...];
    Cardinal       num_values;         /* Number of resources applied */

    XtSetLanguageProc (NULL, NULL, NULL);
    app_shell = XtVaOpenApplication (&app_context, "App-Class", NULL, 0, &argc,
                                argv, NULL, sessionShellWidgetClass, NULL);
    main_w = XmCreateMainWindow (app_shell, "mw", resource_values, num_values);
    XtManageChild (main_w);
    XtRealizeWidget (app_shell);
    XtAppMainLoop (app_context);
}
```

The MainWindow class is subclassed from the ScrolledWindow class, which means that it inherits all the attributes of a ScrolledWindow, including its resources. A ScrolledWindow allows the user to view an arbitrary widget of any size by attaching horizontal and vertical ScrollBars to it. You can think of a MainWindow as a ScrolledWindow with the additional ability to have an optional menu bar, command area, and message area. Because the MainWindow is subclassed from the ScrolledWindow widget, we will be referring to some ScrolledWindow resources and disclosing some facts about the ScrolledWindow. For more information about the ScrolledWindow, see Chapter 10, *ScrolledWindows and ScrollBars.* You may eventually need to learn more about the ScrolledWindow widget to best make use of the MainWindow, but this chapter tries to present the fundamentals of the MainWindow widget, rather than focus on the ScrolledWindow.

While a MainWindow does control the sizes and positions of its widget children like any manager widget, the geometry management it performs is not the classic management style of other manager widgets. The MainWindow is a special-case object that handles only certain types of children and performs only simple widget positioning. It is designed to support the generic main window layout specified by the *Motif Style Guide*. Let's take a look at how the MainWindow can be used in an actual application. Example 4-1 demonstrates how the MainWindow widget fits into a typical application design.*

Example 4-1The show_pix.c program

```
/* show_pix.c -- A minimal example of a MainWindow. Use a Label as the
* workWindow to display a bitmap specified on the command line.
*/
#include <Xm/MainW.h>
#include <Xm/Label.h>
main (int argc, char *argv[])
{
    Widget         toplevel, main_w, label_w;
    XtAppContext   app_context;
    Pixmap         pixmap;
    Arg            al[4];
```

* XtVaAppInitialize() is considered deprecated in X11R6.

```
        Cardinal       ac = 0;

        XtSetLanguageProc (NULL, NULL, NULL);
        toplevel = XtVaOpenApplication (&app_context, "Demos", NULL, 0, &argc,
                                        argv, NULL, sessionShellWidgetClass, NULL);
        if (!argv[1]) {
        printf ("usage: %s bitmap-file\n", argv[0]);
        exit (1);
        }

        ac = 0;
        XtSetArg(al[ac], XmNscrollBarDisplayPolicy, XmAS_NEEDED); ac++;
        XtSetArg(al[ac], XmNscrollingPolicy, XmAUTOMATIC); ac++;
        main_w = XmCreateMainWindow (toplevel, "main_window", al, ac);

        /* Load bitmap given in argv[1] */
        pixmap = XmGetPixmap (XtScreen (toplevel), argv[1], BlackPixelOfScreen
                (XtScreen (toplevel)), WhitePixelOfScreen (XtScreen (toplevel)));
        if (pixmap == XmUNSPECIFIED_PIXMAP) {
        printf ("can't create pixmap from %s\n", argv[1]);
        exit (1);
        }
        /* Now create label using pixmap */
        ac = 0;
        XtSetArg(al[ac], XmNlabelType, XmPIXMAP); ac++;
        XtSetArg(al[ac], XmNlabelPixmap, pixmap); ac++;
        label_w = XmCreateLabel (main_w, "label", al, ac);

        /* set the label as the "work area" of the main window */
        XtVaSetValues (main_w, XmNworkWindow, label_w, NULL);
        XtManageChild (label_w);
        XtManageChild (main_w);
        XtRealizeWidget (toplevel);
        XtAppMainLoop (app_context);
    }
```

In this example, the MainWindow widget is not used to its full potential. It only contains one other widget, a Label widget, that is used to display a bitmap from the file specified as the first argument on the command line (`argv[1]`).* The Label widget is used as the work area window for the MainWindow. We did this intentionally to focus your attention on the scrolled-window aspect of the MainWindow widget. The following command line:

```
% show_pix /usr/X11R6/include/bitmaps/xlogo64
```

produces the output shown in Figure 4-2.

* `XtVaOpenApplication`() parses the command-line arguments that are used when the program is run. The command-line options that are specific to Xlib or Xt are evaluated and removed from the argument list. What is not parsed is left in `argv`; our program reads `argv`[1] as the name of a bitmap to display in the MainWindow.

Figure 4-2 Output of show_pix xlogo64

The file specified on the command line should contain X11 bitmap data, so that the application can create a pixmap. The pixmap is displayed in a Label widget, which has been specified as the `XmNworkWindow` of the MainWindow. As shown in Figure 4-2, the bitmap is simply displayed in the window. However, if a larger bitmap is specified, only a portion of the bitmap can be displayed, so ScrollBars are provided to allow the user to view the entire bitmap. The output of the command:

```
% show_pix /usr/X11R6/include/bitmaps/escherknot
```

is shown in Figure 4-3.

Figure 4-3 Output of show_pix escherknot

The bitmap is obviously too large to be displayed in the MainWindow without either clipping the image or enlarging the window. Rather than resize its own window to an unreasonable size, the MainWindow can display ScrollBars. This behavior is enabled by setting the MainWindow resources `XmNscrollBarDisplayPolicy` to `XmAS_NEEDED` and `XmNscrollingPolicy` to `XmAUTOMATIC`. These values automate the process whereby ScrollBars are managed when they are needed. If there is enough room for the entire bitmap to be displayed, the ScrollBars are not provided. Try resizing the *show_pix* window and see how the ScrollBars appear and disappear as needed. This behavior occurs as a result of setting `XmNscrollBarDisplayPolicy` to `XmAS_NEEDED`.

Since we do not specify a size for the MainWindow, the toolkit sets both the width and height to be `100` pixels. These default values are not a documented feature. Both the MainWindow and the ScrolledWindow suffer from the same problem: if you do not

specifically set the XmNwidth and XmNheight resources, the default size of the widget is not very useful.

The XmNscrollBarDisplayPolicy and XmNscrollingPolicy resources are inherited from the ScrolledWindow widget class. Because XmNscrollingPolicy is set to XmAUTOMATIC, the toolkit creates and manages the ScrollBars automatically. Another possible value for the resource is XmAPPLICATION_DEFINED, which implies that the application is going to create and manage the ScrollBars for the MainWindow and control all of the aspects of their functionality. Application-defined scrolling is the default style for the MainWindow widget, but it is unlikely that you will want to leave it that way in this instance: application-defined scrolling is usually required for hand-drawn X graphics, but since the Label widget knows how to draw itself, we can leave the scrolling policy as XmAUTOMATIC. For complete details on the different scrolling styles, see Chapter 10, *ScrolledWindows and ScrollBars*.

Using the application-defined scrolling policy does not necessarily require you to provide your own scrolling mechanisms. It simply relieves the MainWindow widget of the responsibility of handling the scrolling functionality. If you use a ScrolledList or ScrolledText widget as the work area, you should definitely leave the XmNscrollingPolicy as XmAPPLICATION_DEFINED, since these widgets manage their own ScrollBars. They will handle the scrolling behavior instead of the MainWindow. Example 4-2 shows an example of a program that uses a ScrolledList for the work area in a MainWindow widget.*

Example 4-2The main_list.c program

```
/* main_list.c -- Use the ScrolledList window as the feature
* component of a MainWindow widget.
*/

#include <Xm/MainW.h>
#include <Xm/List.h>

main (int argc, char *argv[])
{
    Widget         app_shell, main_w, list_w;
    XtAppContext   app_context;
    Pixmap         pixmap;

    XtSetLanguageProc (NULL, NULL, NULL);
    app_shell = XtVaOpenApplication (&app_context, "Demos", NULL, 0, &argc,
                                argv, NULL, sessionShellWidgetClass, NULL);
    main_w = XmCreateMainWindow (app_shell, "main_window", NULL, 0);
    list_w = XmCreateScrolledList (main_w, "main_list", NULL, 0);
    XtVaSetValues (list_w, XtVaTypedArg, XmNitems, XmRString,
                  "Red, Green, Blue, Orange, Maroon, Grey, Black, White", 53,
                  XmNitemCount, 8, XmNvisibleItemCount, 5, NULL);
```

* XtVaAppInitialize() is considered deprecated in X11R6.

```
    XtManageChild (list_w);
    /* set the list_w as the "work area" of the main window */
    XtVaSetValues (main_w, XmNworkWindow, XtParent (list_w), NULL);
    XtRealizeWidget (app_shell);
    XtAppMainLoop (app);
}
```

In order to simplify the application, we specified the items in the ScrolledList as a single string:

```
XtVaSetValues (list_w, XtVaTypedArg, XmNitems, XmRString,
               "Red, Green, Blue, Orange, Maroon, Grey, Black, White", 53,
               XmNitemCount, 8, XmNvisibleItemCount, 5, NULL);
```

This technique provides the easiest way to specify a list for a List widget. The items in a List widget must be specified as an array of compound strings. If we took the time to create each list item separately, we would have to create each compound string, assemble the array of `XmString` objects and specify it as the `XmNitems` resource, and then free each string separately after the widget was created. By using `XtVaTypedArg`, the whole list can be created in one line using the List widget's type converter to convert the string into a list of compound strings. We use this form of resource specification frequently in the book to simplify examples. See Volume 4, for a complete discussion on how this kind of type conversion is done. See Chapter 13, *The List Widget*, for details on the List widget; see Chapter 25, *Compound Strings*, for details on XmStrings.

It is important to note that while `XmCreateScrolledList()` creates both a ScrolledWindow widget and a List widget, it returns the List widget. As a result, we must use `XtParent()` to get access to the ScrolledWindow widget, so that it can be specified as the work area of the MainWindow. A common programming error with a ScrolledText or a ScrolledList widget is using the actual Text or List widget rather than its ScrolledWindow parent. Again, we refer you to Chapter 10, *ScrolledWindows and ScrollBars*, for a complete discussion of the use of ScrolledText and ScrolledList compound objects.

The MenuBar

Creating a MenuBar is a fairly complex operation, and one that is completely independent of the MainWindow itself. However, one of the principal reasons for using the MainWindow widget is that it manages the layout of a MenuBar. In this section, we demonstrate the simplest means of creating a MenuBar. Once a MenuBar has been created, you simply tell the MainWindow to include it in the window layout by specifying the MenuBar as the value of the `XmNmenuBar` resource for the MainWindow.

In the Motif toolkit, a MenuBar is not implemented as a separate widget, but as a set of CascadeButtons arranged horizontally in a RowColumn widget. Each CascadeButton is associated with a PulldownMenu that can contain PushButtons, ToggleButtons, Labels,

and Separators. The managing RowColumn widget has a resource setting indicating that it is being used as a MenuBar.You do not need to know any specific details about any of these widgets in order to create a functional MenuBar, since Motif provides convenience routines that allow you to create self-sufficient menu systems. While the specifics on creating PopupMenus, PulldownMenus, and MenuBars are covered in more detail in Chapter 20, *Interacting With the Window Manager*, the basic case that we present in this section is quite simple.

There are a variety of methods that you can use to create and manage a MenuBar, but the easiest method is to use the convenience menu creation routine provided by the Motif toolkit: `XmVaCreateSimpleMenuBar()`.* This function is demonstrated in the following code fragment:

```
XmString  file, edit, help;
Widget    menubar, main_w;
...
/* Create a simple MenuBar that contains three menus */
file = XmStringCreateLocalized ("File");
edit = XmStringCreateLocalized ("Edit");
help = XmStringCreateLocalized ("Help");
menubar = XmVaCreateSimpleMenuBar (main_w, "menubar",
                           XmVaCASCADEBUTTON, file, 'F',
                           XmVaCASCADEBUTTON, edit, 'E',
                           XmVaCASCADEBUTTON, help, 'H',
                           NULL);
XmStringFree (file);
XmStringFree (edit);
XmStringFree (help);
```

The output generated by this code is shown in Figure 4-4.

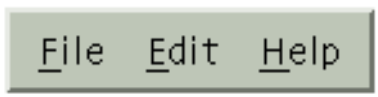


Figure 4-4 A simple MenuBar

Like the functions `XtVaSetValues()` and `XtVaCreateWidget()`, the routine `XmVaCreateSimpleMenuBar()` takes a variable-length argument list of configuration parameters. In addition to resource/value pairs, it also takes special arguments that specify the items in the MenuBar. You can specify RowColumn-specific resource/value pairs just as you would for any varargs routine. Once all the items in a MenuBar have been created, it must be managed using `XtManageChild()`.

* There is also a non-varargs version of this function. It requires you to create each of the buttons in the MenuBar individually and associate it with a PulldownMenu via resources. The varargs function is often easier to use.

If you are specifying an item in the MenuBar, the first parameter is a symbolic constant that identifies the type of the item. Since CascadeButtons are the only elements that can display PulldownMenus, the first parameter should always be set to XmVaCASCADEBUTTON. The label of the CascadeButton is given by the second parameter, which must be a compound string. In the above example, the variable *file* contains a compound string that contains the text File. The third parameter specifies an optional mnemonic character for the CascadeButton that can be used to post the menu from the keyboard. The mnemonic for the *File* menu is F. By convention, the first letter of a menu or menu item label is used as the mnemonic.

We use the compound string creation function, XmStringCreateLocalized(), to create the compound strings for the menu labels. This function creates a compound string with the text encoded in the current locale. For a complete discussion of compound strings, see Chapter 25.

Since you are not creating each CascadeButton using the normal creation routines, you are not returned a handle to each button.You might think that the label string that you assign to each button is used as the widget's name, but this is not the case. The buttons are created sequentially, so the MenuBar assigns the name button_*n* to each button. The value *n* is the position of the button in the MenuBar, where positions are numbered starting with 0 (zero).We will discuss how you can specify resources for items on the MenuBar later in the chapter.

Do not attempt to install callback routines on the CascadeButtons themselves. If you need to know when a particular menu is popped up, you should use the XmNpopupCallback on the MenuShell that contains the PulldownMenu associated with the CascadeButton. The popup and popdown callback lists are described briefly in Chapter 7, *Custom Dialogs*; for more information, see Volume 4, *X Toolkit Intrinsics Programming Manual*.

Creating a PulldownMenu

Every CascadeButton in a MenuBar must have a PulldownMenu associated with it. You can create the items in a PulldownMenu using a method that is similar to the one for creating a MenuBar. A PulldownMenu can be created using the function XmVaCreateSimplePulldownMenu(). This routine is slightly more involved than XmVaCreateSimpleMenuBar(). The routine takes the following form:

```
Widget XmVaCreateSimplePulldownMenu ( Widget         parent,
                                      String         name,
                                      int            post_from_button,
                                      XtCallbackProc callback,...)
```

The *post_from_button* parameter specifies the CascadeButton that posts the PulldownMenu. This parameter is an index (starting at zero) into the array of CascadeButtons in the *parent* widget, which should be a MenuBar. The *name* parameter specifies the widget name for the RowColumn widget that is the PulldownMenu. This

name is not the title of the CascadeButton associated with the menu. The MenuShell that contains the PulldownMenu uses the same name with _popup appended to it. The *callback* parameter specifies a function that is invoked whenever the user activates any of the items in the menu. The rest of the arguments to XmVaCreateSimplePulldownMenu() are either RowColumn resource/value pairs or special arguments that specify the items in the PulldownMenu.

You should not manage a PulldownMenu after you create it because you do not want it to appear until it is posted by the user. The CascadeButton that posts the menu handles managing the menu when it needs to be displayed. The following code fragment shows the use of XmVaCreateSimplePulldownMenu() to create a PulldownMenu:

```
XmString open, save, quit, quit_acc;
Widget menubar, menu;
...
/* First menu is the File menu -- callback is file_cb() */
open = XmStringCreateLocalized ("Open...");
save = XmStringCreateLocalized ("Save...");
quit = XmStringCreateLocalized ("Quit");
quit_acc = XmStringCreateLocalized ("Ctrl-C");
menu = XmVaCreateSimplePulldownMenu (menubar, "file_menu", 0, file_cb,
                              XmVaPUSHBUTTON, open, 'O', NULL, NULL,
                              XmVaPUSHBUTTON, save, 'S', NULL, NULL,
                              XmVaSEPARATOR,
                             XmVaPUSHBUTTON, quit, 'Q', "Ctrl<Key>c", quit_acc,
                              NULL);
XmStringFree (open);
XmStringFree (save);
XmStringFree (quit);
XmStringFree (quit_acc);
...
```

Unlike a MenuBar, which can only contain CascadeButtons, a PulldownMenu can contain a number of different types of elements. As with XmVaCreateSimpleMenuBar(), these elements are specified by a symbolic constant that identifies the type of the item. The symbolic constant is followed by a variable number of additional parameters that depend on the type of the menu item.You can use the following values to specify the items in a PulldownMenu:

XmVaPUSHBUTTON
: The item is a PushButton. It takes four additional parameters: a compound string label, a mnemonic, an accelerator, and a compound string that contains a text representation of the accelerator. When the PushButton is selected, the *callback* routine is called. It is passed an integer value as client_data that indicates the item on the Pulldown-Menu that was activated. The value is an index into the menu that ranges from 0 to n-1; if client_data is two, then the third item in the menu was selected.

XmVaTOGGLEBUTTON
: The item is a ToggleButton. It takes the same four additional parameters as described for XmVaPUSHBUTTON. When the ToggleButton is selected, the value of the button is toggled and the *callback* routine is called. The client_data that is passed to the callback routine is handled the same as for PushButtons.

XmVaCHECKBUTTON
: This value is identical to XmVaTOGGLEBUTTON.

XmVaRADIOBUTTON
: The item is a ToggleButton with RadioBox characteristics, which means that only one item in the menu can be set at a time. The PulldownMenu does not enforce this behavior, so you must either handle it yourself or specify other RowColumn resources to make the menu function like a RadioBox. We demonstrate creating a menu with RadioBox behavior later in the chapter. This value takes the same additional parameters and deals with the callback routine in the same way as ToggleButtons.

XmVaCASCADEBUTTON
: The item is a CascadeButton, which is usually associated with a pullright menu. The value takes two additional parameters: a compound string label and a mnemonic. Pullright menus are, ironically, easier to implement and manage using the not-so-simple menu creation routines described in Chapter 19, *Menus*.

XmVaSEPARATOR
: The item is a Separator and it does not take any additional parameters. Since separators cannot be selected, the callback routine is not called for this item. Adding a separator does not affect the item count with respect to the client_data values that are passed to the callback routine for other menu items.

XmVaSINGLE_SEPARATOR
: This value is identical to XmVaSEPARATOR.

XmVaDOUBLE_SEPARATOR
: This value is identical to XmVaSEPARATOR, except that the separator widget displays a double line instead of a single line.

XmVaTITLE
: The item is a Label that is used to create a title in a menu. It takes one additional parameter: a compound string label. The item is not selectable, so it does not have a mnemonic associated with it and it does not call the callback routine. Adding a title does not affect the item count with respect to the client_data values that are passed to the callback routine for other menu items.

Just as with the CascadeButtons in a MenuBar, the labels associated with each menu item are not the names of the widgets themselves. The names of the buttons are button_*n*, where *n* is the position of the button in the menu (starting with zero). Similarly, the names of the separators and the titles are separator_*n* and label_*n*, respectively. We will

discuss how you can use resources to specify labels, mnemonics, and accelerators for menus and menu items later in the chapter.

Menus are not intended to be changed dynamically. You should not add, delete, or modify the menus on the MenuBar or the menu items in PulldownMenus once an application is running. Rather than delete an item on a menu when it is not appropriate, you should change the sensitivity of the item using XmNsensitive. The menus in an application should be static in the user's eyes; changing the menus would be like changing the functionality of the program while the user is running it. The one exception to this guideline involves menu items that correspond to dynamic objects. For example, if you have a menu that contains an item for each application that is running on a display, it is acceptable for the items on the menu to change to reflect the current state of the display.

SimpleMenu Callback Routines

The callback routine associated with the *File* menu shown earlier is invoked whenever the user selects any of the buttons in the menu. Just like any callback, the routine takes the form of an XtCallbackProc:

```
void file_cb (Widget widget, XtPointer client_data, XtPointer call_data)
```

The widget parameter is a handle to the widget that was selected in the menu. The client_data parameter is the index of the menu item in the menu. The call_data parameter is a pointer to a callback structure that contains data about the callback. Both the client_data and call_data parameters should be cast to their appropriate types before the data that they contain is accessed.

Every Motif callback routine has a callback structure associated with it. The simplest such structure is of type XmAnyCallbackStruct, which has the following form:

```
typedef struct {
    int     reason;
    XEvent  *event;
} XmAnyCallbackStruct;
```

All of the Motif callback structures have these two fields, but they also contain more detailed information about why the callback function was invoked. The callback routine for the *File* menu would be passed an XmPushButtonCallbackStruct, since all of the menu items are PushButtons. This structure has the following form:

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

The `click_count` field is not normally used when a PushButton is in a menu. If one of the items in the menu were a ToggleButton, the `call_data` parameter would be of type `XmToggleButtonCallbackStruct`, which has the following form:

```
typedef struct {
    int     reason;
    XEvent  *event;
    int     set;
} XmToggleButtonCallbackStruct;
```

The `set` field indicates whether the item was selected (turned on) or deselected (turned off).

When a menu contains both PushButtons and ToggleButtons, you can determine which of the two callback structures the `call_data` parameter points to by examining the `reason` field. Since all callback structures have this field, it is always safe to query it. As its name implies, this field indicates why the callback routine was invoked. The value of this field may also indicate the type of the widget that invoked the callback. While we can always determine the type of the `widget` parameter by using the macro `XtIsSubClass()`, using the `reason` field is more straightforward. The PushButton widget uses the value `XmCR_ACTIVATE` to indicate that it has been activated, while the ToggleButton uses `XmCR_VALUE_CHANGED` to indicate that its value has been changed. In our example, the `reason` will always be `XmCR_ACTIVATE`, since there are only PushButtons in the menu. If there were also ToggleButtons in the menu, we would know that the callback was invoked by a ToggleButton if the value were `XmCR_VALUE_CHANGED`.

The `event` field in all of the callback structures is a pointer to an `XEvent` structure. The `XEvent` identifies the actual event that caused the callback routine to be invoked. In this example, the event is not of particular interest.

In the callback function, you can choose to do whatever is appropriate for the item that was selected. The callback structure is probably not going to be of that much help in most cases. However, the `client_data` passed to the function can be used to identify which of the menu items was selected. The following code fragment demonstrates the use of `client_data`:

```
/* a menu item from the "File" pulldown menu was selected */
void file_cb (Widget widget, XtPointer client_data, XtPointer call_data)
{
    extern void OpenNewFile(void), SaveFile(void);
    int item_no = (int) client_data;

    if (item_no == 0)
        /* the "new" button */
        OpenNewFile ();
    else if (item_no == 1)
        /* the "save" button */
        SaveFile();
    else
```

```
        /* the "Quit" button */
        exit (0);
}
```

The callback routines for menu items should be as simple as possible from a structural point of view. A well-designed application should have application-specific entry points such as `OpenNewFile()` and `SaveFile()`, as shown in the previous example. These routines should be defined in separate files that are not necessarily associated with the user-interface portion of the program. The use of modular programming techniques helps considerably when an application is being maintained by a large group of people or when it needs to be ported to other user-interface platforms.

A Sample Application

Let's examine an example program that integrates what we have discussed so far. Example 4-3 modifies the behavior of our first example, which displayed an arbitrary pixmap, by allowing the user to change the bitmap dynamically using a Motif FileSelectionDialog. The program also allows the user to dynamically change the color of the bitmap using a PulldownMenu. As you can see by the size of the program, adding these two simple features is not trivial. Many functions and widgets are required in order to make the program functional. As you read the example, don't worry about unknown widgets or details that we haven't addressed just yet; we will discuss them afterwards. For now, just try to identify the familiar parts and see how everything works together.*

Example 4-3The dynapix.c program

```
/* dynapix.c -- Display a bitmap in a MainWindow, but allow the user
** to change the bitmap and its color dynamically. The design of the
** program is structured on the pulldown menus of the menubar and the
** callback routines associated with them. To allow the user to choose
** a new bitmap, the "Open" button pops up a FileSelectionDialog where
** a new bitmap file can be chosen.
*/
#include <Xm/MainW.h>
#include <Xm/Label.h>
#include <Xm/MessageB.h>
#include <Xm/FileSB.h>

/* Globals: the toplevel window/widget and the label for the bitmap.
** "colors" defines the colors we use, "cur_color" is the current
** color being used, and "cur_bitmap" references the current bitmap
** file.
*/
Widget  toplevel, label;
String  colors[] = {"Black", "Red", "Green", "Blue"};
Pixel   cur_color;
```

* `XtVaAppInitialize`() is considered deprecated in X11R6. `XmStringGetLtoR`() is deprecated in Motif 2.0, and is replaced by `XmStringUnparse`().

```
/* make large enough for full pathnames */
char    cur_bitmap[1024] = "xlogo64";

main (int argc, char *argv[])
{
    Widget        main_w, menubar, menu, widget;
    XtAppContext  app;
    Pixmap        pixmap;
    XmString      file, edit, help, open, quit, red, green, blue, black;
    void          file_cb(Widget, XtPointer, XtPointer);
    void          change_color(Widget, XtPointer, XtPointer);
    void          help_cb(Widget, XtPointer, XtPointer);
    Arg           al[10];
    Cardinal      ac = 0;

    XtSetLanguageProc (NULL, NULL, NULL);
    /* Initialize toolkit and parse command line options. */
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv,
                                    NULL, sessionShellWidgetClass, NULL);
    /* main window contains a MenuBar and a Label displaying a pixmap */
    ac = 0;
    XtSetArg(al[ac], XmNscrollBarDisplayPolicy, XmAS_NEEDED); ac++;
    XtSetArg(al[ac], XmNscrollingPolicy, XmAUTOMATIC); ac++;
    main_w = XmCreateMainWindow (toplevel, "main_window", al, ac);
    /* Create a simple MenuBar that contains three menus */
    file = XmStringCreateLocalized ("File");
    edit = XmStringCreateLocalized ("Edit");
    help = XmStringCreateLocalized ("Help");
    menubar = XmVaCreateSimpleMenuBar (main_w, "menubar",
                                    XmVaCASCADEBUTTON, file, 'F',
                                    XmVaCASCADEBUTTON, edit, 'E',
                                    XmVaCASCADEBUTTON, help, 'H',
                                    NULL);
    XmStringFree (file);
    XmStringFree (edit);
    /* don't free "help" compound string yet -- reuse it later */
    /* Tell the menubar which button is the help menu */
    if ((widget = XtNameToWidget (menubar, "button_2")) != (Widget) 0)
    XtVaSetValues (menubar, XmNmenuHelpWidget, widget, NULL);
    /* First menu is the File menu -- callback is file_cb() */
    open = XmStringCreateLocalized ("Open...");
    quit = XmStringCreateLocalized ("Quit");
    XmVaCreateSimplePulldownMenu (menubar, "file_menu", 0, file_cb,
                              XmVaPUSHBUTTON, open, 'N', NULL, NULL,
                              XmVaSEPARATOR,
                              XmVaPUSHBUTTON, quit, 'Q', NULL, NULL,
                              NULL);
    XmStringFree (open);
    XmStringFree (quit);
    /* Second menu is the Edit menu -- callback is change_color() */
    black = XmStringCreateLocalized (colors[0]);
    red = XmStringCreateLocalized (colors[1]);
    green = XmStringCreateLocalized (colors[2]);
    blue = XmStringCreateLocalized (colors[3]);
```

```
    menu = XmVaCreateSimplePulldownMenu (menubar, "edit_menu", 1, change_color,
                                XmVaRADIOBUTTON, black, 'k', NULL, NULL
                                XmVaRADIOBUTTON, red, 'R', NULL, NULL,
                                XmVaRADIOBUTTON, green, 'G', NULL, NULL,
                                XmVaRADIOBUTTON, blue, 'B', NULL, NULL,
                                /* RowColumn resources to enforce */
                                XmNradioBehavior, True,
                                /* radio behavior in Menu */
                                XmNradioAlwaysOne, True,
                                NULL);
    XmStringFree (black);
    XmStringFree (red);
    XmStringFree (green);
    XmStringFree (blue);
    /* Initialize menu so that "black" is selected. */
    if ((widget = XtNameToWidget (menu, "button_0")) != (Widget) 0)
             XtVaSetValues (widget, XmNset, XmSET, NULL);
    /* Third menu is the help menu -- callback is help_cb() */
    XmVaCreateSimplePulldownMenu (menubar, "help_menu", 2, help_cb,
                   XmVaPUSHBUTTON, help, 'H', NULL, NULL,  NULL);
    XmStringFree (help); /* we're done with it; now we can free it */
    XtManageChild (menubar);
    /* user can still specify the initial bitmap */
    if (argv[1])
         (void) strcpy (cur_bitmap, argv[1]);
    /* initialize color */
    cur_color = BlackPixelOfScreen (XtScreen (toplevel)),
    /* create initial bitmap */
    pixmap = XmGetPixmap (XtScreen (toplevel), cur_bitmap, cur_color,
    WhitePixelOfScreen (XtScreen (toplevel)));
    if (pixmap == XmUNSPECIFIED_PIXMAP) {
        puts ("can't create initial pixmap");
        exit (1);
    }
    /* Now create label using pixmap */
    ac = 0;
    XtSetArg(al[ac], XmNlabelType, XmPIXMAP); ac++;
    XtSetArg(al[ac], XmNlabelPixmap, pixmap); ac++;
    label = XmCreateLabel (main_w, "label", al, ac);
    XtManageChild (label);
    /* set the label as the "work area" of the main window */
    XtVaSetValues (main_w, XmNmenuBar, menubar, XmNworkWindow, label, NULL);
    XtManageChild (main_w);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

/*
** Popdown routine for the File Selection Box
*/
void hide_fsb (Widget w, XtPointer client_data, XtPointer call_data){
/* This also pops down the XmDialogShell parent of the
** File Selection Box
*/
```

```
    XtUnmanageChild (w);
}

/* Any item the user selects from the File menu calls this function.
** It will either be "Open" (item_no == 0) or "Quit" (item_no == 1).
*/
void file_cb( Widget    widget,      /* menu item that was selected */
              XtPointer client_data, /* the index into the menu */
              XtPointer call_data)   /* unused */
{
    static Widget   dialog;   /* make it static for reuse */
    void            load_pixmap(Widget, XtPointer, XtPointer);
    int             item_no = (int) client_data;

    if (item_no == 1) /* the "quit" item */
         exit (0);
    /* "Open" was selected. Create a Motif FileSelectionDialog w/callback */
    if (!dialog) {
        dialog = XmCreateFileSelectionDialog (toplevel, "file_sel", NULL, 0);
        XtAddCallback (dialog, XmNokCallback, load_pixmap, NULL);
        XtAddCallback (dialog, XmNcancelCallback, hide_fsb, NULL);
    }
    /* This also pops up the XmDialogShell parent of the File selection box */
    XtManageChild (dialog);
}

/* The OK button was selected from the FileSelectionDialog (or, the user
** double-clicked on a file selection). Try to read the file as a bitmap.
** If the user changed colors, we call this function directly from
** change_color()to reload the pixmap. In this case, we pass NULL as the
** callback struct so we can identify this special case.
*/
void load_pixmap (Widget dialog, XtPointer client_data, XtPointer call_data)
{
    Pixmap    pixmap;
    char      *file = NULL;
    XmFileSelectionBoxCallbackStruct *cbs;

    cbs = (XmFileSelectionBoxCallbackStruct *) call_data;
    if (cbs) {
        file = (char *) XmStringUnparse (cbs->value, NULL,
                        XmCHARSET_TEXT, XmCHARSET_TEXT,
                        NULL, 0, XmOUTPUT_ALL);
    if (file == (char *) 0)
        return; /* internal error */
    (void) strcpy (cur_bitmap, file);
    XtFree (file); /* free allocated data from XmStringUnparse() */
    }
    pixmap = XmGetPixmap (XtScreen (toplevel), cur_bitmap, cur_color,
                          WhitePixelOfScreen (XtScreen (toplevel)));
    if (pixmap == XmUNSPECIFIED_PIXMAP)
        printf ("Can't create pixmap from %s\n", cur_bitmap);
    else {
        Pixmap old;
```

```
        XtVaGetValues (label, XmNlabelPixmap, &old, NULL);
        XmDestroyPixmap (XtScreen (toplevel), old);
        XtVaSetValues (label, XmNlabelType, XmPIXMAP, XmNlabelPixmap, pixmap,
                       NULL);
    }
}
/* called from any of the "Edit" menu items. Change the color of the
** current bitmap being displayed. Do this by calling load_pixmap().
*/
void change_color ( Widget     widget,      /* selected menu item  */
                    XtPointer  client_data, /* the index into the menu */
                    XtPointer  call_data)   /* unused */
{
    XColor   xcolor, unused;
    Display  *dpy = XtDisplay (label);
    Colormap cmap = DefaultColormapOfScreen (XtScreen (label));
    int      item_no = (int) client_data;

    if (XAllocNamedColor (dpy, cmap, colors[item_no], &xcolor, &unused) == 0 ||
                        cur_color == xcolor.pixel)
         return;

    cur_color = xcolor.pixel;
    load_pixmap (widget, NULL, NULL);
}

#define MSG \
"Use the FileSelection dialog to find bitmap files to\n\
display in the scrolling area in the main window. Use\n\
the edit menu to display the bitmap in different colors."

/* The help button in the help menu from the menubar was selected.
** Display help information defined above for how to use the program.
** This is done by creating a Motif information dialog box. Again,
** make the dialog static so we can reuse it.
*/
void help_cb (Widget widget, XtPointer client_data, XtPointer call_data)
{
    static Widget dialog;
    if (!dialog) {
        Arg args[5];
        int n = 0;
        XmString msg = XmStringCreateLocalized (MSG);
        XtSetArg (args[n], XmNmessageString, msg); n++;
       dialog = XmCreateInformationDialog (toplevel, "help_dialog", args, n);
    }
    /* This also pops up the XmDialogShell parent of the XmMessageBox */
    XtManageChild (dialog);
}
```

The output of the program is shown in Figure 4-5.

Figure 4-5 Output of dynapix.c

The beginning of the program is pretty much as expected. After the toolkit is initialized, the MainWindow and the MenuBar are created the same way as in the previous examples. Just after the MenuBar is created, however, we make the following calls:

```
if ((widget = XtNameToWidget (menubar, "button_2")) != (Widget) 0)
              XtVaSetValues (menubar, XmNmenuHelpWidget, widget, NULL);
```

The purpose of these statements is to inform the MenuBar which of its CascadeButtons contains the *Help* menu. Setting the MenuBar's `XmNmenuHelpWidget` resource to the CascadeButton returned by `XtNameToWidget()` causes the MenuBar to position the menu specially. The *Help* menu is placed at the far right on the MenuBar; this position is necessary for the application to conform to Motif style guidelines. For details on how to support a help system, see Chapter 7, *Custom Dialogs* and Chapter 27, *Advanced Dialog Programming.*

PulldownMenus are created next in the expected manner. The only variation is for the *Edit* menu, where each item in the menu represents a color. Since only one color can be used at a time, the color that is currently being used is marked with a diamond-shape indicator*. In order to get this radio-box behavior, each menu item in the PulldownMenu is a `XmVaRADIOBUTTON` and the menu is told to treat the items as a RadioBox. The analogy is that of an old car radio, where selecting a new station causes the other selectors to pop out. Just as you can only have the radio tuned to one station at a time, you may only have one color set at a time. The RadioBox functionality is managed automatically by the RowColumn widget that is used to implement the PulldownMenu. Setting the `XmNradioBehavior` and `XmNradioAlwaysOne` RowColumn resources to `True` provides the RadioBox behavior. See Chapter 12, *Labels and Buttons*, for a complete description and further examples of this type of behavior. Figure 4-6 shows the RadioBox-style *Edit* menu.

* From Motif 2.0 onwards, the shape partly depends upon the XmDisplay object `XmNenableToggleVisual` resource, and may appear round, as in Figure 4-6.

Figure 4-6 The Edit menu for dynapix.c

Although the RowColumn manages the RadioBox automatically, we need to turn the radio on by setting the initial color. After the PulldownMenu is created, the menu (RadioBox) is initialized so that its first item is selected, since we know that we are using black as the initial color. `XtNameToWidget()` is used again to get the appropriate button from the menu. Since the menu items were created using `XmVaRADIOBUTTON`, the widget that is returned is a ToggleButton.The `XmNset` resource is used to turn the button on. Once the menu has been initialized, the Motif toolkit handles everything automatically.

Note that when we create the *Help* menu, there is only one item in the menu. You might think that it is redundant to have a single *Help* item in the *Help* menu, but this design is an element of Motif style. The *Motif Style Guide* states that items on the MenuBar should always post PulldownMenu, not perform application actions directly.

It is important to note that `XmVaCreateSimplePulldownMenu()` returns the RowColumn widget that contains the items in the menu, even though the routine creates both the RowColumn widget and its MenuShell parent. The routine does not return the MenuShell widget that is actually popped up and down when the menu posted. To get a handle to that widget, you need to use `XtParent()` on the RowColumn widget. This design makes sense, since you need access to the RowColumn widget much more often than you need access to the MenuShell.

Once all of the items have been installed, the MenuBar is managed using XtManageChild(). The approach to creating MenuBars, PulldownMenus, menu items, and their associated callback routines that we have described here is meant to be simple and straightforward. In some cases, you may find that these techniques are too limiting. For example, you cannot specify different callback routines for different items in the same menu, you cannot pass different client data for different items, and you cannot name the widgets individually. The most inconvenient aspect of this method, however, is that it requires so much redundant code in order to build a realistically sized MenuBar. Our intent here is to introduce the basic concepts of menus and to demonstrate the recommended design approach for applications. We describe how the menu creation process can be generalized for large menu systems in Chapter 19, *Menus*.

The rest of Example 4-3 is composed of callback routines that are used by the PulldownMenu items. For example, when the user selects either of the items in the *File* menu, the function `file_cb()` is called. If the *Quit* item is selected, the `client_data` parameter is`1` and the program exits. If the *Open* item is selected, `client_data` is `0` and a FileSelectionDialog is popped up to allow the user to select a new bitmap file. The dialog is created using the convenience routine `XmCreateFileSelectionDialog()`, which produces the results shown in Figure 4-7. Two callback routines are installed for the dialog: `load_pixmap()`, which is called when the user presses the *OK* button, and `hide_fsb()`, which is called when the user selects the *Cancel* button. For more detailed information on the FileSelectionDialog, see Chapter 6, *Selection Dialogs*.

Figure 4-7 File FileSelectionDialog for dynapix.c

The `load_pixmap()` function loads a new bitmap from a file and displays it in the Label widget. This function uses the same method for loading a pixmap as was used earlier in `main()`.Since the function is invoked as a callback by the FileSelectionDialog, we need to get the value of the file selection. The value is taken from the `value` field of the FileSelectionDialog's callback structure, `XmFileSelectionBoxCallbackStruct`. Since the filename is represented as a compound string, it must be converted to a character string. The conversion is done using `XmStringUnparse()`*, which creates a regular C string for use by `XmGetPixmap()`. The `load_pixmap()` routine is also called directly from `change_color()`, so we need to check the `call_data` parameter. This parameter is `NULL` if the routine is not invoked as a callback.

* `XmStringGetLtoR`() is considered deprecated from Motif 2.0 onwards.

If XmGetPixmap() succeeds, we get the old pixmap and destroy it using XmDestroyPixmap() before we install the new pixmap. XmGetPixmap() loads and caches a pixmap. If the function is called more than once for a given image, it returns the cached image, which saves space because a new version of the pixmap is not allocated for each call. XmDestroyPixmap() decrements the reference count for the image; if the reference count reaches to zero, the pixmap is actually destroyed. Otherwise, another reference to it may exist, so nothing is done. It is important to use these two functions in conjunction with each other. However, if you use other pixmap-loading functions to create pixmaps, you cannot use XmDestroyPixmap() to free them.

The function change_color() is used as the callback routine for items in the *Edit* menu. The names of the colors are stored in the colors array. The index of a color in this array is the same as the index of the corresponding menu item in the menu.The color name is parsed and loaded using XAllocNamedColor(), provided that the string exists in the RGB database (usually */usr/X11R6/lib/rgb.txt*). If the routine is successful, it returns a non-zero status and the XColor structure is filled with the RGB data and pixel value. In this case, load_pixmap() is called to reload the pixmap with the new color. If XAllocNamedColor() returns zero, or if the returned pixel value is the same as the current one, change_color() returns, as there is no point in reloading an identical pixmap. For additional information about loading and using colors, see Volume 1, and Volume 2.

The help_cb() function is the callback routine for the *Help* menu item on the *Help* menu. It simply displays an InformationDialog that contains a message describing how to use the program. See Chapter 5, *Introduction to Dialogs*,and Chapter 27, *Advanced Dialog Programming*, for a complete description of these dialogs and suggestions on implementing a functional help system.

The Command and Message Areas

We have already covered most of what you need to know about the MainWindow of an application in this chapter and Chapter 3, *Overview of the Motif Toolkit*. The material in the rest of the chapter is considered somewhat advanced, so you could skip the remaining sections and be relatively secure in moving onto the next chapter. The remaining material provides details about the MainWindow widget that need to be discussed in order to make this chapter complete.

The greatest difficulty with the command and message areas of the MainWindow is that these objects are better defined in the Motif specification than in the Motif toolkit. The command area is intended to support a tty-style command-line interface to an application. The command area is not supposed to act like *xterm* or any sort of terminal emulator; it is just a single-line text area for entering individually typed commands for an application.The message area is just an output-only area that is used for error and status messages as needed by an application. While both of these areas are optional MainWindow elements, the

message area is usually more common than the command area. Nevertheless, let's begin by discussing the command area.

A command area is especially convenient for applications that are being converted from a tty-style interface to a graphical user interface. Properly converted, such applications can do rather well as GUI-based programs, although the conversion can be more difficult than you might expect. For example, a PostScript interpreter could be implemented using a command area in the MainWindow. However, since PostScript is a verbose language, it does not work well with single-line text entry fields.

Example 4-4 shows how the command area can be used to allow the user to input standard UNIX commands. The output of the commands is displayed in the ScrolledText object, which is the work area of the MainWindow. For simplicity, we've kept the MenuBar small so as to dedicate most of the program to the use of the command area.*

Example 4-4The cmd_area.c program

```
/* cmd_area.c -- use a ScrolledText object to view the
** output of commands input by the user in a Command window.
*/

#include <Xm/Text.h>
#include <Xm/MainW.h>
#include <Xm/Command.h>
#include <stdio.h>  /* For popen() */

/* main() -- initialize toolkit, create a main window, menubar,
** a Command Area and a ScrolledText to view the output of commands.
*/

main (int argc, char *argv[])
{
    Widget         toplevel, main_w, menubar, menu, command_w, text_w;
    XtAppContext   app;
    XmString       file, quit;
    extern void    exit(int);
    void           exec_cmd(Widget, XtPointer, XtPointer);
    Arg            args[5];
    int            n = 0;

    XtSetLanguageProc (NULL, NULL, NULL);

    /* initialize toolkit and create toplevel shell */
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv, NULL,
                                sessionShellWidgetClass, NULL);

    (void) close (0); /* don't let commands read from stdin */
```

* XtVaAppInitialize() is considered deprecated in X11R6. XmStringGetLtoR() is deprecated in Motif 2.0: XmStringUnparse() is the preferred function.

```
    /* MainWindow for the application -- contains menubar, ScrolledText
    ** and CommandArea (which prompts for filename).
    */
    n = 0;
    XtSetArg(args[n], XmNcommandWindowLocation,
                        XmCOMMAND_BELOW_WORKSPACE); n++;
    main_w = XmCreateMainWindow (toplevel, "main_w", args, n);

    /* Create a simple MenuBar that contains one menu */
    file = XmStringCreateLocalized ("File");
    menubar = XmVaCreateSimpleMenuBar (main_w, "menubar",
                        XmVaCASCADEBUTTON, file, 'F', NULL);
    XmStringFree (file);

    /* "File" menu has only one item (Quit), so make callback exit() */
    quit = XmStringCreateLocalized ("Quit");
    menu = XmVaCreateSimplePulldownMenu (menubar, "file_menu", 0,
                        (void (*)()) exit,
                        XmVaPUSHBUTTON, quit, 'Q', NULL, NULL, NULL);
    XmStringFree (quit);

    /* Menubar is done -- manage it */
    XtManageChild (menubar);

    /* Create ScrolledText -- this is work area for the MainWindow */
    n = 0;
    XtSetArg (args[n], XmNrows, 24);  n++;
    XtSetArg (args[n], XmNcolumns, 80);  n++;
    XtSetArg (args[n], XmNeditable, False);  n++;
    XtSetArg (args[n], XmNeditMode, XmMULTI_LINE_EDIT); n++;
    text_w = XmCreateScrolledText (main_w, "text_w", args, n);
    XtManageChild (text_w);

    /* store text_w as user data in "File" menu for file_cb() callback */
    XtVaSetValues (menu, XmNuserData, text_w, NULL);

    /* Create the command area -- this must be a Command class widget */
    file = XmStringCreateLocalized ("Command:");
    n = 0;
    XtSetArg(args[n], XmNpromptString, file); n++;
    command_w = XmCreateCommand (main_w, "command_w", args, n);
    XmStringFree (file);
    XtAddCallback (command_w, XmNcommandEnteredCallback, exec_cmd,
                        (XtPointer) text_w);
                        XtVaSetValues (command_w,
                        XmNmenuBar,  menubar,
                        XmNcommandWindow,  command_w,
                        XmNworkWindow,  XtParent (text_w),
                        NULL);
    XtManageChild (command_w);
    XtManageChild (main_w);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}
```

```
    /* execute the command and redirect output to the ScrolledText window */
    voidexec_cmd ( Widget    cmd_widget,  /* command, not Text widget */
                   XtPointer  client_data, /* passed text_w client_data */
                   XtPointer  call_data)
    {
        char            *cmd, buf[BUFSIZ];
        XmTextPosition  pos;
        FILE            *pp, *popen();
        Widget          text_w = (Widget) client_data;
        XmCommandCallbackStruct *cbs = (XmCommandCallbackStruct *) call_data;

        cmd = (char *) XmStringUnparse (cbs->value, XmFONTLIST_DEFAULT_TAG,
XmCHARSET_TEXT, XmCHARSET_TEXT, NULL, 0, XmOUTPUT_ALL);

        if (!cmd || !*cmd) {
            /* nothing typed? */
            if (cmd)
                XtFree (cmd);
            return;
        }
        /* make sure the file is a regular text file and open it */
        if ((pp = popen (cmd, "r")) == (FILE *) 0)
            perror (cmd);
        XtFree (cmd);
        if (pp == (FILE *) 0)
            return;
        /* Clear output from any previous command */
        XmTextSetString (text_w, “”);

        /* put the output of the command in the Text widget by reading
        ** until EOF (meaning that the command has terminated).
        */
        for (pos = 0; fgets (buf, sizeof (buf), pp) ; pos += strlen (buf))
            XmTextReplace (text_w, pos, pos, buf);

        (void) pclose (pp);
    }
```

This example uses a Command widget for the command area. The output of the program is shown in Figure 4-8. The Command widget provides a command entry area and a command history area. However, you do not necessarily have to use a Command widget for the command area. A TextField widget can be used instead to provide a simple command area.

Figure 4-8 Output from the cmd_area program

When we created the MainWindow, we set the XmNcommandWindowLocation resource to XmCOMMAND_BELOW_WORKSPACE, which caused the command area to be placed below the work window. Although the default value of the resource is XmCOMMAND_ABOVE_WORKSPACE, the *Style Guide* recommends that the command area be positioned beneath the work window, rather than above it. You need to explicitly set the value of XmNcommandWindowLocation to ensure that the command area is positioned appropriately.

Note that we use the ScrolledWindow that is created by XmCreateScrolledText() for the work window, rather than the scrolling area provided by the MainWindow. Since XmCreateScrolledText() returns a Text widget, we are careful to use the parent of the Text widget for the XmNworkWindow resource of the MainWindow. We set the areas of the MainWindow using the standard XtVaSetValues() routine*.

Note that it is not entirely necessary to explicitly specify the roles each child of the MainWindow will perform. When you create a widget as a child of a MainWindow widget, the MainWindow checks the type of the widget you are adding. If the new widget is a RowColumn that is being used as a MenuBar (XmNrowColumnType is XmMENU_BAR), the MainWindow automatically uses it for the menu bar.This same check is performed for a Command widget, which is automatically used as the command area. The resources you can use to specify child roles are:

* In Motif 2.0 and later, XmMainWindowSetAreas() is considered deprecated.

```
XmNmenuBar                      XmNcommandWindow
XmNverticalScrollBar            XmNhorizontalScrollBar
XmNworkWindow                   XmNmessageWindow
```

Once one of these values is set, it cannot be reset to `NULL`, although it can be reset to another widget.

The message area is important in most applications, since it is typically the place where brief status and informational messages are displayed. The message area can be implemented using different widgets, such as a read-only Text widget, a read-only ScrolledText object, or a Label widget. Using a Label widget as the message area is quite simple and really doesn't require any explanation.Chapter 18, *Text Widgets*, describes how to use a read-only text area for the message area in a MainWindow.

If you specify the `XmNmessageWindow` resource, the message area is positioned across the bottom of the MainWindow. If you are not satisfied with how the MainWindow handles the layout of the message area, you can make the message area widget a child of the work area manager widget and handle the layout yourself.

Using Resources

Resources specific to the MainWindow and its sub-elements can be useful when configuring the default appearance of your application. If you set these resources in an *app-defaults* file, the specifications can also provide a framework for users to follow when they want to set their own configuration parameters. Even users who are sophisticated enough to figure out how X resource files work still copy existing files and modify them to their own tastes. To assist users, the app-defaults file for an application should be informative and complete, even though it might be lengthy.

Of course, the first step in specifying resources in an app-defaults file is to determine exactly which aspects of the program you want to be configurable. Remember, consistency is the only way to keep from completely confusing a user. Once you have decided which portions of the application are going to be configurable, you can set resource values by specifying complete widget hierarchies. As an example, let's specify some resources for the menu system from *dynapix.c*. The application creates the *File* menu in the following way:

```
XmVaCreateSimplePulldownMenu (menubar, "file_menu", 0, file_cb,
                              XmVaPUSHBUTTON, open, 'O', NULL, NULL,
                              XmVaSEPARATOR,
                              XmVaPUSHBUTTON, quit, 'Q', NULL, NULL,
                              NULL);
```

We can add accelerators to both the *Open* and *Quit* menu items using the following resource specifications:

```
dynapix.main_window.menubar*button_0.accelerator: Ctrl<Key>O
dynapix.main_window.menubar*button_0.acceleratorText: Ctrl+O
```

```
dynapix.main_window.menubar*button_1.accelerator: Ctrl<Key>C
dynapix.main_window.menubar*button_1.acceleratorText: Ctrl+C
```

The result is shown in Figure 4-9.

Figure 4-9 The File menu for dynapix.c with accelerators

These resource settings work because `XmNaccelerator` and `XmNacceleratorText` were not hard-coded by the application. By the same token, the labels of the MenuBar titles and the menu items in the PulldownMenus are hard-coded values that cannot be modified through resources. To relax this restriction, you could try setting the `label` and `mnemonic` parameters to `NULL` in calls to `XmVaCreateSimplePulldownMenu()`. Unfortunately, this technique makes resource specification awfully messy, since the CascadeButtons in the MenuBar and the various PulldownMenus all have names of the form `button_`*n*. The other alternative is to use the more advanced methods of menu creation that are described in Chapter 19, *Menus*.

The MainWindow provides a few other resources that control different visual attributes: they are `XmNmainWindowMarginHeight`, `XmNmainWindowMarginWidth`, and `XmNshowSeparator`. The `XmNshowSeparator` resource controls whether or not Separator widgets are displayed between the different areas of a MainWindow. The margin resources specify the width and height of the MainWindow's margins. Generally, these resources should not be set by the application, but left to the user to specify. For example:

```
*XmMainWindow.showSeparator: True
*XmMainWindow.mainWindowMarginWidth: 10
*XmMainWindow.mainWindowMarginHeight: 10
```

The class name for the MainWindow widget is `XmMainWindow`. If these resource settings were specified in an app-defaults file, they would affect all of the MainWindow widgets in the application. If a user makes these specifications in his *.Xdefaults* file, they would apply to all MainWindow widgets in all applications.

Summary

This chapter introduced you to the concepts involved in creating the main window of an application.To a lesser degree, we showed you how the MainWindow widget can be used

to accomplish some of the necessary tasks. We identified the areas involved in a MainWindow and used some convenience routines to build some adequate prototypes.

The MainWindow can be difficult to understand because of its capabilities as a ScrolledWindow and because it supports the management of so many other objects. The work area of a MainWindow usually contains a manager widget that contains other widgets. Although the MainWindow can handle the layout of its different areas, we do not necessarily encourage you to use all its of its features. For larger, production-style applications, you would probably be better off using the MainWindow for the sake of the MenuBar, while placing the rest of the layout in the hands of a more general-purpose manager widget. These are described in Chapter 8, *Manager Widgets*.

You could also decide not to use the MainWindow widget at all. If done properly, you could probably use one of the manager widget classes described in Chapter 8, *Manager Widgets*, and still be Motif-compliant. Depending on your application, you might find this technique easier to deal with than the MainWindow widget.

Exercises

Based on the material in this chapter, you should be able to do the following exercises:

1. Modify *dynapix.c* to have a new PulldownMenu that controls the background color of the pixmap.
2. Modify *dynapix.c* so that it has a command area. The callback for the Command widget should understand either filenames or color names. If you feel adventurous, try to have it understand both the command `file` and the command `color`. Each command would take a second argument indicating the file or color to use.

In this chapter:

- *The Purpose of Dialogs*
- *The Anatomy of a Dialog*
- *Creating Motif Dialogs*
- *Dialog Resources*
- *Dialog Callback Routines*
- *Piercing the Dialog Abstraction*
- *Dialog Modality*
- *Summary*

5

Introduction to Dialogs

This chapter describes the fundamental concepts that underlie all Motif dialogs. It provides a foundation for the more advanced material in the following chapters. In the course of the introduction, the chapter also provides information about Motif's predefined MessageDialog classes.

In Chapter 4, *The Main Window*, we discussed the top-level windows that are managed by the window manager and that provide the overall framework for an application. Most applications are too complex to do everything in one main top-level window. Situations arise that call for secondary windows, or *transient windows*, that serve specific purposes. These windows are commonly referred to as *dialog boxes*, or more simply as dialogs.

Dialog boxes play an integral role in a GUI-based interface such as Motif.The examples in this book use dialogs in many ways, so just about every chapter can be used to learn more about dialogs. We've already explored some of the basic concepts in Chapter 2, *The Motif Programming Model*, and Chapter 3, *Overview of the Motif Toolkit*. However, the use of dialogs in Motif is quite complex, so we need more detail to proceed further.

The *Motif Style Guide* makes a set of generic recommendations about how all dialogs should look. The *Style Guide* also specifies precisely how certain dialogs should look, how they should respond to user events, and under what circumstances the dialogs should be used. We refer to these dialogs as predefined Motif dialogs, since the Motif toolkit implements each of them for you. These dialogs are completely self-sufficient, opaque objects that require very little interaction from your application. In most situations, you can create the necessary dialog using a single convenience routine and you're done. If you need more functionality than what is provided by a predefined Motif dialog, you may have to create your own customized dialog. In this case, building and handling the dialog requires a completely different approach.

There are three chapters on basic dialog usage in this book - two on the predefined Motif dialogs and one on customized dialogs. There is also an additional chapter later in the book that deals with more advanced dialog topics. This first chapter discusses the most common

class of Motif dialogs, called MessageDialogs. These are the simplest kinds of dialogs; they typically display a short message and use a small set of standard responses, such as *OK*, *Yes*, or *No*. These dialogs are transient, in that they are intended to be used immediately and then dismissed. MessageDialogs define resources and attributes that are shared by most of the other dialogs in the Motif toolkit, so they provide a foundation for us to build upon in the later dialog chapters. Although Motif dialogs are meant to be opaque objects, we will examine their implementation and behavior in order to understand how they really work. This information can help you understand not only what is happening in your application, but also how to create customized dialogs.

Chapter 6, *Selection Dialogs*, describes another set of predefined Motif dialogs, called SelectionDialogs. Since these dialogs are the next step in the evolution of dialogs, most of the material in this chapter is applicable there as well. SelectionDialogs typically provide the user with a list of choices. These dialogs can remain displayed on the screen so that they can be used repeatedly. Chapter 7, *Custom Dialogs*, addresses the issues of creating customized dialogs, and Chapter 27, *Advanced Dialog Programming*, discusses some advanced topics in X and Motif programming using dialogs as a backdrop.

The Purpose of Dialogs

For most applications, it is impossible to develop an interface that provides the full functionality of the application in a single main window. As a result, the interface is typically broken up into discrete functional modules, where the interface for each module is provided in a separate dialog box.

As an example, consider an electronic mail application. The broad range of different functions includes searching for messages according to patterns, composing messages, editing an address book, reporting error messages, and so on. Dialog boxes are used to display simple messages, as shown in Figure 5-1.

Figure 5-1 A Message dialog

They are also used to prompt the user to answer simple questions, as shown in Figure 5-2.

Figure 5-2 A Question dialog

A dialog box can also present a more complicated interaction, as shown in Figure 5-3.

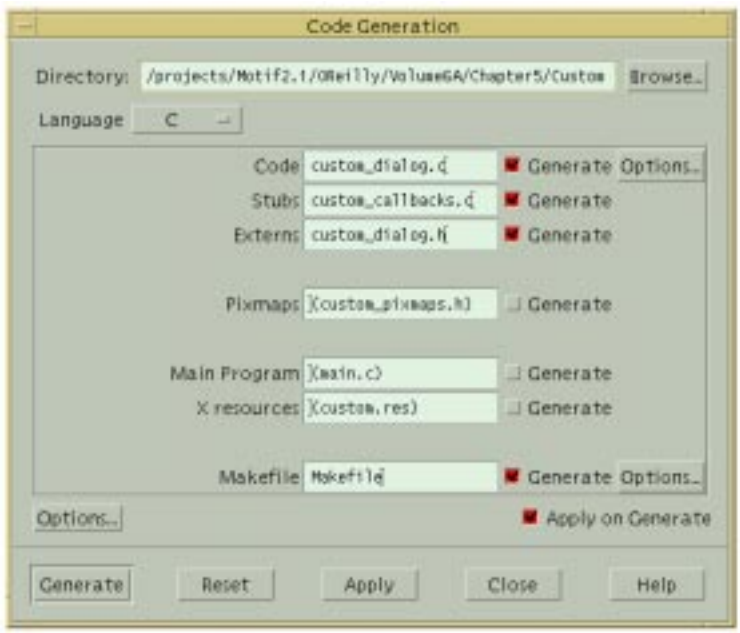


Figure 5-3 A Custom dialog

In Figure 5-3, many different widget classes are used to provide an interface that allows the user to generate code from a popular Motif GUI builder. The purpose of a dialog is to focus on one particular task in an application. Since the scope of these tasks is usually quite limited, an application usually provides them in dialog boxes, rather than in its main window.

There is actually no such thing as a dialog widget class in the Motif toolkit. A dialog is actually made up of a DialogShell widget and a manager widget child that implements the visible part of the dialog. The DialogShell interacts with the window manager to provide

the transient window behavior required of dialogs. When we refer to a dialog widget, we are really talking about the manager widget and all of its children collectively.

When you write a custom dialog, you simply create and manage the children of the DialogShell in the same way that you create and manage the children of a top-level application shell. The predefined Motif dialogs follow the same approach, except that the toolkit creates the manager widget and all of its children internally. Most of the standard Motif dialogs are composed of a DialogShell and either a MessageBox or SelectionBox widget. Each of these widget classes creates and manages a number of internal widgets without application intervention. See Chapter 3, *Overview of the Motif Toolkit*, to review the various types of predefined Motif dialogs.

All of the predefined Motif dialogs are subclassed from the BulletinBoard widget class. As such, a BulletinBoard can be thought of as the generic dialog widget class, although it can certainly be used as generic manager widget (see Chapter 8, *Manager Widgets*). Indeed, a dialog widget is a manager widget, but it is usually not treated as such by the application. The BulletinBoard widget provides the keyboard traversal mechanisms that support gadgets, as well as a number of dialog-specific resources.

It is important to note that for the predefined Motif dialogs, each dialog is implemented as a single widget class, even though there are smaller, primitive widgets under the hood. When you create a MessageBox widget, you automatically get a set of Labels and PushButtons that are laid out as described in the *Motif Style Guide*. What is not created automatically is the DialogShell widget that manages the MessageBox widget. You can either create the shell yourself and place the MessageBox in it or use a Motif convenience routine that creates both the shell and its dialog widget child.

The Motif toolkit uses the DialogShell widget class as the parent for all of the predefined Motif dialogs. In this context, a MessageBox widget combined with a DialogShell widget creates what the Motif toolkit calls a MessageDialog. A careful look at terminology can help you to distinguish between actual widget class and Motif compound objects. The name of the actual widget class ends in `Box`, while the name of the compound object made up of the widget and a DialogShell ends in `Dialog`. For example, the convenience routine `XmCreateMessageBox()` creates a MessageBox widget, which you need to place inside of a DialogShell yourself. Alternatively, `XmCreateMessageDialog()` creates a MessageDialog composed of a MessageBox and a DialogShell.

Another point about terminology involves the commonly-used term dialog box. When we say dialog box, we are referring to a compound object composed of a DialogShell and a dialog widget, not the dialog widget alone. This terminology can be confusing, since the Motif toolkit also provides widget classes that end in `box`.

One subtlety in the use of MessageBox and SelectionBox widgets is that certain types of behavior depend on whether or not the widget is a direct child of a DialogShell. For example, the *Motif Style Guide* says that clicking on the *OK* button in the action area of a

MessageDialog invokes the action of the dialog and then dismisses the dialog. Furthermore, pressing the RETURN key anywhere in the dialog is equivalent to clicking on the *OK* button. However, none of this takes place when the MessageBox widget is not a direct child of a DialogShell.

Perhaps the most important thing to remember is how the Motif toolkit treats dialogs. Once a dialog widget is placed in a DialogShell, the toolkit tends to treat the entire combination as a single entity. In fact, as we move on, you'll find that the toolkit's use of convenience routines, callback functions, and popup widget techniques all hide the fact that the dialog is composed of these discrete elements. While the Motif dialogs are really composed of many primitive widgets, such as PushButtons and TextFields, the single-entity approach implies that you never access the subwidgets directly. If you want to change the label for a button, you set a resource specific to the dialog class, rather than getting a handle to the button widget and changing its resource. Similarly, you always install callbacks on the dialog widget itself, instead of installing them directly on buttons in the control or action areas.

This approach may be confusing for those already familiar with Xt programming, but not yet familiar with the Motif toolkit. Similarly, those who learn Xt programming through experiences with the Motif toolkit might get a misconception of what Xt programming is all about. We try to point out the inconsistencies between the two approaches so that you will understand the boundaries between the Motif toolkit and its Xt foundations.

The Anatomy of a Dialog

As described in Chapter 3, *Overview of the Motif Toolkit*, dialogs are typically broken down into two regions known as the control and action areas. The control area is also referred to as the work area. The control area contains the widgets that provide the functionality of the dialog, such as Labels, ToggleButtons, and List widgets. The action area contains PushButtons whose callback routines actually perform the action of the dialog box. While most dialogs follow this pattern, it is important to realize that these two regions represent user-interface concepts and do not necessarily reflect how Motif dialogs are implemented.

Figure 5-4 shows these areas in a sample dialog box.

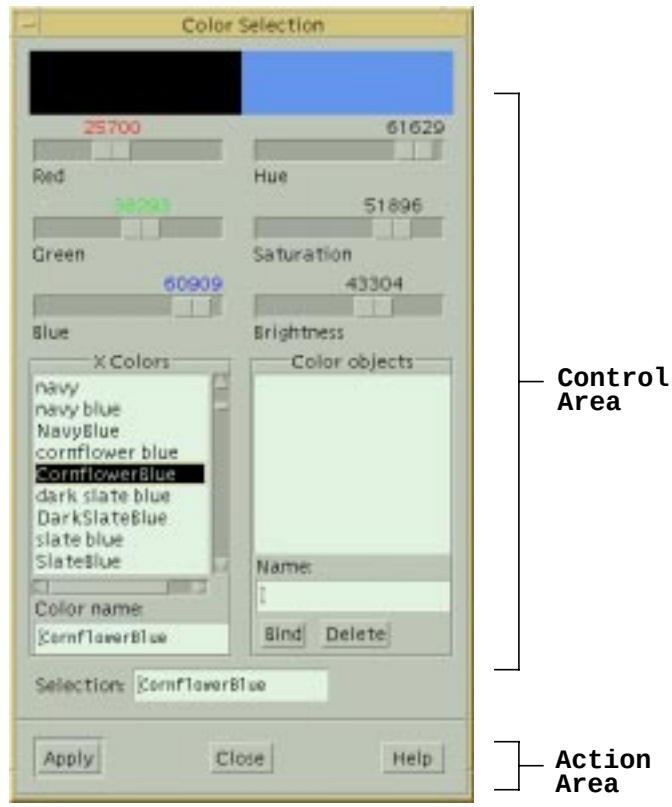


Figure 5-4 A sample dialog

The *Motif Style Guide* describes in a general fashion how the control and action areas for all dialogs should be laid out. For predefined Motif dialogs, the control area is rigidly specified. For customized dialogs, there is only a general set of guidelines to follow. The guidelines for the action area specify a number of common actions that can be used in both predefined Motif dialogs and customized dialogs. These actions have standard meanings that help ensure consistency between different Motif applications.

By default, the predefined Motif MessageDialogs provide three action buttons, which are normally labelled *OK*, *Cancel*, and *Help*, respectively. SelectionDialogs provide a fourth button, normally labelled *Apply*, which is placed between the *OK* and *Cancel* buttons. This button is created but not managed, so it is not visible unless the application explicitly manages it. The *Style Guide* specifies that the *OK* button applies the action of the dialog and dismisses it, while the *Apply* button applies the action but does not dismiss the dialog. The *Cancel* button dismisses the dialog without performing any action other than resetting

the dialog to the initial state, and the *Help* button provides any help that is available for the dialog*. When you are creating custom dialogs, or even when you are using the predefined Motif dialogs, you may need to provide actions other than the default ones. If so, you should change the labels on the buttons so that the actions are obvious. You should try to use the common actions defined by the *Motif Style Guide* if they are appropriate, since these actions have standard meanings. We will address this issue further as it comes up in discussion; it is not usually a problem until you create your own customized dialogs, as described in Chapter 7, *Custom Dialogs*.

Creating Motif Dialogs

Under most circumstances, creating a predefined Motif dialog box is very simple. All Motif dialog types have corresponding convenience routines that simplify the task of creating and managing them. For example, a standard MessageDialog can be created as shown in the following code fragment:

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

extern Widget    parent;
Widget           dialog;
Arg              arg[5];
XmString         xms;
int              n = 0;

xms = XmStringCreateLocalized ("Hello World");
XtSetArg (arg[n], XmNmessageString, xms); n++;
dialog = XmCreateMessageDialog (parent, "message", arg, n);
XmStringFree (xms);
```

The convenience routine does almost everything automatically.The only thing that we have to do is specify the message that we want to display.

Dialog Header Files

As we mentioned earlier, there are two basic types of predefined Motif dialog boxes: MessageDialogs and SelectionDialogs. MessageDialogs present a simple message, to which a yes (*OK*) or no (*Cancel*) response usually suffices. There are six types of MessageDialogs: ErrorDialog, InformationDialog, QuestionDialog, TemplateDialog, WarningDialog, and WorkingDialog. These types are not actually separate widget classes, but rather instances of the generic MessageDialog that are configured to display different graphic symbols. All of the MessageDialogs are compound objects that are composed of a MessageBox widget and a DialogShell. When using MessageDialogs, you must include the file *<Xm/MessageB.h>*.

* A more complete list of the pre-defined actions is given in Chapter 7, *Custom Dialogs*. Note that Figure 5-4 has a button labelled *Close* rather than *Cancel*, because the dialog in this instance is popped down without any reset.

SelectionDialogs allow for more complicated interactions. The user can select an item from a list or type an entry into a TextField widget before acting on the dialog. There are essentially four types of SelectionDialogs, although the situation is a bit more complex than for MessageDialogs. The PromptDialog is a specially configured SelectionDialog; both of these dialogs are compound objects that are composed of a SelectionBox widget and a DialogShell. The Command widget and the FileSelectionDialog are based on separate widget classes. However, they are both subclassed from the SelectionBox and share many of its features. When we use the general term "selection dialogs", we are referring to these three widget classes plus their associated dialog shells. To use a SelectionDialog, you must include the file *<Xm/SelectioB.h>*.* For FileSelectionDialogs, the appropriate include file is *<Xm/FileSB.h>*, and for the Command widget it is *<Xm/Command.h>*.

Creating a Dialog

You can use any of the following convenience routines to create a dialog box.They are listed according to the header file in which they are declared:

```
<Xm/MessageB.h>:
Widget XmCreateMessageBox(Widget, char *, ArgList, Cardinal)
Widget XmCreateMessageDialog(Widget, char *, ArgList, Cardinal)
Widget XmCreateErrorDialog(Widget, char *, ArgList, Cardinal)
Widget XmCreateInformationDialog(Widget, char *, ArgList, Cardinal)
Widget XmCreateQuestionDialog(Widget, char *, ArgList, Cardinal)
Widget XmCreateTemplateDialog(Widget, char *, ArgList, Cardinal)
Widget XmCreateWarningDialog(Widget, char *, ArgList, Cardinal)
Widget XmCreateWorkingDialog(Widget, char *, ArgList, Cardinal)
<Xm/SelectioB.h>:
Widget XmCreateSelectionBox(Widget, char *, ArgList, Cardinal)
Widget XmCreateSelectionDialog(Widget, char *, ArgList, Cardinal)
Widget XmCreatePromptDialog(Widget, char *, ArgList, Cardinal)
<Xm/FileSB.h>:
Widget XmCreateFileSelectionBox(Widget, char *, ArgList, Cardinal)
Widget XmCreateFileSelectionDialog(Widget, char *, ArgList, Cardinal)
<Xm/Command.h>:
Widget XmCreateCommand(Widget, char *, ArgList, Cardinal)
```

Each of these routines creates a dialog widget. In addition, the routines that end in `Dialog` automatically create a DialogShell as the parent of the dialog widget. All of the convenience functions for creating dialogs use the standard Motif creation routine format. For example, `XmCreateMessageDialog()` takes the following form:

```
Widget XmCreateMessageDialog (Widget parent, char *name,
                              ArgList arglist; Cardinal argcount)
```

In this case, we are creating a common MessageDialog, which is a MessageBox with a DialogShell parent.The *`parent`* parameter specifies the widget that acts as the owner or

* Yes, you read that right. It does, in fact, read *SelectioB.h*.The reason for the missing *n* is there is a fourteen-character filename limit on UNIX System V machines.

parent of the DialogShell. Note that the parent must not be a gadget, since the parent must have a window associated with it. The dialog widget itself is a child of the DialogShell. You are returned a handle to the newly created dialog widget, not the DialogShell parent. For the routines that just create a dialog widget, the *parent* parameter is simply a manager widget that contains the dialog.

The *arglist* and *argcount* parameters for the convenience routines specify resources using the old-style ArgList format, just like the rest of the Motif convenience routines. A varargs-style interface is not available for creating dialogs. However, you can use the varargs-style interface for setting resources on a dialog after is has been created by using XtVaSetValues().

Setting Resources

There are a number of resources and callback functions that apply to almost all of the Motif dialogs. These resources deal with the action area buttons in the dialogs. Other resources only apply to specific types of dialogs; they deal with the different control area components such as Labels, TextFields, and List widgets. The different resources are listed below, grouped according to the type of dialogs that they affect:

General dialog resources:

```
XmNokLabelString                XmNokCallback
XmNcancelLabelString            XmNcancelCallback
XmNhelpLabelString              XmNhelpCallback
```

MessageDialog resources:

```
XmNmessageString                XmNsymbolPixmap
```

SelectionDialog resources:

```
XmNapplyLabelString             XmNapplyCallback
XmNselectionLabelString         XmNlistLabelString
```

FileSelectionDialog resources:

```
XmNfilterLabelString            XmNdirListLabelString
XmNfileListLabelString
```

Command resources:

```
XmNpromptString
```

The labels and callbacks of the various buttons in the action area are specified by resources based on the standard Motif dialog button names. For example, the XmNokLabelString resource is used to set the label for the *OK* button. XmNokCallback is used to specify the callback routine that the dialog should call when that button is activated. As discussed earlier, it may be appropriate to change the labels of these buttons, but the resource and callback names will always have names that correspond to their default labels.

The XmNmessageString resource specifies the message that is displayed by the MessageDialog. The XmNsymbolPixmap resource specifies the iconic symbol that is associated with each of the MessageDialog types. This resource is rarely changed, so discussion of it is deferred until Chapter 27, *Advanced Dialog Programming*.

The other resources apply to the different types of selection dialogs. For example, XmNselectionLabelString sets the label that is placed above the list area in a SelectionDialog. These resources are discussed in Chapter 6, *Selection Dialogs*.

All of these resources apply to the Labels and PushButtons in the different dialogs. It is important to note that they are different from the usual resources for Labels and PushButtons. For example, the Label resource XmNlabelString would normally be used to specify the label for both Label and PushButton widgets. Dialogs use their own resources to maintain the abstraction of the dialog widget as a discrete user-interface object.

Another important thing to remember about the resources that refer to widget labels is that their values must be specified as compound strings. Compound strings allow labels to be rendered in arbitrary fonts and to span multiple lines. See Chapter 25, *Compound Strings*, for more information.

The following code fragment demonstrates how to specify dialog resources and callback routines:

```
Widget         dialog;
XmString       msg, yes, no;
extern void    my_callback(Widget, XtPointer, XtPointer);

dialog = XmCreateQuestionDialog (parent, "dialog", NULL, 0);
yes = XmStringCreateLocalized ("Yes");
no = XmStringCreateLocalized ("No");
msg = XmStringCreateLocalized ("Do you want to quit?");
XtVaSetValues (dialog, XmNmessageString, msg, XmNokLabelString, yes,
                       XmNcancelLabelString, no, NULL);
XtAddCallback (dialog, XmNokCallback, my_callback, NULL);
XtAddCallback (dialog, XmNcancelCallback, my_callback, NULL);

XmStringFree (yes);
XmStringFree (no);
XmStringFree (msg);
```

Dialog Management

None of the Motif toolkit convenience functions manage the widgets that they create, so the application must call XtManageChild() explicitly. It just so happens that managing a dialog widget that is the immediate child of a DialogShell causes the entire dialog to pop up. Similarly, unmanaging the same dialog widget causes it and its DialogShell parent to pop down. This behavior is consistent with the Motif toolkit's treatment of the dialog/shell combination as a single object abstraction. The toolkit is treating its own dialog widgets as

opaque objects and trying to hide the fact that there are Dialog Shells associated with them. The toolkit is also making the assumption that when the programmer manages a dialog, she wants it to pop up immediately.

This practice is somewhat confusing to experienced programmers of Xt, who are used to calling the routines XtPopup() and XtPopdown() to display and hide a dialog. You should note that managing or unmanaging *any* immediate child of a Motif DialogShell will cause the whole dialog to appear or disappear respectively: this behavior is not restricted to just the built-in Motif dialog objects*.

For reference, XtPopup() and XtPopdown() take the following forms:

```
void XtPopup (Widget shell, XtGrabKind grab_kind)
void XtPopdown (Widget shell)
```

The *shell* parameter to the function must be a shell widget; in this case it happens to be a DialogShell. If you created the dialog using one of the Motif convenience routines, you can get a handle to the DialogShell by calling XtParent() on the dialog widget.

The *grab_kind* parameter can be one of XtGrabNone, XtGrabNonexclusive, or XtGrabExclusive. We almost always use XtGrabNone, since the other values imply a *server grab*, which means that other windows on the desktop are locked out. Grabbing the server results in what is called *modality*; it implies that the user cannot interact with anything but the current dialog. While a grab may be desirable in some cases, the Motif toolkit provides some predefined resources that handle the grab for you automatically. The advantage of using this alternate method is that it allows the client to communicate more closely with the Motif Window Manager (*mwm*) and it provides for different kinds of modality. These methods are discussed later in this chapter. For detailed information on XtPopup() and the different uses of *grab_kind*, see Volume 4, *X Toolkit Intrinsics Programming Manual*.

Let's take a closer look at how dialogs are really used in an application. Examining the overall design and the mechanics that are involved will help to clarify a number of issues about managing and unmanaging dialogs and DialogShells. The program listed in Example 5-1 displays an InformationDialog when the user presses a PushButton in the application's main window.†

Example 5-1 The hello_dialog.c program

```
* hello_dialog.c -- your typical Hello World program using
** an InformationDialog.
*/

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

* To be more precise, the ChangeManaged() method of the DialogShell calls XtPopup() and XtPopdown() internally, provided that the XmNmappedWhenManaged resource of the DialogShell is true (the default).

† XtVaAppInitialize() is considered deprecated in X11R6.

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

main (int argc, char *argv[])
{
    XtAppContext     app;
    Widget           toplevel, rc, hpb, gpb;
    /* callbacks for the pushbuttons -- pops up dialog */
    void             popup_callback(Widget, XtPointer, XtPointer);
    void             exit_callback(Widget, XtPointer, XtPointer);

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

    rc = XmCreateRowColumn (toplevel, "rowcol", NULL, 0);

    hpb = XmCreatePushButton (rc, "Hello", NULL, 0);
    XtAddCallback (hpb,
           XmNactivateCallback,
           popup_callback,
           (XtPointer) "Hello World");

    gpb = XmCreatePushButton (rc, "Goodbye", NULL, 0);
    XtAddCallback (gpb, XmNactivateCallback, exit_callback, NULL);

    XtManageChild (hpb);
    XtManageChild (gpb);
    XtManageChild (rc);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

/* callback for the “Hello” PushButton.
** Popup an InformationDialog displaying the text passed as the client
** data parameter.
*/
void popup_callback (Widget button, XtPointer client_data,
                     XtPointer call_data)
{
    Widget     dialog;
    XmString   xm_string;
    void       activate_callback(Widget, XtPointer, XtPointer);
    Arg        args[5];
    int        n = 0;
    char       *text = (char *) client_data;

    /* set the label for the dialog */
    xm_string = XmStringCreateLocalized (text);
    XtSetArg (args[n], XmNmessageString, xm_string); n++;
    /* Create the InformationDialog as child of button */
    dialog = XmCreateInformationDialog (button, "info", args, n);
    /* no longer need the compound string, free it */
    XmStringFree (xm_string);
```

```
    /* add the callback routine */
    XtAddCallback (dialog, XmNokCallback, activate_callback, NULL);
    /* manage the MessageBox: has the side effect of displaying the */
    /*       XmDialogShell parent */
    XtManageChild (dialog);
}

/*
** callback routine when the user pressed the "Goodbye" button
*/
void exit_callback (Widget w, XtPointer client_data, XtPointer call_data){
    exit (0);
}

/* callback routine for when the user presses the OK button.
** Yes, despite the fact that the OK button was pressed, the
** widget passed to this callback routine is the dialog!
*/
void activate_callback (Widget dialog,
                        XtPointer client_data,
                        XtPointer call_data)
{
    puts ("OK was pressed.");
}
```

The output of this program is shown in Figure 5-5.

Figure 5-5 Output of the hello_dialog program

Dialogs are often invoked from callback routines attached to PushButtons or other interactive widgets. Once the dialog is created and popped up, control of the program is returned to the main event-handling loop (`XtAppMainLoop()`),where normal event processing resumes. At this point, if the user interacts with the dialog by selecting a control or activating one of the action buttons, a callback routine for the dialog is invoked. In Example 5-1, we happen to use a MessageDialog, but the type of dialog used is irrelevant to the model.

When the PushButton in the main window is pressed, `popup_callback()` is called. A text string that is used as the message to display in the InformationDialog is passed as client

data. The dialog uses a single callback routine, `activate_callback()`, for the `XmNokCallback` resource. This function is invoked when the user presses the *OK* button. The callback simply prints a message to standard output that the button has been pressed. Similar callback routines could be installed for the *Cancel* and *Help* buttons through the `XmNcancelCallback` and `XmNhelpCallback` resources.

Closing Dialogs

You might notice that activating either the *OK* or the *Cancel* button in the previous example causes the dialog to be automatically popped down. The *Motif Style Guide* says that when any button in the action area of a predefined Motif dialog is pressed, except for the *Help* button, the dialog should be dismissed. The Motif toolkit takes this specification at face value and enforces the behavior, which is consistent with the idea that Motif dialogs are self-contained, self-sufficient objects. They manage everything about themselves from their displays to their interactions with the user. And when it's time to go away, they unmanage themselves. Your application does not have to do anything to cause any of the behavior to occur.

Unfortunately, this behavior does not take into account error conditions or other exceptional events that may not necessarily justify the dialog's dismissal. For example, if pressing *OK* causes a file to be updated, but the operation fails, you may not want the dialog to be dismissed. If the dialog is still displayed, the user can try again without having to repeat the actions that led to popping up the dialog.

The `XmNautoUnmanage` resource provides a way around the situation. This resource controls whether the dialog box is automatically unmanaged when the user selects an action area button other than the *Help* button. If `XmNautoUnmanage` is `True`, after the callback routine for the button is invoked, the DialogShell is popped down and the dialog widget is unmanaged automatically. However, if the resource is set to `False`, the dialog is not automatically unmanaged. The value of this resource defaults to `True` for MessageDialogs and SelectionDialogs; it defaults to `False` for FileSelectionDialogs.

Since it is not always appropriate for a dialog box to unmanage itself automatically, it turns out to be easier to set `XmNautoUnmanage to False` in most circumstances. This technique makes dialog management easier, since it keeps the toolkit from indiscriminately dismissing a dialog simply because an action button has been activated. While it is true that we could program around this situation by calling `XtPopup()` or `XtManageChild()` from a callback routine in error conditions, this type of activity is confusing because of the double-negative action it implies. In other words, programming around the situation is just undoing something that should not have been done in the first place.

This discussion brings up some issues about when a dialog should be unmanaged and when it should be destroyed. If you expect the user to have an abundant supply of computer memory, you may reuse a dialog by retaining a handle to the dialog, as shown in

Example 5-4 later in this chapter. There are also performance considerations that may affect whether you choose to destroy or reuse dialogs. It takes less time to reuse a dialog than it does to create a new one, provided that your application is not so large that it is consuming all of the system's resources. If you do not retain a handle to a dialog, and if you need to conserve memory and other resources, you should destroy the dialog whenever you pop it down.

Another method the user might use to close a dialog is to select the *Close* item from the window menu. This menu can be pulled down from the title bar of a window. Since the menu belongs to the window manager, rather than the shell widget or the application, you cannot install any callback routines for its menu items. However, you can use the `XmNdeleteResponse` resource to control how the DialogShell responds to a *Close* action*. It can have one of the following values:

`XmUNMAP`
: This value causes the dialog to be unmapped. The dialog disappears from the screen, but it is not destroyed, nor is it iconified. The dialog widget and its windows are still intact and can be redisplayed using `XtPopup()`. This value is the default for DialogShells.

`XmDESTROY`
: This value destroys the DialogShell and calls its `XmNdestroyCallback`. Note that all of the shell's children are also destroyed, including the dialog widget and its subwidgets. When the dialog is destroyed, you cannot redisplay the dialog or reference its handle again. If you need the dialog again, you have to create another one. Examples of using the `XmNdestroyCallback` are presented in Chapter 27, *Advanced Dialog Programming*.

`XmDO_NOTHING`
: This value causes the toolkit to take no action. The value should only be specified in circumstances where you want to handle the event on your own. However, handling the event involves much more than installing a simple callback routine. It requires building a lower-level mechanism that interprets the proper events when they are sent by the window manager. The most common thing to do in such cases is to activate the default action of the dialog or to interpose a prompting mechanism to verify the user's action. This topic is discussed in Chapter 20, *Interacting With the Window Manager*.

It may be convenient for your application to know when a dialog has been popped up or down. If so, you can install callbacks that are invoked whenever either of these events take place. The actions of popping up and down dialogs can be monitored through the `XmNpopupCallback` and `XmNpopdownCallback` callback routines. For example, when the function associated with a `XmNpopupCallback` is invoked, you could position the

* The Motif VendorShell, from which the DialogShell is subclassed, is responsible for trapping the notification and determining what to do next, based on the value of the resource.

dialog automatically, rather than allowing the window manager to control the placement. See Chapter 7, *Custom Dialogs*, for more information on these callbacks.

Generalizing Dialog Creation

Posting dialogs that display informative messages is something just about every application is going to do frequently. Rather than write a separate routine for each case where a message needs to be displayed, we can generalize the process by writing a single routine that handles most, if not all, cases. Example 5-2 shows the `PostDialog()` routine. This routine creates a MessageDialog of a given type and displays an arbitrary message. Rather than use the convenience functions provided by Motif for each of the MessageDialog types, the routine uses the generic function `XmCreateMessageDialog()` and configures the symbol to be displayed by setting the `XmNdialogType` resource.

Example 5-2 The PostDialog() routine

```
/*
** PostDialog() -- a generalized routine that allows the programmer
** to specify a dialog type (message, information, error, help, etc.),
** and the message to display.
*/
Widget PostDialog (Widget parent, int dialog_type, char *msg)
{
    Widget     dialog;
    XmString   text;

    dialog = XmCreateMessageDialog (parent, "dialog", NULL, 0);
    text = XmStringCreateLocalized (msg);
    XtVaSetValues (dialog,
                   XmNdialogType, dialog_type,
                   XmNmessageString, text,
                   NULL);
XmStringFree (text);
XtManageChild (dialog);
return dialog;
}
```

This routine allows the programmer to specify several parameters: the parent widget, the type of dialog that is to be used, and the message that is to be displayed. The function returns the new dialog widget, so that the calling routine can modify it, unmanage it, or keep a handle to it. You may have additional requirements that this simplified example does not satisfy. For instance, the routine does not allow you to specify callback functions for the buttons in the action area and it does not handle the destruction of the widget when it is no longer needed. You could extend the routine to handle these issues, or you could control them outside the context of the function. You may also want to extend the routine so that it reuses the same dialog each time it is called and so that it allows you to disable the different action area buttons. All of these issues are discussed again in Chapter 6, *Selection Dialogs*, and in Chapter 27, *Advanced Dialog Programming*.

Dialog Resources

The following sections discuss resources that are specific to Motif dialogs. In most cases, these resources are BulletinBoard widget resources, since all Motif dialogs are subclassed from this class. However, they are not intended to be used by generic BulletinBoard widgets. The resources only apply when the widget is an immediate child of a DialogShell widget; they are really intended to be used exclusively by the predefined Motif dialog classes. Remember that the resources must be set on the dialog widget, not the DialogShell. See Chapter 8, *Manager Widgets*, for details on the generic BulletinBoard resources.

The Default Button

All predefined Motif dialogs have a *default button* in their action area. The default button is activated when the user presses the RETURN key in the dialog. The *OK* button is normally the default button, but once the dialog is displayed, the user can change the default button by using the arrow keys to traverse the action buttons. The action button with the keyboard focus is always the default button. Since the default button can be changed by the user, the button that is the default is only important when the dialog is initially popped up. The importance of the default button lies in its ability to influence the user's default response to the dialog.

You can change the default button for a MessageDialog by setting the `XmNdefaultButtonType` resource on the dialog widget. This resource is specific to MessageDialogs; it cannot be set for the various types of selection dialogs. The resource can have one of the following values:

`XmDIALOG_OK_BUTTON`
: This value specifies that the default button is the furthest button on the left of the dialog. By default, this button is the *OK* button, although its label may have been changed to another string.

`XmDIALOG_CANCEL_BUTTON`
: This value specifies that the *Cancel* button is the default button. This value is appropriate in situations where the action of the dialog is destructive, such as for a WarningDialog that is posted in order to warn the user of a possibly dangerous action.

`XmDIALOG_HELP_BUTTON`
: This value specifies the *Help* button, which is always the furthest button on the right of a Motif dialog. This button is rarely set as the default button.

`XmDIALOG_NONE`
: This value specifies that there is no default button.

The values for `XmNdefaultButtonType` come up again later, when we discuss `XmMessageBoxGetChild()` and again in Chapter 6, *Selection Dialogs*, when we

consider the routine XmSelectionBoxGetChild()*. An example of how the default button type can be used is shown in Example 5-3.

Example 5-3 The WarningMsg() function

```
/* WarningMsg() -- Inform the user that she is about to embark on a
** dangerous mission and give her the opportunity to back out.
*/
void WarningMsg (Widget parent, XtPointer client_data, XtPointer call_data)
{
static Widget  dialog;
XmString       text, ok_str, cancel_str;
char           *msg = (char *) client_data;

if (!dialog)
    dialog = XmCreateWarningDialog (parent, "warning", NULL, 0);
text = XmStringCreateLocalized (msg);
ok_str = XmStringCreateLocalized ("Yes");
cancel_str = XmStringCreateLocalized ("No");
XtVaSetValues (dialog,  XmNmessageString, text,
               XmNokLabelString,     ok_str,
               XmNcancelLabelString, cancel_str,
               XmNdefaultButtonType, XmDIALOG_CANCEL_BUTTON,
               NULL);
XmStringFree (text);
XmStringFree (ok_str);
XmStringFree (cancel_str);
XtManageChild (dialog);
}
```

The intent of this function is to create a dialog that tries to discourage the user from performing a destructive action. By using a WarningDialog and by making the *Cancel* button the default choice, we have given the user adequate warning that the action may have dangerous consequences.The output of a program running this code fragment is shown in Figure 5-6.

Figure 5-6 An instance of the WarningMsg() routine

You can also set the default button for a dialog by the setting the BulletinBoard resource XmNdefaultButton. This technique works for both MessageDialogs and

* Strictly speaking, the *GetChild() routines are deprecated in Motif 2.0 and later. You should prefer the routine XtNameToWidget().

SelectionDialogs. The resource value must be a widget ID, which means that you have to get a handle to a subwidget in the dialog to set the resource. You can get the handle to subwidgets using `XmMessageBoxGetChild()` or `XmSelectionBoxGetChild()`*. Since this method breaks the Motif dialog abstraction, we describe it later in the *Internal Widgets* Section.

Initial Keyboard Focus

When a dialog widget is popped up, one of the internal widgets in the dialog has the keyboard focus. This widget is typically the default button for the dialog, which makes sense in most cases. However, there are situations where it is appropriate for another widget to have the initial keyboard focus. For example, when a PromptDialog is popped up, it makes sense for the TextField to have the keyboard focus so that the user can immediately start typing a response.

The `XmNinitialFocus` resource can be used to handle this situation. Since this resource is a Manager widget resource, it can be used for both MessageDialogs and SelectionDialogs, although it is normally only used for SelectionDialogs. The resource specifies the subwidget that has the keyboard focus the first time that the dialog is popped up. If the dialog is popped down and popped up again later, it remembers the widget that had the keyboard focus when it was popped down and that widget is given the keyboard focus again. The resource value must again be a widget ID. The default value of `XmNinitialFocus` for MessageDialogs is the subwidget that is also the `XmNdefaultButton` for the dialog. For SelectionDialogs, the text entry area is the default value for the resource.

Button Sizes

The `XmNminimizeButtons` resource controls how the dialog sets the widths of the action area buttons. If the resource is set to `True`, the width of each button is set so that it is as small as possible while still enclosing the entire label, which means that each button will have a different width. The default value of `False` specifies that the width of each button is set to the width of the widest button, so that all buttons have the same width.

The Dialog Title

When a new shell widget is mapped to the screen, the window manager creates its own window that contains the title bar, resize handles, and other window decorations and makes the window of the DialogShell the child of this new window. This technique is called reparenting a window; it is only done by the window manager in order to add window decorations to a shell window. The window manager reparents instances of all of the shell

* `XtNameToWidget`() should be used in preference in Motif 2.0 and later.

widget classes except OverrideShell. These shells are used for menus and thus should not have window manager decorations.

Most window managers that reparent shell windows display titles in the title bars of their windows. For predefined Motif dialogs, the Motif toolkit sets the default title to the name of the dialog widget with the string _popup appended. Since this string is almost certainly not an appropriate title for the window, you can change the title explicitly using the XmNdialogTitle BulletinBoard resource. (Do not confuse this title with the message displayed in MessageDialog, which is set by XmNmessageString.) The value for XmNdialogTitle must be a compound string. The BulletinBoard in turn sets the XmNtitle resource of the DialogShell; the value of this resource is a regular C string.

So, you can set the title for a dialog window in one of two ways. The following code fragment shows how to set the title using the XmNdialogTitle resource:

```
XmString title_string = XmStringCreateLocalized ("Dialog Box");
Widget dialog = XmCreateMessageDialog (parent, "dialog_name", NULL, 0);

XtVaSetValues (dialog, XmNdialogTitle, title_string, NULL);
XmStringFree (title_string);
```

This technique requires creating a compound string. If you set the XmNtitle resource directly on the DialogShell, you can use a regular C string, as in the following code fragment:

```
dialog = XmCreateMessageDialog (parent, "dialog_name", NULL, 0);
XtVaSetValues (XtParent (dialog), XmNtitle, "Dialog Box", NULL);
```

While the latter method is easier and does not require creating and freeing a compound string, it does break the abstraction of treating the dialog as a single entity.

Dialog Resizing

The XmNnoResize resource controls whether or not the window manager allows the dialog to be resized. If the resource is set to True, the window manager does not display resize handles in the window manager frame for the dialog. The default value of False specifies that the window manager should provide resize handles. Since some dialogs cannot handle resize events very well, you may find it better aesthetically to prevent the user from resizing them.

This resource is an attribute of the BulletinBoard widget, even though it only affects the shell widget parent of a dialog widget. The resource is provided as a convenience to the programmer, so that she is not required to get a handle to the DialogShell. The resource only affects the presence of resize handles in the window manager frame; it does not deal with other window manager controls. See Chapter 20, *Interacting With the Window Manager*, for details on how to specify the window manager controls for a DialogShell, or any shell widget, directly.

Dialog Render Tables*

The BulletinBoard widget provides resources that enable you to specify the render tables that are used for all of the Button, Label, and Text widget descendants of the BulletinBoard. Since Motif dialog widgets are subclassed from the BulletinBoard, you can use these resources to make sure that fonts and other appearance resources that are used within a dialog are consistent. The `XmNbuttonRenderTable`† resource specifies the render table that is used for all of the button descendants of the dialog. The resource is set on the dialog widget itself, not on its individual children. Similarly, the `XmNlabelRenderTable`‡ resource is used to set the render table for all of the Label descendants of the dialog and `XmNtextRenderTable`§ is used for all of the Text and TextField descendants.

If one of these resources is not set, the toolkit determines the render table by searching up the widget hierarchy for an ancestor that holds the `XmQTspecifyRenderTable` trait. BulletinBoard, VendorShell, MenuShell, and derived widget classes hold this trait. If an ancestor is found, the render table resource is set to the value of that render table resource in the ancestor widget. See Chapter 24, for more information on render tables.

You can override the `XmNbuttonRenderTable`, `XmNlabelRenderTable`, and `XmNtextRenderTable` resources on a per-widget basis by setting the `XmNrenderTable` resource directly on individual widgets. Of course, you must break the dialog abstraction and retrieve the widgets internal to the dialog itself to set this resource. While we describe how to retrieve the widgets later in this chapter, we do not necessarily recommend configuring dialogs down to this level of detail.

Dialog Callback Routines

As mentioned earlier, the predefined Motif dialogs have their own resources to reference the labels and callback routines for the action area PushButtons. Instead of accessing the PushButton widgets in the action area to install callbacks, you use the resources `XmNokCallback`, `XmNcancelCallback`, and `XmNhelpCallback` on the dialog widget itself. These callbacks correspond to each of the three buttons, *OK*, *Cancel*, and *Help*.

Installing callbacks for a dialog is no different than installing them for any other type of Motif widget; it may just seem different because the dialog widgets contain so many subwidgets. The following code fragment demonstrates the installation of simple callback for all of the buttons in a MessageDialog:

---

* As of Motif 2.0, the `XmFontList` is obsolete, and is replaced by the `XmRenderTable` type. For backwards compatibility, the `XmFontList` is implemented through an `XmRenderTable`

† `XmNbuttonFontList` is deprecated, and is replaced by `XmNbuttonRenderTable`

‡ `XmNlabelFontList` is deprecated, and is replaced by `XmNlabelRenderTable`

§ `XmNtextFontList` is deprecated, and is replaced by `XmNtextRenderTable`

```
...
dialog = XmCreateMessageDialog (w, "notice", NULL, 0);
...
XtAddCallback (dialog, XmNokCallback, ok_pushed, (XtPointer) "Hi");
XtAddCallback (dialog, XmNcancelCallback, cancel_pushed, (XtPointer) "Bye");
XtAddCallback (dialog, XmNhelpCallback, help_pushed, NULL);
XtManageChild (dialog);
...

/* ok_pushed() --the OK button was selected. */
void ok_pushed (Widget widget, XtPointer client_data, XtPointer call_data)
{
    char *message = (char *) client_data;
    printf ("OK was selected: %s\n", message);
}

/* cancel_pushed() --the Cancel button was selected. */
void cancel_pushed (Widget widget, XtPointer client_data, XtPointer call_data)
{
    char *message = (char *) client_data;
    printf ("Cancel was selected: %s\n", message);
}

/* help_pushed() --the Help button was selected. */
void help_pushed (Widget widget, XtPointer client_data, XtPointer call_data)
{
    printf ("Help was selected\n");
}
```

In this example, a dialog is created and callback routines for each of the three responses are added using `XtAddCallback()`. We also provide simple client data to demonstrate how the data is passed to the callback routines. These callback routines simply print the fact that they have been activated; the messages they print are taken from the client data.

All of the dialog callback routines take three parameters, just like any standard callback routine. The `widget` parameter is the dialog widget that contains the button that was selected; it is not the DialogShell widget or the PushButton that the user selected from the action area. The second parameter is the `client_data`, which is supplied to `XtAddCallback()`, and the third is the `call_data`, which is provided by the internals of the widget that invoked the callback.

The `client_data` parameter is of type `XtPointer`, which means that you can pass arbitrary values to the function, depending on what is necessary. However, you cannot pass a `float` or a double value or an actual data structure. If you need to pass such values, you must pass the address of the variable or a pointer to the data structure. In keeping with the philosophy of abstracting and generalizing code, you should use the `client_data` parameter as much as possible because it eliminates the need for some global variables and it keeps the structure of an application modular.

For the predefined Motif dialogs, the call_data parameter is a pointer to a data structure that is filled in by the dialog box when the callback is invoked. The data structure contains a callback reason and the event that invoked the callback. The structure is of type XmAnyCallbackStruct, which is declared as follows:

```
typedef struct {
    int      reason;
    XEvent   *event;
} XmAnyCallbackStruct;
```

The value of the reason field is an integer value that can be any one of XmCR_HELP, XmCR_OK, or XmCR_CANCEL. The value specifies the button that the user pressed in the dialog box. The values for the reason field remain the same, no matter how you change the button labels for a dialog. For example, you can change the label for the *OK* button to say *Help*, using the resource XmNokLabelString, but the reason parameter will still be XmCR_OK when the button is activated.

Because the reason field provides information about the user's response to the dialog in terms of the button that was pushed, we can simplify the previous code fragment and use one callback function for all of the possible actions. The callback function can determine which button was selected by examining reason. Example 5-4 demonstrates this simplification.*

Example 5-4 The reason.c program

```
/* reason.c -- examine the reason field of the callback structure
** passed as the call_data of the callback function. This field
** indicates which action area button in the dialog was pressed.
*/
#include <Xm/RowColumn.h>
#include <Xm/MessageB.h>
#include <Xm/PushB.h>

/* main() --create a pushbutton whose callback pops up a dialog box */
main (int argc, char *argv[])
{
    XtAppContext   app;
    Widget         toplevel, rc, pb1, pb2;
    void           pushed(Widget, XtPointer, XtPointer);

    XtSetLanguageProc (NULL, NULL, NULL);
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv, NULL,
                                    sessionShellWidgetClass, NULL);
    rc = XmCreateRowColumn (toplevel, "rowcol", NULL, 0);
    pb1 = XmCreatePushButton (rc, "Hello", NULL, 0);
    XtAddCallback (pb1, XmNactivateCallback, pushed,
                   (XtPointer) "Hello World");
    pb2 = XmCreatePushButton (rc, "Goodbye", NULL, 0);
```

* XtVaAppInitialize() is considered deprecated in X11R6.

```
    XtAddCallback (pb2, XmNactivateCallback, pushed,
                   (XtPointer) "Goodbye World");
    XtManageChild (pb1);
    XtManageChild (pb2);
    XtManageChild (rc);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

/* pushed() --the callback routine for the main app's pushbuttons.
** Create and popup a dialog box that has callback functions for
** the OK, Cancel and Help buttons.
*/
void pushed (Widget widget, XtPointer client_data, XtPointer call_data)
{
    static Widget  dialog;
    char           *message = (char *) client_data;
    XmString       t = XmStringCreateLocalized (message);

    /* See if we've already created this dialog -- if so,
    ** we don't need to create it again. Just set the message
    ** and manage it (pop it up).
    */
    if (!dialog) {
        void callback(Widget, XtPointer, XtPointer);
        Arg args[5];
        int n = 0;
        XtSetArg (args[n], XmNautoUnmanage, False); n++;
        dialog = XmCreateMessageDialog (widget, "notice", args, n);
        XtAddCallback (dialog, XmNokCallback, callback,
                       (XtPointer) "Hi");
        XtAddCallback (dialog, XmNcancelCallback, callback,
                       (XtPointer) "Foo");
        XtAddCallback (dialog, XmNhelpCallback, callback,
                       (XtPointer) "Bar");
    }
    XtVaSetValues (dialog, XmNmessageString, t, NULL);
    XmStringFree (t);
    /* Managing child of DialogShell pops up the dialog */
    XtManageChild (dialog);
}

/* callback() --One of the dialog buttons was selected.
** Determine which one by examining the "reason" parameter.
*/
void callback (Widget widget, XtPointer client_data, XtPointer call_data)
{
    char       *button;
    char       *message = (char *) client_data;
    XmAnyCallbackStruct   *cbs = (XmAnyCallbackStruct *) call_data;

    switch (cbs->reason) {
        case XmCR_OK     : button = "OK";     break;
        case XmCR_CANCEL : button = "Cancel"; break;
```

```
            case XmCR_HELP   : button = "Help";   break;
        }
        printf ("%s was selected: %s\n", button, message);

        if (cbs->reason != XmCR_HELP) {
            /* the ok and cancel buttons "close" the widget:.
            ** Unmanaging child of DialogShell pops down the dialog.
            */
            XtUnmanageChild (widget);
        }
    }
```

Another interesting change in this application is the way `pushed()` determines if the dialog has already been created. By making the dialog widget handle `static` to the `pushed()` callback function, we retain a handle to this object across multiple button presses. For each invocation of the callback, the dialog's message is reset and it is popped up again.

Considering style guide issues again, it is important to know when it is appropriate to dismiss a dialog. As noted earlier, the toolkit automatically unmanages a dialog whenever any of the action area buttons are activated, except for the *Help* button. This behavior is controlled by `XmNautoUnmanage`, which defaults to `True`. However, if you set this resource to False, the callback routines for the buttons in the action area have to control the behavior on their own. In Example 5-4, the callback routine pops down the dialog when the reason is XmCR_OK or `XmCR_CANCEL`, but not when it is `XmCR_HELP`.

Piercing the Dialog Abstraction

As described earlier, Motif treats dialogs as if they are single user-interface objects. However, there are times when you need to break this abstraction and work with some of the individual widgets that make up a dialog. This section describes how the dialog convenience routines work, how to work directly with the DialogShell, and how to access the widgets that are internal to dialogs.

Convenience Routines

The fact that Motif dialogs are self-sufficient does not imply that they are black boxes that perform magic that you cannot perform yourself. For example, the convenience routines for the MessageDialog types follow these basic steps:

1. Create a popup shell widget of type `xmDialogShellWidgetClass` using `XtCreatePopupShell()`.
2. Create a widget of type `xmMessageBoxWidgetClass` as the child of the DialogShell.
3. Set the `XmNdialogType` resource for the dialog.

4. Install a callback routine for the XmNdestroyCallback resource of the MessageBox, so that it automatically destroys its DialogShell parent.

The XmNdialogType resource can be set to one of the following values:

```
XmDIALOG_ERROR          XmDIALOG_INFORMATION      XmDIALOG_MESSAGE
XmDIALOG_QUESTION       XmDIALOG_TEMPLATE         XmDIALOG_WARNING
XmDIALOG_WORKING
```

The type of the dialog does not affect the kind of widget that is created. The only thing the type affects is the graphical symbol that is displayed in the control area of the dialog. The convenience routines set the resource based on the routine that is called (e.g. XmCreateErrorDialog() sets the resource to XmDIALOG_ERROR). The widget automatically sets the graphical symbol based on the dialog type. You can change the type of a dialog after it is created using XtVaSetValues(); modifying the type also changes the dialog symbol that is displayed.

The Motif dialog convenience routines create DialogShells internally to support the single-object dialog abstraction. With these routines, the toolkit is responsible for the DialogShell, so the dialog widget uses its XmNdestroyCallback to destroy its parent upon its own destruction. If the dialog is unmapped or unmanaged, so is its DialogShell parent. The convenience routines do not add any resources or call any functions to support the special relationship between the dialog widget and the DialogShell, since most of the code that handles the interaction is written into the internals of the BulletinBoard.

The DialogShell

As your programs become more complex, you may eventually have to access the DialogShell parent of a dialog widget in order to get certain things done. This section examines DialogShells as independent widgets and describes how they are different from other shell widgets. There are three main features of a DialogShell that differentiate it from a SessionShell and a TopLevelShell*.

- A DialogShell cannot be iconified by the user or by the application.
- When the parent of a DialogShell is iconified, withdrawn, unmapped, or destroyed, the DialogShell children of that window are withdrawn or destroyed.
- A DialogShell is always placed on top of the shell widget that owns the parent of the DialogShell.†

The DialogShell is subclassed from the TransientShell and VendorShell classes. A shell that is subclassed from TransientShell cannot be iconified independently of its parent. However, if the parent of a DialogShell is iconified or unmapped, the DialogShell is

* The ApplicationShell is considered deprecated in X11R6.

† This is at the whim of the window manager. For *mwm*, this is true. See Chapter 20 for more details.

unmapped as well. If the parent is destroyed, so is the DialogShell and the dialog within it. Remember, the parent of the DialogShell is another widget somewhere in the application, such as a Label, a PushButton, a SessionShell, or even another DialogShell. For example, if the callback for PushButton creates a dialog, the PushButton might be designated as the owner of the dialog. If the shell that contains the PushButton is iconified, the dialog is also withdrawn from the screen. If the PushButton's shell or the PushButton itself is destroyed, the dialog is destroyed as well.

The parent-child relationship between a DialogShell and its parent is different from the classic case, where the parent actually contains the child within its geometrical bounds. The DialogShell widget is a popup child of its parent, which means that the usual geometry-management relationship does not apply. Nonetheless, the parent widget must be managed in order for the child to be displayed. If a widget has popup children, those children are not mapped to the screen if the parent is not managed, which means that you must never make a menu item the parent of a DialogShell.

Assuming that the parent is displayed, the window manager attempts to place the DialogShell based on the value of the `XmNdefaultPosition` BulletinBoard resource. The default value of this resource is `True`, which means that the window manager positions the DialogShell so that it is centred on top of its parent. If the resource is set to `False`, the application and the window manager negotiate about where the dialog is placed. This resource is only relevant when the BulletinBoard is the immediate child of a DialogShell, which is always the case for Motif dialogs. If you want, you can position the dialog by setting the `XmNx` and `XmNy` resources for the dialog widget. Positioning the dialog on the screen must be done through a `XmNmapCallback` routine, which is called whenever the application calls `XtManageChild()`. See Chapter 7, *Custom Dialogs*, for a discussion about dialog positioning.

The Motif Window Manager imposes an additional constraint on the stacking order of the DialogShell and its parent. *mwm* always forces the DialogShell to be directly on top of its parent in the stacking order. The result is that the shell that contains the widget acting as the parent of the DialogShell cannot be placed on top of the dialog. This behavior is defined by the *Motif Style Guide* and is enforced by the Motif Window Manager and the Motif toolkit. Many end-users have been known to report the behavior as an application-design bug, so you may want to describe this behavior explicitly in the documentation for your application, in order to prepare the user ahead of time.*

Internally, DialogShell widgets communicate frequently with dialog widgets in order to support the single-entity abstraction promoted by the Motif toolkit. However, you may find that you need to access the DialogShell part of a Motif dialog in order to query information

* Other window managers behave differently. See Chapter 20 for more details about window manager interaction.

from the shell or to perform certain actions on it. The include file <*Xm/DialogS.h*> provides a convenient macro for identifying whether or not a particular widget is a DialogShell:

```
#define XmIsDialogShell(w)    XtIsSubclass(w, xmDialogShellWidgetClass)
```

If you need to use this macro, or you want to create a DialogShell using XmCreateDialogShell(), you need to include <*Xm/DialogS.h*>.

The macro is useful if you want to determine whether or not a dialog widget is the direct child of a DialogShell. For example, earlier in this chapter, we mentioned that the *Motif Style Guide* suggests that if the user activates the *OK* button in a MessageDialog, the entire dialog should be popped down. If you have created a MessageDialog without using `XmCreateMessageDialog()` and you want to be sure that the same thing happens when the user presses the *OK* button in that dialog, you need to test whether or not the parent is a DialogShell before you pop down the dialog. The following code fragment shows the use of the macro in this type of situation:

```
/* traverse up widget tree until we find a dialog shell parent */
Widget GetDialogShellChild (Widget widget)
{
    Widget parent;
    while (widget) {
        if (((parent = XtParent (widget)) != (Widget) 0)
            if (XmIsDialogShell (parent))
                return widget;
        widget = parent;
    }
    return (Widget) 0;
}

/* traverse up the tree to find any shell ancestor */
Widget GetShell (Widget w)
{
    while (widget && !XtIsShell (widget))
        widget = XtParent (widget);
    return widget;
}

void ok_callback (Widget w, XtPointer client_data, XtPointer call_data)
{
    Widget top;
    /* do whatever the callback needs to do */
    ...
    /* if immediate parent is not a DialogShell, mimic the same
    ** behavior as if it were.
    */

    /* Motif DialogShell */
    if ((top = GetDialogShellChild (w)) != (Widget) 0)
         XtUnmanageChild (top);
    /* Probably a topLevelShellWidgetClass */
    if ((top = GetShell (w)) != (Widget)))
```

```
        XtPopdown (top);
}
```

The Motif toolkit defines similar macros for all of its widget classes. For example, the header file <*Xm/MessageB.h*> defines the macro `XmIsMessageBox()`:

```
#define XmIsMessageBox(w)     XtIsSubclass (w, xmMessageBoxWidgetClass)
```

This macro determines whether or not a particular widget is subclassed from the MessageBox widget class. Since all of the MessageDialogs are really instances of the MessageBox class, the macro covers all of the different types of MessageDialogs. If the widget is a MessageBox, the macro returns True whether or not the widget is an immediate child of a DialogShell. Note that this macro does not return `True` if the widget is a DialogShell.

Internal Widgets

All of the Motif dialog widgets are composed of primitive subwidgets such as Labels, PushButtons, and TextField widgets. For most tasks, it is possible to treat a dialog as a single entity. However, there are some situations when it is useful to be able to get a handle to the widgets internal to the dialog. For example, one way to set the default button for a dialog is to use the `XmNdefaultButton` resource. The value that you specify for this resource must be a widget ID, so this is one of those times when it is necessary to get a handle to the actual subwidgets contained within a dialog.

You can retrieve a subwidget of any component using `XtNameToWidget()`*, which has the following form:

```
Widget XtNameToWidget (Widget widget, char *child)
```

The *`widget`* parameter is a handle to a dialog widget, not its DialogShell parent. The *`child`* parameter is the name associated with a descendant of *`widget`*. The children of a MessageBox have the following constant names:

Symbol	Message	Separator
OK	Cancel	Help

For example, the Cancel button in a MessageDialog can be accessed as follows:

```
Widget cancel_b = XtNameToWidget (message_box, "Cancel");
```

One method that you can use to customize the predefined Motif dialogs is to unmanage the subwidgets that are inappropriate for your purposes. To get the widget ID for a widget, so that you can pass it to `XtUnmanageChild()`, you need to call `XtNameToWidget()`. You

* The Motif convenience routines, `XmMessageBoxGetChild`(), `XmSelectionBoxGetChild`(), and so forth, are considered deprecated in Motif 2.0. `XmFileSelectionBoxGetChild`() has not been maintained in particular, and Motif 2 components of the FileSelectionBox cannot be accessed using this convenience function.

can also use this routine to get a handle to a widget that you want to temporarily disable. These techniques are demonstrated in the following code fragment:

```
text = XmStringCreateLocalized ("You have new mail.");
XtSetArg (args[0], XmNmessageString, text);
dialog = XmCreateInformationDialog (parent, "message", args, 1);
XmStringFree (text);
XtSetSensitive (XtNameToWidget (dialog, “Help”), False);
XtUnmanageChild (XtNameToWidget (dialog, “Cancel”));
```

The output of a program using this code fragment is shown in Figure 5-7.

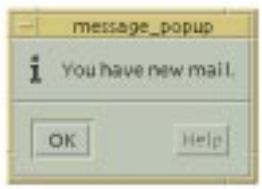


Figure 5-7 MessageDialog with unmanaged Cancel, and insensitive Help buttons

Since the message in this dialog is so simple, it does not make sense to have both an *OK* and a *Cancel* button, so we unmanage the latter. On the other hand, it does make sense to have a *Help* button. However, there is currently no help available, so we make the button unselectable by desensitizing it using `XtSetSensitive()`.

Dialog Modality

The concept of forcing the user to respond to a dialog is known as *modality*. Modality governs whether or not the user can interact with other windows on the desktop while a particular dialog is active. Dialogs are either modal or modeless. There are three levels of modality: primary application modal, full application modal, and system modal. In all cases, the user must interact with a modal dialog before control is released and normal input is resumed. In a system modal dialog, the user is prevented from interacting with any other window on the display. Full application modal dialogs allow the user to interact with any window on the desktop except those that are part of the same application as the modal window. Primary application modal dialogs allow the user to interact with any other window on the display except for the window that is acting as the parent for this particular dialog.

For example, if the user selected an action that caused an error dialog to be displayed, the dialog could be primary application modal, so that the user would have to acknowledge the error before she interacts with the same window again. This type of modality does not restrict her ability to interact with another window in the same application, provided that the other window is not the one acting as the parent for the modal dialog.

Modal dialogs are perhaps the most frequently misused feature of a graphical user interface. Programmers who fail to grasp the concept of event-driven programming and design, whereby the user is in control, often fall into the convenient escape route that modal dialogs provide. This problem is difficult to detect, let alone cure, because there are just as many right ways to invoke modal dialogs as there are wrong ways. Modality should be used in moderation, but it should also be used consistently. Let's examine a common scenario. Note that this example does not necessarily favor using modal dialogs; it is presented as a reference point for the types of things that people are used to doing in tty-based programs.

A text editor has a function that allows the user to save its text to a file. In order to save the text, the program needs a filename. Once it has a filename, the program needs to check that the user has sufficient permission to open or create the file and it also needs to see if there is already some text in the file. If an error condition occurs, the program needs to notify the user of the error, ask for a new filename, or get permission to overwrite the file's contents. Whatever the case, some interaction with the user is necessary in order to proceed. If this were a typical terminal-based application, the program flow would be similar to that in the following code fragment:

```
FILE *fp;
char buf [BUFSIZ], file [BUFSIZ];
extern char *index();
printf ("What file would you like to use? ");

if (!(fgets (file, sizeof (file), stdin)) || file[0] == 0) {
    puts ("Cancelled."); return;
}
/* get rid of newline terminator */
*(index (file, '\n')) = 0;

/* "a+" creates file if it doesn't exist */
if (!(fp = fopen (file, "a+"))) {
    perror (file); return;
}

if (ftell (fp) > 0) {
    /* There's junk in the file already */
    printf ("Overwrite contents of %s? ", file);
    buf[0] = 0;
    if (!(fgets (buf, sizeof (buf), stdin)) || buf[0] == 0 || buf[0] == 'n' ||
             buf[0] == 'N') {
        puts ("Cancelled.");
        fclose (fp);
        return;
    }
}
rewind (fp);
```

This style of program flow is still possible with a graphical user interface system using modal dialogs. In fact, the style is frequently used by engineers who are trying to port tty-

based applications to Motif. It is also a logical approach to programming, since it does one task followed by another, asking only for information that it needs when it needs it.

However, in an event-driven environment, where the user can interact with many different parts of the program simultaneously, displaying a series of modal dialogs is not the best way to handle input and frequently it's just plain wrong as a design approach. You must adopt a new paradigm in interface design that conforms to the capabilities of the window system and meets the expectations of the user. It is essential that you understand the event-driven model if you want to create well-written, easy-to-use applications.

Window-based applications should be modelled on the behavior of a person filling out a form, such as an employment application or a medical questionnaire. Under this scenario, you are given a form asking various questions. You take it to your seat and fill it out however you choose. If it asks for your license number, you can get out your driver's license and copy down the number. If it asks for your checking account number, you can examine your check book for that information. The order in which you fill out the application is entirely up to you. You are free to examine the entire form and fill out whatever portions you like, in whatever order you like.

When the form is complete, you return it to the person who gave it to you. The attendant can check it over to see if you forgot something. If there are errors, you typically take it back and continue until it's right. The attendant can simply ask you the question straight out and write down whatever you say, but this prevents him from doing other work or dealing with other people. Furthermore, if you don't know the answer to the question right away, then you have to take the form back and fill it out the way you were doing it before. No matter how you look at it, this process is not an interview where you are asked questions in sequence and must answer them that way. You are supposed to prepare the form off-line, without requiring interaction from anyone else.

Window-based applications should be treated no differently. Each window, or dialog, can be considered to be a form of some sort. Allow the user to fill out the form at her own convenience and however she chooses. If she wants to interact with other parts of the application or other programs on the desktop, she should be allowed to do so. When the user selects one of the buttons in the action area, this action is her way of returning the form. At this time, you may either accept it or reject it. At no point in the process so far have we needed a modal dialog.

Once the form has been submitted, you can take whatever action is appropriate. If there are errors in any section of the dialog, you may need to notify the user of the error. Here is where a modal dialog can be used legitimately. For example, if the user is using a FileSelectionDialog to specify the file she wants to read and the file is unreadable, then you must notify her so that she can make another selection. In this case, the notification is usually in the form of an ErrorDialog, with a message that explains the error and an *OK* button. The user can read the message and press the button to acknowledge the error.

It is often difficult to judge what types of questions or how much information is appropriate in modal dialogs. The rule of thumb is that questions in modal dialogs should be limited to simple, yes/no questions. You should not prompt for any information that is already available through an existing dialog, but instead bring up that dialog and instruct the user to provide the necessary information there. You should also avoid posting modal dialogs that prompt for a filename or anything else that requires typing. You should be requesting this type of information through the text fields of modeless dialog boxes.

As for the issue of forcing the user to fill out forms in a particular order, it may be perfectly reasonable to require this type of interaction. You should implement these restrictions by managing and unmanaging separate dialogs, rather than by using modal dialogs to prevent interaction with all but a single dialog.

All of these admonitions are not to suggest that modal dialogs are rare or that you should avoid using them at all costs. On the contrary, they are extremely useful in certain situations, are quite common, and are used in a wide variety of ways--even those that we might not recommend. We have presented all of these warnings because modal dialogs are frequently misused and programs that use fewer of them are usually better than those that use more of them. Modal dialogs interrupt the user and disrupt the flow of work in an application. There is no sanity checking to prevent you from misusing dialogs so it is up to you to keep the use of modal dialogs to a minimum.

Implementing Modal Dialogs

Once you have determined that you need to implement a modal dialog, you can use the `XmNdialogStyle` resource to set the modality of the dialog. This resource is defined by the BulletinBoard widget class; it is only relevant when the widget is an immediate child of a DialogShell. The resource can be set to one of the following values:

```
XmDIALOG_MODELESS
XmDIALOG_PRIMARY_APPLICATION_MODAL
XmDIALOG_FULL_APPLICATION_MODAL
XmDIALOG_SYSTEM_MODAL
```

`XmDIALOG_MODELESS` is the default value for the resource, so unless you change the value any dialog that you create will be modeless.

When you use one of the modal values, the user has no choice but to respond to your dialog box before continuing to interact with the application. If you use modality at all, you should probably avoid using `XmDIALOG_SYSTEM_MODAL`, since it is rarely necessary to restrict the user from interacting with all of the other applications on the desktop. This style of modality is typically reserved for system-level interactions. Under the Motif Window Manager, when a system modal dialog is popped up, if the user moves the mouse outside of the modal dialog, the cursor turns into the international “do not enter” symbol. Attempts to interact with other windows cause the server to beep.

Example 5-5 shows a sample program that displays a dialog box that the user must reply to before continuing to interact with the application.*

Example 5-5 The modal.c program

```
/* modal.c -- demonstrate modal dialogs. Display two pushbuttons
** each activating a modal dialog.
*/
#include <Xm/RowColumn.h>
#include <Xm/MessageB.h>
#include <Xm/PushB.h>

/* main() --create a pushbutton whose callback pops up a dialog box */
main (int argc, char *argv[])
{
    XtAppContext     app;
    Widget           toplevel, button, rowcolumn;
    void             pushed(Widget, XtPointer, XtPointer);

    XtSetLanguageProc (NULL, NULL, NULL);
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv,
                                    NULL,sessionShellWidgetClass, NULL);
    rowcolumn = XmCreateRowColumn (toplevel, "rowcolumn", NULL, 0);
    button = XmCreatePushButton (rowcolumn, "Application Modal", NULL, 0);
    XtAddCallback (button, XmNactivateCallback, pushed,
                           (XtPointer) XmDIALOG_FULL_APPLICATION_MODAL);
    XtManageChild (button);
    button = XmCreatePushButton (rowcolumn, "System Modal", NULL, 0);
    XtAddCallback (button, XmNactivateCallback, pushed,
                           (XtPointer) XmDIALOG_SYSTEM_MODAL);
    XtManageChild (button);
    XtManageChild (rowcolumn);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

/* pushed() --the callback routine for the main app's pushbutton.
** Create either a full-application or system modal dialog box.
*/
void pushed (Widget widget, XtPointer client_data, XtPointer call_data)
{
    static Widget  dialog;
    XmString       t;
    void           dlg_callback(Widget, XtPointer, XtPointer);
    unsigned char  modality = (unsigned char) client_data;

    /* See if we've already created this dialog -- if so,
    ** we don't need to create it again. Just re-pop it up.
    */
    if (!dialog) {
        Arg args[5];
```

* XtVaAppInitialize() is considered deprecated in X11R6.

```
            int n = 0;
            XmString ok = XmStringCreateLocalized ("OK");
            XtSetArg(args[n], XmNautoUnmanage, False); n++;
            XtSetArg(args[n], XmNcancelLabelString, ok); n++;
            dialog = XmCreateInformationDialog (widget, "notice", args, n);
            XtAddCallback (dialog, XmNcancelCallback, dlg_callback, NULL);
            XtUnmanageChild (XtNameToWidget (dialog, "OK"));
            XtUnmanageChild (XtNameToWidget (dialog, "Help"));
        }
        t = XmStringCreateLocalized ("You must reply to this message now!");
        XtVaSetValues (dialog, XmNmessageString, t, XmNdialogStyle, modality,
                       NULL);
        XmStringFree (t);
        XtManageChild (dialog);
    }

    void dlg_callback (Widget dialog, XtPointer client_data, XtPointer call_data)
    {
        XtUnmanageChild (dialog);
    }
```

The output of this program is shown in Figure 5-8.

Figure 5-8 Output of modal.c

This program demonstrates both application modal and system modal dialogs.The value for the `XmNdialogType` resource is passed as client data to the callback routine that posts the dialog.

Forcing an Immediate Response

In Example 5-5, once the dialog is posted, the function returns so that `XtAppMainLoop()` can continue to process the events. If the function does not return, the application will not respond to user events and, for that matter, the dialog will not even be displayed. Just because a dialog is realized and managed does not mean that it is displayed on the screen, as events must be processed in order for it to appear. See Chapter 27, *Advanced Dialog Programming*, for a discussion of this phenomenon. (See Volume 1, *Xlib Programming Manual*, for more information on event processing.)

However, there are situations where it would be nice not to have to return from the function and break its flow of control. As an example, consider a function that allows the user to perform a particularly dangerous action, such as removing or overwriting a file. What you'd like to do is prompt the user first and allow her to reconsider the action before proceeding. If she confirms the action, you'd like to continue from within the same function without having to return in order to process events.

In order to write this type of function, we need to find a way to process the events that display and manage the dialog without returning to the main loop. The user also needs to be able to respond to the dialog, so we really need to allow normal event processing to continue in the context of the function. Let's assume that there is a hypothetical function, `AskUser()`, that we can use in the following way:

```
if (AskUser ("Are you sure you want to do this?") == YES) {
    /* proceed with action... */
}
```

The function `AskUser()` should post a full application modal MessageDialog, wait for the user to respond to the dialog, and return a predefined value for either `YES` or `NO`. The magic of the function is to get around the requirement that events can only be read and processed directly from `XtAppMainLoop()`. The code for such a function is shown in Example 5-6.

Example 5-6 The AskUser() routine

```
#define YES 1
#define NO  2
/*
** AskUser() -- a generalized routine that asks the user a question
** and returns the Yes/No response.
*/
int AskUser (Widget parent, char *question)
{
    static Widget       dialog;
    XmString            text, yes, no;
    static int          answer;
    void                response(Widget, XtPointer, XtPointer);
    extern XtAppContext app;

    if (!dialog) {
        dialog = XmCreateQuestionDialog (parent, "dialog", NULL, 0);
        yes = XmStringCreateLocalized ("Yes");
        no = XmStringCreateLocalized ("No");
        XtVaSetValues (dialog,
                       XmNdialogStyle, XmDIALOG_FULL_APPLICATION_MODAL,
                       XmNokLabelString, yes,
                       XmNcancelLabelString, no,
                       NULL);
        XtSetSensitive (XtNameToWidget (dialog, "Help"), False);
        XtAddCallback (dialog, XmNokCallback, response,
                       (XtPointer) &answer);
        XtAddCallback (dialog, XmNcancelCallback, response,
```

```
                        (XtPointer) &answer);
            XmStringFree (yes);
            XmStringFree (no);
        }
        answer = 0;
        text = XmStringCreateLocalized (question);
        XtVaSetValues (dialog, XmNmessageString, text, NULL);
        XmStringFree (text);
        XtManageChild (dialog);
        /* while the user hasn't provided an answer, simulate main loop.
        ** The answer changes as soon as the user selects one of the
        ** buttons and the callback routine changes its value.
        */
        while (answer == 0)
            XtAppProcessEvent (app, XtIMAll);
        XtUnmanageChild (dialog);
        return answer;
    }

    /* response() --The user made some sort of response to the
    ** question posed in AskUser(). Set the answer (client_data)
    ** accordingly and destroy the dialog.
    */
    void response (Widget widget, XtPointer client_data, XtPointer call_data)
    {
        int *answer = (int *) client_data;
        XmAnyCallbackStruct *cbs = (XmAnyCallbackStruct *) call_data;

        switch (cbs->reason) {
            case XmCR_OK     : *answer = YES;   break;
            case XmCR_CANCEL : *answer = NO;    break;
            default          : return;
        }
    }
```

The first parameter to the function is the widget that acts as the parent of the new dialog. It is important to choose this widget wisely. The parent widget must not be a gadget or an unrealized widget; it should be a widget that is currently mapped to the screen. Widgets that are menu items are not good candidates, since they are not mapped to the screen for very long. The top-level shell widget of the widget that caused the callback function to be invoked is typically a good choice. The second parameter is the string that is displayed in the dialog.

The routine is intended to be used to display a dialog that asks a Yes/No question, so we change the *OK* and *Cancel* labels to say *Yes* and *No*, respectively. The routine creates a QuestionDialog as a static `Widget`, which allows us to reuse the dialog, rather than create it each time the function is called. This technique may improve performance on some machines. The modality of the dialog and the labels for the PushButtons in the action area are set at creation time, but the actual message string is set each time that the function is called, since the message can change. When we install the callback routines for the buttons, we use the address of the `answer` variable as the client data. As a result, when the user

responds to the question by selecting the Yes or *No* button, the callback routine has access to the variable and can change its value accordingly.

The `while` loop is where the application waits for the user to make a selection. The loop exits when the variable `answer is` changed from its initial value (0) to either `YES` (1) or NO (2) by the callback routine. By using `XtAppProcessEvent()`, we have effectively reproduced the `XtAppMainLoop()` function that is used in the main application. Rather than returning to that level and breaking our flow of control, we have introduced a miniature main loop in the function itself.

While the `AskUser()` routine in Example 5-6 is useful as it is written, there are a number of enhancements that will make it even more useful. By using what we've learned in this chapter, we can come up with a simple, yet extremely robust interface for prompting the user for responses to questions without breaking the natural flow of control in the application. Example 5-7 demonstrates a generalized version of `AskUser()` in a complete application. The program *ask_user.c* allows the user to execute UNIX commands that create and remove a temporary file.*

Example 5-7 The ask_user.c program

```
/* ask_user.c -- the user is presented with two pushbuttons.
** The first creates a file (/tmp/foo) and the second removes it.
** In each case, a dialog pops up asking for verification of the action.
**
** This program is intended to demonstrate an advanced implementation
** of the AskUser() function. This time, the function is passed the
** strings to use for the OK button and the Cancel button as well as
** the button to use as the default value.
**/
#include <Xm/DialogS.h>
#include <Xm/SelectioB.h>
#include <Xm/RowColumn.h>
#include <Xm/MessageB.h>
#include <Xm/PushB.h>

#define YES 1
#define NO  2
/* Generalize the question/answer process by creating a data structure
** that has the necessary labels, questions and everything needed to
** execute a command.
*/
typedef struct {
    char *label;     /* label for pushbutton used to invoke cmd */
    char *question;  /* question for dialog box to confirm cmd */
    char *yes;       /* what the "OK" button says */
    char *no;        /* what the "Cancel" button says */
    int   dflt;      /* which should be the default answer */
    char *cmd;       /* actual command to execute (using system()) */
```

* `XtVaAppInitialize`() is considered deprecated in X11R6.

```
} QandA;

QandA touch_foo = {"Create", "Create /tmp/foo?", "Yes", "No",
                   YES, "touch /tmp/foo"};
QandA rm_foo = {"Remove", "Remove /tmp/foo?", "Yes", "No",
                   NO, "rm /tmp/foo"};
XtAppContext app;

main (int argc, char *argv[])
{
    Widget     toplevel, button, rowcolumn;
    XmString   label;
    Arg        args[2];
    void       pushed(Widget, XtPointer, XtPointer);

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

    rowcolumn = XmCreateRowColumn (toplevel, "rowcolumn", NULL, 0);
    label = XmStringCreateLocalized (touch_foo.label);
    XtSetArg(args[0], XmNlabelString, label);
    button = XmCreatePushButton (rowcolumn, "button", args, 1);
    XtAddCallback (button,
    XmNactivateCallback, pushed, (XtPointer) &touch_foo);
    XtManageChild (button);
    XmStringFree (label);
    label = XmStringCreateLocalized (rm_foo.label);
    XtSetArg (args[0], XmNlabelString, label);
    button = XmCreatePushButton (rowcolumn, "button", args, 1);
    XtAddCallback (button, XmNactivateCallback, pushed,
                   (XtPointer) &rm_foo);
    XtManageChild (button);
    XmStringFree (label);
    XtManageChild (rowcolumn);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

/* pushed() --when a button is pressed, ask the question described
** by the QandA parameter (client_data). Execute the cmd if YES.
*/
void pushed (Widget widget, XtPointer client_data, XtPointer call_data)
{
    QandA *quest = (QandA *) client_data;
    if (AskUser (widget, quest->question,
                 quest->yes, quest->no, quest->dflt) == YES) {
        printf ("Executing: %s\n", quest->cmd);
        system (quest->cmd);
    } else
        printf ("Not executing: %s\n", quest->cmd);
}

/* AskUser() -- a generalized routine that asks the user a question
```

```
** and returns a response. Parameters are: the question, the labels
** for the "Yes" and "No" buttons, and the default selection to use.
*/
AskUser (Widget parent, char *question, char *ans1, char *ans2, int default_ans)
{
    static Widget    dialog = NULL; /* static to avoid multiple creation */
    XmString         text, yes, no;
    static int       answer;
    void             response(Widget, XtPointer, XtPointer);

    if (!dialog) {
        dialog = XmCreateQuestionDialog (parent, "dialog", NULL, 0);
        XtVaSetValues (dialog, XmNdialogStyle,
                       XmDIALOG_FULL_APPLICATION_MODAL, NULL);
        XtSetSensitive (XtNameToWidget (dialog, "Help"), False);
        XtAddCallback (dialog, XmNokCallback, response,
                       (XtPointer) &answer);
        XtAddCallback (dialog, XmNcancelCallback, response,
                       (XtPointer) &answer);
    }
answer = 0;
text = XmStringCreateLocalized (question);
yes = XmStringCreateLocalized (ans1);
no = XmStringCreateLocalized (ans2);
XtVaSetValues (dialog,
              XmNmessageString, text,
              XmNokLabelString, yes,
              XmNcancelLabelString, no,
              XmNdefaultButtonType,
             (default_ans == YES?
                   XmDIALOG_OK_BUTTON:
                   XmDIALOG_CANCEL_BUTTON),
              NULL);
    XmStringFree (text);
    XmStringFree (yes);
    XmStringFree (no);
    XtManageChild (dialog);
    while (answer == 0)
        XtAppProcessEvent (app, XtIMAll);
    XtUnmanageChild (dialog);
    /* make sure the dialog goes away before returning. Sync with server
    ** and update the display.
    */
    XSync (XtDisplay (dialog), 0);
    XmUpdateDisplay (parent);
    return answer;
}

/* response() --The user made some sort of response to the
** question posed in AskUser(). Set the answer (client_data)
** accordingly.
*/
void response (Widget widget, XtPointer client_data, XtPointer call_data)
{
```

```
        int *answer = (int *) client_data;
        XmAnyCallbackStruct *cbs = (XmAnyCallbackStruct *) call_data;

        if (cbs->reason == XmCR_OK)
            *answer = YES;
        else if (cbs->reason == XmCR_CANCEL)
            *answer = NO;
    }
```

The new version of `AskUser()` is more dynamic than before, since more of the dialog is configurable upon each invocation of the function. The routine now allows you to specify the message, the labels for the *OK* and *Cancel* buttons, and the default button for the dialog. The flexibility of the routine is achieved at the cost of a few more lines of source code and additional parameters to the function. The performance of the function is completely unaffected.

One case that the new version of `AskUser()` does not deal with is the need for additional buttons in the action area of the dialog. For example, what if you need to provide a *Cancel* button in addition to the *Yes* and *No* answers? Let's say that the user has selected the *Quit* menu item in a text editor application. Since the user has yet to update the changes to the file that she has been editing, the application posts a dialog that asks her if she wants to update her changes before exiting. There are three possible responses:

- Yes, update the changes and exit (*Yes*).
- No, don't update the changes, but exit anyway (*No*).
- Don't update the changes and don't exit the application (*Cancel*).

One easy way to provide these three choices is to set the label for the *Help* button to *Cancel* using the `XmNhelpLabelString` resource. Then you just need to modify the callback function so that it handles the `XmCR_HELP` reason and returns a new value for the *Cancel* button.

However, this solution does not work if you want to provide help in addition to these choices. The default MessageDialog only provides three buttons in the action area, although you can add additional action area buttons to the dialog. For more information on how to handle this situation, see Chapter 7, *Custom Dialogs*.

Summary

Dialogs are used extensively in all window-oriented applications and their uses are quite diverse. As a result, it is impossible to provide numerous examples of the use of any one particular style of dialog. This chapter introduced the implementation of Motif dialogs by using the predefined MessageDialogs as examples. We described how to create the dialogs, how to set various dialog resources, how to handle dialog callback routines, and how to

implement modal dialogs. Although our examples used MessageDialogs, much of the discussion is applicable to other types of Motif dialogs.

The next chapter deals with the predefined Motif selection dialogs. These dialogs allow you to provide the user with a group of choices from which to make a selection. Chapter 7, *Custom Dialogs*, discusses how you can breakaway from the predefined Motif dialogs and build dialogs on your own. Chapter 27, *Advanced Dialog Programming*, gets into advanced topics in Xt and Motif programming, using various types of MessageDialogs as examples.

In this chapter:

- *Types of SelectionDialogs*
- *SelectionDialogs*
- *PromptDialogs*
- *The Command Widget*
- *FileSelectionDialogs*
- *Summary*

6

Selection Dialogs

This chapter describes the predefined Motif selection-style dialogs. These dialogs display a list of items, such as files or commands, and allow the user to select items.

In Chapter 5, *Introduction to Dialogs*, we introduced the idea that dialogs are transient windows that perform a single task in an application. Dialogs may perform tasks that range from displaying a simple message, to asking a question, to providing a highly interactive window that obtains information from the user. The previous chapter also introduced MessageDialogs and discussed how they are used by the Motif toolkit.This chapter discusses SelectionDialogs, which are at the next level of complexity in predefined Motif dialogs.

In general, SelectionDialogs are used to present the user with a list of choices. The user can also enter a new selection or edit an existing one by typing in a text area in the dialog. SelectionDialogs are appropriate when the user is supposed to respond to the dialog with more than just a simple yes or no answer. With respect to the action area, SelectionDialogs have the same default button as MessageBoxes (e.g., *OK*, *Cancel*, and *Help*). The dialogs also provide an *Apply* button, but the button is not always managed by default. SelectionDialogs are meant to be less transient than MessageDialogs, since the user is expected to do more than read a message.

Types of SelectionDialogs

As explained in Chapter 5, there are four kinds of SelectionDialogs. The SelectionDialog and the PromptDialog are compound objects composed of a SelectionBox and a DialogShell. To use these objects, you need to include the header file <*Xm/SelectioB.h*>. The FileSelectionDialog is another compound object made up of a FileSelectionBox and a DialogShell. The include file for this object is <*Xm/FileSB.h*>. The Command widget is somewhat different, in that it is typically used as part of a larger interface, rather than as a dialog. To use the Command widget, include the file <*Xm/Command.h*>. You can create each of these dialogs using the associated convenience routines:

```
Widget XmCreateSelectionBox (Widget parent, char *name, ArgList args, Cardinal num_args)
Widget XmCreateSelectionDialog (Widget parent, char *name, ArgList args, Cardinal num_args)
```

```
Widget XmCreatePromptDialog (Widget parent, char *name, ArgList args, Cardinal num_args)
Widget XmCreateFileSelectionBox (Widget parent, char *name, ArgList args,
                                                    Cardinal num_args)
Widget XmCreateFileSelectionDialog (Widget parent, char *name, ArgList args,
                                                    Cardinal num_args)
Widget XmCreateCommand (Widget parent, char *name, ArgList args, Cardinal num_args)
```

Like the MessageDialog convenience routines, each of the SelectionDialog routines creates a dialog widget. In addition, routines that end in `Dialog` automatically create a DialogShell as the parent of the dialog widget. Note that the Command widget does not provide a convenience routine that creates a DialogShell; to put a Command widget in a DialogShell, you must create the DialogShell yourself. All of the convenience functions use the standard format for Motif creation routines.

The SelectionBox resource `XmNdialogType` specifies the type of dialog that has been created. The resource is set automatically by the dialog convenience routines. Unlike the `XmNdialogType` resource for MessageDialogs, the SelectionBox resource cannot be changed once the dialog has been created. The resource can have one of the following values:

```
XmDIALOG_WORK_AREA              XmDIALOG_PROMPT
XmDIALOG_SELECTION              XmDIALOG_COMMAND
XmDIALOG_FILE_SELECTION
```

These values should be self-explanatory, with the exception of `XmDIALOG_WORK_AREA`. This value is set when a SelectionBox is not the child of a DialogShell and it is not one of the other types of dialogs. In other words, if you create a SelectionDialog using `XmCreateSelectionDialog()`, the value is `XmDIALOG_SELECTION`, but if you use `XmCreateSelectionBox()`, the value is `XmDIALOG_WORK_AREA`. When a SelectionBox is created as the child of a DialogShell, the *Apply* button is automatically managed, except if `XmNdialogType` is set to `XmDIALOG_PROMPT`. Otherwise, the button is created but not managed.

The different types of SelectionDialogs are meant to be used for unique purposes. Each dialog provides different components that the user can interact with to perform a task. In the following sections, we examine each of the SelectionDialogs in turn.

SelectionDialogs

The SelectionDialog provides a ScrolledList that allows the user to select from a list of choices, as well as a TextField where the user can type in choices. When the user makes a selection from the list, the selected item is displayed in the text entry area. The user can also type new or existing choices into the text entry area directly. The dialog does not take any action until the user activates one of the buttons in the action area or presses the RETURN key. If the user double-clicks on an item in the List, the item is displayed in the text area and the *OK* button is automatically activated. Example 6-1 demonstrates the use of a SelectionDialog.*

Example 6-1 The select_dlg.c program

```
/* select_dlg.c -- display two pushbuttons: days and months.
** When the user selections one of them, post a selection
** dialog that displays the actual days or months accordingly.
** When the user selects or types a selection, post a dialog
** the identifies which item was selected and whether or not
** the item is in the list.
**
** This program demonstrates how to use selection boxes,
** methods for creating generic callbacks for action area
** selections, abstraction of data structures, and a generic
** MessageDialog posting routine.
*/

#include <Xm/SelectioB.h>
#include <Xm/RowColumn.h>
#include <Xm/MessageB.h>
#include <Xm/PushB.h>

Widget PostDialog();
char *days[] = {"Sunday", "Monday", "Tuesday", "Wednesday", "Thursday",
                "Friday", "Saturday"};
char *months[] = {"January", "February", "March", "April", "May", "June",
            "July", "August", "September", "October", "November", "December"};
typedef struct {
    char   *label;
    char   **strings;
    int    size;
} ListItem;

ListItem month_items = {"Months", months, XtNumber (months)};
ListItem days_items = {"Days", days, XtNumber (days)};

/* main() --create two pushbuttons whose callbacks pop up a dialog */
main (int argc, char *argv[])
{
    Widget        toplevel, button, rc;
    XtAppContext  app;
    void          pushed();

    XtSetLanguageProc (NULL, NULL, NULL);
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv, NULL,
                                    sessionShellWidgetClass, NULL);
    rc = XmCreateRowColumn (toplevel, "rowcolumn", NULL, 0);
    button = XmCreatePushButton (rc, month_items.label, NULL, 0);
    XtAddCallback (button, XmNactivateCallback, pushed,
                   (XtPointer) &month_items);
    XtManageChild (button);
    button = XmCreatePushButton (rc, days_items.label, NULL, 0);
```

* XtVaAppInitialize() is deprecated in X11R6. XmStringGetLtoR() is depecated in Motif 2.0: XmStringUnparse() is the preferred funtion.

```
    XtAddCallback (button, XmNactivateCallback, pushed,
                   (XtPointer) &days_items);
    XtManageChild (button);
    XtManageChild (rc);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}
/* pushed() --the callback routine for the main app's pushbutton.
** Create a dialog containing the list in the items parameter.
*/
void pushed (Widget widget, XtPointer client_data, XtPointer call_data)
{
    Widget      dialog;
    XmString    t, *str;
    int         i;
    extern void dialog_callback();
    ListItem *items = (ListItem *) client_data;

    str = (XmString *) XtMalloc (items->size * sizeof (XmString));
    t = XmStringCreateLocalized (items->label);

    for (i = 0; i < items->size; i++)
        str[i] = XmStringCreateLocalized (items->strings[i]);

    dialog = XmCreateSelectionDialog (widget, "selection", NULL, 0);
    XtVaSetValues (dialog,
                   XmNlistLabelString, t,
                   XmNlistItems, str,
                   XmNlistItemCount, items->size,
                   XmNmustMatch, True,
                   NULL);
    XtSetSensitive (XtNameToWidget (dialog, “Help”), False);
    XtAddCallback (dialog, XmNokCallback, dialog_callback, NULL);
    XtAddCallback (dialog, XmNnoMatchCallback, dialog_callback, NULL);
    XmStringFree (t);

    while (--i >= 0)
        XmStringFree (str[i]); /* free elements of array */

    XtFree ((char *) str); /* now free array pointer */
    XtManageChild (dialog);
}

/* dialog_callback() --The OK button was selected or the user
** input a name by himself. Determine whether the result is
** a valid name by looking at the "reason" field.
*/
void dialog_callback (Widget widget, XtPointer client_data,
                      XtPointer call_data)
{
    char   msg[256], *prompt, *value;
    int    dialog_type;
    XmSelectionBoxCallbackStruct *cbs =
    (XmSelectionBoxCallbackStruct *) call_data;
```

```
    switch (cbs->reason) {
        case XmCR_OK       : prompt = "Selection: ";
                             dialog_type = XmDIALOG_MESSAGE;
                             break;
        case XmCR_NO_MATCH : prompt = "Not a valid selection: ";
                             dialog_type = XmDIALOG_ERROR;
                             break;
        default            : prompt = "Unknown selection: ";
                             dialog_type = XmDIALOG_ERROR;
                             break;
    }
value = (char *) XmStringUnparse (cbs->value, XmFONTLIST_DEFAULT_TAG,
                    XmCHARSET_TEXT, XmCHARSET_TEXT, NULL, 0, XmOUTPUT_ALL);
    sprintf (msg, "%s%s", prompt, value);
    XtFree (value);
    (void) PostDialog (XtParent (XtParent (widget)), dialog_type, msg);
    if (cbs->reason != XmCR_NO_MATCH) {
        XtUnmanageChild (widget);
        /* The shell parent of the Selection box */
        XtDestroyWidget (XtParent (widget));
    }
}
/*
** Destroy the shell parent of the Message box, and thus the box itself
*/
void destroy_dialog (Widget dialog, XtPointer client_data, XtPointer call_data)
{
    XtDestroyWidget (XtParent (dialog));
    /* The shell parent of the Message box */
}
/*
** PostDialog() -- a generalized routine that allows the programmer
** to specify a dialog type (message, information, error, help,
** etc..), and the message to show.
*/
Widget PostDialog (Widget parent, int dialog_type, char *msg)
{
    Widget dialog;
    XmString text;
    dialog = XmCreateMessageDialog (parent, "dialog", NULL, 0);
    text = XmStringCreateLocalized (msg);
    XtVaSetValues (dialog, XmNdialogType, dialog_type,
                   XmNmessageString, text, NULL);
    XmStringFree (text);
    XtUnmanageChild (XtNameToWidget (dialog, “Cancel”));
    XtSetSensitive (XtNameToWidget (dialog, “Help”), False);
    XtAddCallback (dialog, XmNokCallback, destroy_dialog, NULL);
    XtManageChild (dialog);
    return dialog;
}
```

The output of the program is shown in Figure 6-1.

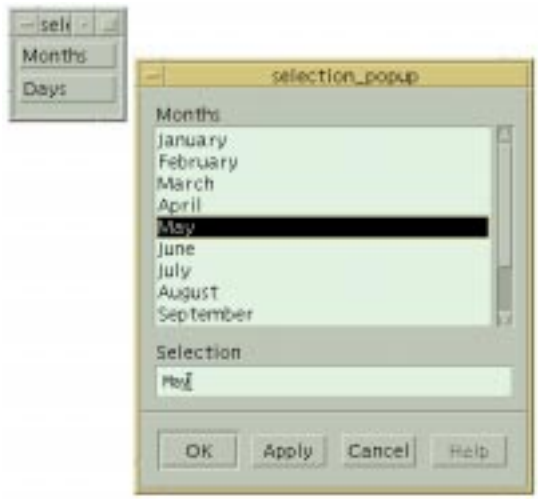


Figure 6-1 Output of the select_dlg program

The program displays two PushButtons, one for months and one for the days of the week. When either button is activated, a SelectionDialog that displays the list of items corresponding to the button is popped up. In keeping with the philosophy of modular programming techniques, we have broken the application into three routines - two callbacks and one general-purpose message posting function. The lists of day and month names are stored as arrays of strings.We have declared a data structure, `ListItem`, to store the label and the items for a list. Two instances of this data structure are initialized to the correct values for the lists of months and days. We pass these data structures as the `client_data` to the callback function `pushed()`. This callback routine is associated with both of the PushButtons.

The `pushed()` callback function creates the SelectionDialogs. Since the list of items for a SelectionDialog must be specified as an array of `XmString` values, the list passed in the `client_data` parameter must be converted. We create an array of compound strings the size of the list and copy each item into the new array using `XmStringCreateLocalized()`. The resulting list is used as the value for the `XmNlistItems` resource. The number of items in the list is specified as the value of the `XmNlistItemCount` resource. This value must be given for the list to be displayed. It must be less than or equal to the actual number of items in the list. We also set the `XmNlistLabelString` resource to specify the label for the list of items in the dialog. The SelectionDialog also provides the `XmNlistVisibleItemCount` resource for specifying the number of visible items in the list. We let the dialog use the default value for this resource.

The final resource that we set for the SelectionDialog is XmNmustMatch. This resource controls whether an item that the user types in the text entry area must match one of the items in the list. By setting the resource to True, we are specifying that the user cannot make up a month or day name. When the user activates the *OK* button or presses the RETURN key, the widget checks the item in the text entry area against those in the list. If the selection doesn't match any of the items in the list, the program pops up a dialog that indicates the error.

Once the dialog is created, we desensitize its *Help* button because we are not providing help. We install a callback routine for the *OK* button using the XmNokCallback. To handle the case when the user types an item that does not match, we also install a callback routine for the XmNnoMatchCallback. The dialog_callback() routine is used to handle both cases. We use the reason field of the callback structure to determine why the callback was called and act accordingly. The value field of the callback structure contains the selected item. If the item is valid, we use the value to create a dialog that confirms the selection. Otherwise, we post an error dialog that indicates the invalid selection. In both cases we use the generalized function, PostDialog(), to display the MessageDialog. If the selection is valid, the routine pops down and destroys the SelectionDialog. Otherwise, we leave the dialog posted so that the user can make another selection.

Just as a point of discussion, you should realize that it was an arbitrary decision to have the PostDialog() function accept char strings rather than an XmString. The routine could be modified to use an XmString, but doing so doesn't buy us anything. If you find that your application deals with one string format more often than the other, you may want to modify your routines accordingly. You should be aware that converting from one type of string to the other is relatively expensive; if it is done frequently, you may see an effect on performance. Another option is for your routine to accept both string types as different parameters. You can pass a valid value for one parameter and NULL for the other parameter and deal with them accordingly. For more information on handling compound strings, see Chapter 25.

Callback Routines

The SelectionDialog provides callbacks for its action buttons in the same way as the MessageDialog. Instead of accessing the PushButton widgets to install callbacks, you use the resources XmNokCallback, XmNapplyCallback, XmNcancelCallback, and XmNhelpCallback on the dialog widget itself. These callbacks correspond to each of the four buttons, *OK*, *Apply*, *Cancel*, and *Help*. The SelectionDialog also provides the XmNnoMatchCallback for handling the case when the item in the text entry area does not match an item in the list.

All of these callback routines take three parameters, just like any standard callback routine. The callback structure that is passed to all of the callback routines in the call_data

parameter is of type XmSelectionBoxCallbackStruct. This structure is similar to the one used by MessageDialogs, but it has more fields. The structure is declared as follows:

```
typedef struct {
    int          reason;
    XEvent       *event;
    XmString     value;
    int          length;
} XmSelectionBoxCallbackStruct;
```

The value of the reason field is an integer value that specifies the reason that the callback routine was invoked. The field can be one of the following values:

```
XmCR_OK          XmCR_APPLY          XmCR_CANCEL
XmCR_HELP        XmCR_NO_MATCH
```

The value and length fields represent the compound string version of the item that the user selected from the list or typed into the text entry area. In order to get the actual character string for the item, you have to use XmStringUnparse()* to convert the compound string into a character string. (See Chapter 25 for a discussion of compound strings.)

Internal Widgets

The SelectionDialog is obviously composed of primitive subwidgets, like PushButtons, Labels, a ScrolledList, and a TextField widget. For most tasks, it is possible to treat the dialog as a single entity because the dialog provides resources that manage the different components. However, there are some situations where it is useful to be able to get a handle to the widgets internal to the dialog. The XtNameToWidget()† routine allows you to access the internal widgets. This routine takes the following form:

```
Widget XtNameToWidget (Widget widget, char *child_name)
```

The *widget* parameter is a handle to a dialog widget, not its DialogShell parent. The *child_name* parameter specifies a particular subwidget in the dialog. For the SelectionDialog, the following are the names of the built-in components:

```
OK              Cancel          Help          Apply
Items           Selection       Text          Separator     ItemsList
```

These names are fairly self-explanatory: Selection is the Label associated with the SelectionDialog TextField widget, Items is the Label for the items list, and so forth. Note that ItemsList is the List itself, and not the ScrolledWindow containing the List. This means that ItemsList is not a direct child of the SelectionDialog, and hence you need to access the widget using a wildcard specification in XtNameToWidget(), as follows:

```
Widget list = XtNameToWidget (dialog, "*ItemsList");
```

---

* XmStringGetLtoR() is considered deprecated from Motif 2.0.

† XmSelectionBoxGetChild() is deprecated as of Motif 2.0.

One use of `XtNameToWidget()` is to get a handle to the *Apply* button so that you can manage it. When you create a SelectionBox that is not a child of a DialogShell, the toolkit creates the *Apply* button, but it is unmanaged by default. The *Apply* button is available to the PromptDialog, but it is unmanaged by default. To use the button, you must manage it and specify a callback routine, as in the following code fragment:

```
XtAddCallback (dialog, XmNapplyCallback, dialog_callback, NULL);
XtManageChild (XtNameToWidget (dialog, "Apply"));
```

The callback routine is the same as the one we set for the *OK* button, but the `reason` field in the callback structure will indicate that it was called as a result of the *Apply* button being activated.

PromptDialogs

The PromptDialog is unique among the SelectionDialogs, in that it does not create a ScrolledList object. For the PromptDialog, the following are the names of the built-in components:

```
OK                  Cancel          Help            Apply
Selection           Text            Separator
```

This dialog allows the user to type a text string in the text entry area and then enter it by selecting the *OK* button or by pressing the RETURN key. Example 6-2 shows an example of creating and using a PromptDialog.

Example 6-2 The prompt_dlg.c program

```
/* prompt_dlg.c -- prompt the user for a string. Two PushButtons
** are displayed. When one is selected, a PromptDialog is displayed
** allowing the user to type a string. When done, the PushButton's
** label changes to the string.
*/
#include <Xm/SelectioB.h>
#include <Xm/RowColumn.h>
#include <Xm/PushB.h>

main (int argc, char *argv[])
{
    XtAppContext   app;
    Widget         toplevel, rc, button;
    void           pushed(Widget, XtPointer, XtPointer);

    XtSetLanguageProc (NULL, NULL, NULL);

    /* Initialize toolkit and create toplevel shell */
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv, NULL,
                                    sessionShellWidgetClass, NULL);
    /* RowColumn managed both PushButtons */
    rc = XmCreateRowColumn (toplevel, "rowcol", NULL, 0);
    /* Create two pushbuttons -- both have the same callback */
```

```
    button = XmCreatePushButton (rc, "PushMe 1", NULL, 0);
    XtAddCallback (button, XmNactivateCallback, pushed, NULL);
    XtManageChild (button);
    button = XmCreatePushButton (rc, "PushMe 2", NULL, 0);
    XtAddCallback (button, XmNactivateCallback, pushed, NULL);
    XtManageChild (button);
    XtManageChild (rc);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

/*
** Destroy the prompt dialog's shell parent, and thus also the prompt
*/
void destroy_dialog (Widget w, XtPointer client_data, XtPointer call_data)
{
    XtDestroyWidget (XtParent (w));
}

/* pushed() --the callback routine for the main app's pushbuttons.
** Create a dialog that prompts for a new button name.
**/
void pushed (Widget widget, XtPointer client_data, XtPointer call_data)
{
    Widget       dialog;
    XmString     t = XmStringCreateLocalized ("Enter New Button Name:");
    void         read_name(Widget, XtPointer, XtPointer);
    Arg          args[5];
    int          n = 0;
    /* Create the dialog -- the PushButton acts as the DialogShell's
    ** parent (not the parent of the PromptDialog).
    */
    XtSetArg (args[n], XmNselectionLabelString, t); n++;
    XtSetArg (args[n], XmNautoUnmanage, False); n++;
    dialog = XmCreatePromptDialog (widget, "prompt", args, n);
    XmStringFree (t);
    /* always destroy compound strings when done */
    /* When the user types the name, call read_name()... */
    XtAddCallback (dialog, XmNokCallback, read_name, (XtPointer) widget);
    /* If the user selects cancel, just destroy the dialog */
    XtAddCallback (dialog, XmNcancelCallback, destroy_dialog, NULL);
    /* No help is available... */
    XtSetSensitive (XtNameToWidget (dialog, "Help"), False);
    XtManageChild (dialog);
}

/* read_name() --the text field has been filled in. */
void read_name (Widget widget, XtPointer client_data, XtPointer call_data)
{
    Widget push_button = (Widget) client_data;
    XmSelectionBoxCallbackStruct *cbs;

    cbs = (XmSelectionBoxCallbackStruct *) call_data;
    XtVaSetValues (push_button, XmNlabelString, cbs->value, NULL);
```

```
        /* Name's fine -- go ahead and enter it */
        XtDestroyWidget (XtParent (widget));
    }
```

The output of the program is shown in Figure 6-2.

Figure 6-2 Output of the prompt_dlg program

The callback routine for each of the PushButtons, `pushed()`, creates a PromptDialog that prompts the user to enter a new name for the PushButton. The PushButton is passed as the `client_data` to the `XmNokCallback` routine, `read_name()`, so that the routine can set the label of the PushButton directly from inside the callback. The `read_name()` function destroys the dialog once it has set the label, since the dialog is no longer needed.

If the *Cancel* button is pressed, the text is not needed, so we can simply destroy the dialog, using the `destroy_dialog`() callback. We set `XmNautoUnmanage` to `False` for the dialog because the application is assuming the responsibility of managing the dialog. There is no help for the dialog so the *Help* button is disabled by setting it insensitive.

The text area in the PromptDialog is a TextField widget, so you can get a handle to it and set TextField widget resources accordingly. Use `XtNameToWidget()` to access the widget. In order to promote the single-entity abstraction, the dialog provides two resources that affect the TextField widget. You can set the `XmNtextString` resource to change the value of the text string in the widget. Like other string resources, the value for this resource must be a compound string. The `XmNtextColumns` resource specifies the width of the TextField in columns.

One frustrating feature of the predefined SelectionDialogs is that when they are popped up, the TextField widget does not necessarily receive the keyboard focus by default. If the user is not paying attention, starts typing, and then presses the RETURN key, all of the keystrokes will be thrown away except the RETURN, which will activate the *OK* button. This problem is solved through the `XmNinitialFocus` resource. This resource specifies the widget that has the keyboard focus the first time that the dialog is popped up. The text entry area is the default value of the resource for SelectionDialogs. You can also program around the problem by using `XmProcessTraversal()` to set the focus to a particular widget.

The Command Widget

A Command widget allows the user to enter commands and have them saved in a history list widget for later reference. The Command widget is composed of a text entry area and a command history list. Unlike all of the other predefined Motif dialogs, this widget does not provide any action area buttons. The widget does provide a convenient interface for applications that have a command-driven interface, such as a debugger.

You can use the convenience routine `XmCreateCommand()` to create a Command widget or you can use `XtVaCreateWidget()` with the class `xmCommandWidgetClass`. Motif does not provide a convenience routine for creating a Command widget in a DialogShell. The rationale is that the Command widget is intended to be used on a more permanent basis, since it accumulates a history of command input. A Command widget is typically used as part of a larger interface, such as in a MainWindow, which is why it does not have action buttons. (See Chapter 4, *The Main Window*, for an example.) If you want to create a CommandDialog, you will have to create the DialogShell widget yourself and make the Command widget its immediate child. See Chapter 5, for more information about DialogShells.

The Command widget class is subclassed from SelectionBox. There are similarities between the two widgets, in that the user has the ability to select items from a list. However, the list is composed of the commands that have been previously entered. When the user enters a command, it is added to the list. If the user selects an item from the command history list, the command is displayed in the text entry area. Although the Command widget inherits resources from the SelectionBox, many of the resources are not applicable since the Command widget does not have any action area buttons. None of the SelectionBox resources for setting the labels and callbacks of the buttons apply to the Command widget. For the Command widget, the following are the names of the built-in components:

```
Selection          Text          ItemsList
```

The Command widget provides a number of resources that can be used to control the command history list. The `XmNhistoryItems` and `XmNhistoryItemCount` resources specify the list of commands and the number of commands in the list. The `XmNhistoryVisibleItemCount` resource controls the number of items that are visible in the command history. `XmNhistoryMaxItems` specifies the maximum number of items in the history list. When the maximum value is reached, a command is removed from the beginning of the list to make room for each new command that is entered.

The Command widget provides two callback resources, `XmNcommandEnteredCallback` and `XmNcommandChangedCallback`, for the text entry area. When the user changes the text in the command entry area, the `XmNcommandChangedCallback` is invoked. If the user presses the RETURN key or double-clicks on an item in the command history list, the `XmNcommandEnteredCallback` is called. The callback routine for each of the callbacks takes the usual three parameters. The callback structure passed to the routines in the `call_`

`data` parameter is of type `XmCommandCallbackStruct`, which is identical to the `XmSelectionBoxCallbackStruct`. The possible values for the `reason` field in the structure are `XmCR_COMMAND_ENTERED` and `XmCR_COMMAND_CHANGED`.

You can get a handle to the subwidgets of the Command widget using function `XtNameToWidget()`*.

In order to support the idea that the dialog is a single widget, the toolkit also provides a number of convenience routines that you can use to modify the Command widget. The function `XmCommandSetValue()` sets the text in the command entry area of the dialog. The function takes the following form:

```
void XmCommandSetValue (Widget widget, XmString command)
```

The *command* is displayed in the command entry area. The Command widget resource `XmNcommand` specifies the text for the command entry area, so you can also set this resource directly. Alternatively, you can use `XmTextSetString()` on the Text widget in the dialog to set the command. However, note that the string you specify to this function is a regular character string, not a compound string.

If you want to append some text to the string in the command entry area, you can use the routine `XmCommandAppendValue()`, which takes the following form:

```
void XmCommandAppendValue (Widget widget, XmString command)
```

The *command* is added to the end of the string in the command entry area. The function `XmCommandError()` displays an error message in the history area of the Command widget. The function takes the following form:

```
void XmCommandError (Widget widget, XmString message)
```

The error message is displayed until the user enters the next command.

FileSelectionDialogs

Like the Command widget, the FileSelectionBox is subclassed from SelectionBox. The FileSelectionDialog looks somewhat different than the other selection dialogs because of its complexity and its unusual widget layout and architecture. Functionally, the FileSelectionDialog allows the user to move through the file system and select a file or a directory for use by the application. The dialog also lets the user specify a filter that controls the files that are displayed in the dialog. This filter is generally specified as a regular expression reminiscent of the classic UNIX meta-characters (e.g., `*` matches all files, while `*.c` matches all files that end in *.c*).

* `XmSelectionBoxGetChild`() is deprecated as of Motif 2.0.

Figure 6-3 shows a FileSelectionDialog.

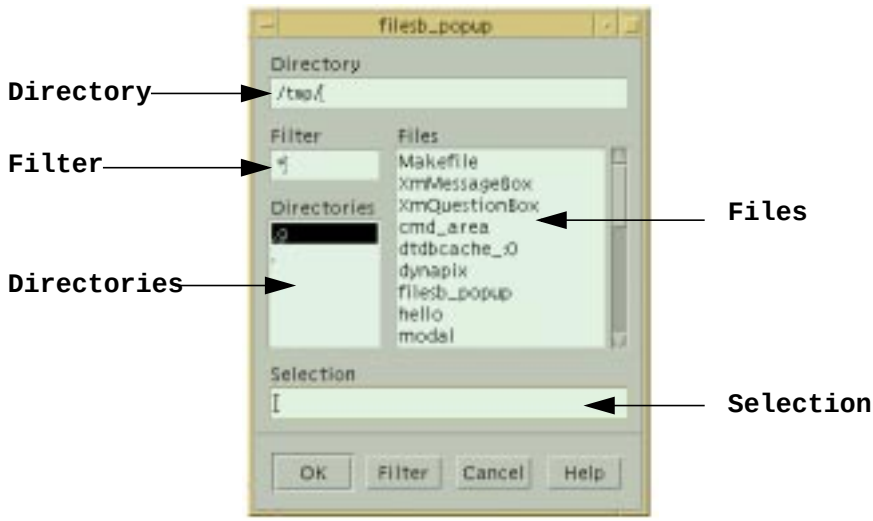


Figure 6-3 A typical FileSelectionDialog

The control area of the FileSelectionDialog has potentially five components*: a `Filter` text area, a current `Directory` field, a *Directories* list displaying the directories in the current directory specified by the filter, a *Files* list area containing files within the current directory, and a *Selection* area. If the user selects a directory, the *Directory* field is modified to reflect the selection. The *Files* list shows the files in the current directory. The *Selection* text entry area specifies the file selected by the user. If the user selects a file from the *Files* list, the full pathname is displayed in the *Selection* text entry area.

The Motif 1.2 FileSelectionBox contained only four areas: the data displayed in the Filter and Directory fields was concatenated into a single TextField, with the filter pattern appended onto the current directory name.

For backwards compatibility, this is also true of the Motif 2.x FileSelectionBox, depending upon the value of the `XmNpathMode` resource. If `XmNpathMode` is `XmPATH_MODE_FULL`, the FileSelectionBox has Motif 1.2 behavior; for separate filter and directory fields, set `XmNpathMode` to `XmPATH_MODE_RELATIVE`. Figure 6-3 displays the File Selection Box in the path relative mode.

The FileSelectionDialog has four buttons in its action area. The *OK*, *Cancel*, and *Help* buttons are the same as for other SelectionDialogs. The *Filter* button acts on the directory

* The Motif 1.2 FileSelectionBox has only four component areas.

and pattern specified in the filter text entry area. For example, the user could enter */usr/src/motif/lib/Xm* as the directory and * as the filter*. When the user selects the *Filter* button or presses RETURN in the Text widget, the directory part of the filter is searched and all of the directories within that directory are displayed in the directories list. The pattern part is then used to find all of the matching files in the directory and the files are shown in the files list. Only files are placed in this list; directories are excluded since they are listed separately.

While this process seems straightforward, it become confusing in Motif 1.2 for users and programmers alike because of the way that the widget parsed the filter in the single Filter field. For example, consider the following string: */usr/src/motif/lib/Xm*. This pathname appears to be a common directory path, but in fact, the widget interpreted the filter so that the directory is */usr/src/motif/lib* and the pattern is *Xm*. If searched, the directories list will contain all the directories in */usr/src/motif/lib* and the files list won't contain anything because `Xm` is a directory, not a pattern that matches any files. Since users frequently made this mistake when using the FileSelectionDialog, you had to be sure to explain the operation of the dialog in the documentation for your application. For Motif 2.1, with the filter and directory portions placed in separate text fields, the issue is much clarified as far as the user is concerned.

For a File Selection Box which has the path mode as full (Motif 1.2 compatible), the convention that the widget follows is to use the last `/` in the filter to separate the directory part from the pattern part. Fortunately, the FileSelectionDialog provides resources and other mechanisms to retrieve the proper parts of the filter specification. We will demonstrate how to use these mechanisms in the next few subsections.

Creating a FileSelectionDialog

The convenience function for creating a FileSelectionDialog is `XmCreateFileSelectionDialog()`. The routine is declared in <*Xm/FileSB.h*>. The function creates a FileSelectionBox widget and its DialogShell parent and returns the FileSelectionBox. Alternatively, you can create a FileSelectionBox widget using either `XmCreateFileSelectionBox()` or `XtVaCreateWidget()` with the widget class specified as `xmFileSelectionBoxWidgetClass`. In this case, you could use the widget as part of a larger interface, or put it in a DialogShell yourself.

Example 6-3 demonstrates how a FileSelectionDialog can be created. This program produces the dialog shown in Figure 6-3. The intent of the program is to display a single FileSelectionDialog and print the selection that is made. We will provide a more realistic example shortly. For now, you should notice how little code is actually required to create the dialog.†

* In Motif 1.2, the user would enter */usr/src/motif/lib/Xm/** in the single Filter field.

Example 6-3 The show_files.c program

```
/* show_files.c -- introduce FileSelectionDialog; print the file
** selected by the user.
*/

#include <Xm/FileSB.h>

main (int argc, char *argv[])
{
    Widget         toplevel, text_w, dialog;
    XtAppContext   app;
    extern void    exit(int);
    void           echo_file(Widget, XtPointer, XtPointer);
    Arg            args[2];

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

    /* Create a simple Motif 2.1 FileSelectionDialog */
    XtSetArg (args[0], XmNpathMode, XmPATH_MODE_RELATIVE);
    dialog = XmCreateFileSelectionDialog (toplevel, "filesb", args, 1);
    XtAddCallback (dialog, XmNcancelCallback, (void (*)()) exit, NULL);
    XtAddCallback (dialog, XmNokCallback, echo_file, NULL);
    XtManageChild (dialog);
    XtAppMainLoop (app);
}

/* callback routine when the user selects OK in the FileSelection
** Dialog. Just print the file name selected.
**/
void echo_file (Widget widget, /* file selection box */
                XtPointer client_data,
                XtPointer call_data)
{
    char *filename;
    XmFileSelectionBoxCallbackStruct *cbs =
                        (XmFileSelectionBoxCallbackStruct *) call_data;

    filename = (char *) XmStringUnparse (cbs->value,
                                         XmFONTLIST_DEFAULT_TAG,
                                         XmCHARSET_TEXT,
                                         XmCHARSET_TEXT,
                                         NULL, 0, XmOUTPUT_ALL);
    if (!filename)
        /* must have been an internal error */
        return;
    if (!*filename) {
        /* nothing typed? */
```

† XtVaAppInitialize() is deprecated in X11R6. XmStringGetLtoR() is deprecated in Motif 2.0: prefer XmStringUnparse().

```
            puts ("No file selected.");
            /* even "" is an allocated byte */
            XtFree (filename);
            return;
        }
        printf ("Filename given: \"%s\"\n", filename);
        XtFree (filename);
    }
```

The program simply prints the selected file when the user activates the *OK* button. The user can change the file by selecting an item from the files list or by typing directly in the selection text entry area. The user can also activate the dialog by double-clicking on an item in the files list. The FileSelectionDialog itself is very simple to create; most of the work in the program is done by the callback routine for the *OK* button.

Internal Widgets

A FileSelectionDialog is made up of a number of subwidgets, including Text, List, and PushButton widgets. You can get the handles to these children using the routine `XtNameToWidget()`.*

FileSelectionDialog can manage a work area child: you can customize the operation of a FileSelectionDialog by adding a work area that contains other components. For a detailed discussion of this technique, see Chapter 7, *Custom Dialogs*.

Getting the children of a FileSelectionDialog is not necessary in most cases because the Motif toolkit provides FileSelectionDialog resources that access most of the important resources of the children. You should only get handles to the children if you need to change resources that are not involved in the file selection mechanisms. For the FileSelectionBox widget, the following are the names of the built-in components:

```
Apply           Cancel          Help            ItemsList
Items           OK              Selection       Tex
Separator       FilterLabel     FilterText      Dir
DirList         DirL            DirText
```

These values should be self-explanatory; *DirL* is the Label associated with the new Motif 2.1 separate Directory field (`XmNpathMode` equals `XmPATH_MODE_RELATIVE`), and *DirText* is the separate Directory TextField itself. Note that as in the case of the SelectionDialog, *ItemsList* is the List itself, and not the ScrolledWindow containing the List. This means that ItemsList is not a direct child of the FileSelectionDialog, and hence you need to access the widget using a wildcard specification in `XtNameToWidget`(), as follows:

```
Widget list = XtNameToWidget (fsb, "*ItemsList");
```

* `XmFileSelectionBoxGetChild`() is deprecated from Motif 2.0 onwards. It has not been maintained: there are no bit masks to access the Motif 2.0 Directory Label (`DirL`) and Text (`DirText`) which are displayed when `XmNpathMode` is `XmPATH_MODE_RELATIVE`.

Callback Routines

The XmNokCallback, XmNcancelCallback, XmNapplyCallback, XmNhelpCallback, and XmNnoMatchCallback callbacks can be specified for a FileSelectionDialog as they are for SelectionDialog. The callback routines take the usual parameters, but the callback structure passed in the call_data parameter is of type XmFileSelectionBoxCallbackStruct. The structure is declared as follows:

```
typedef struct {
    int          reason;
    XEvent       *event;
    XmString     value;
    int          length;
    XmString     mask;
    int          mask_length;
    XmString     dir;
    int          dir_length;
    XmString     pattern;
    int          pattern_length;
} XmFileSelectionBoxCallbackStruct;
```

The value of the reason field is an integer value that specifies the reason that the callback routine was invoked. The possible values are the same as those for a SelectionDialog:

```
XmCR_OK              XmCR_APPLY              XmCR_CANCEL
XmCR_HELP            XmCR_NO_MATCH
```

The value field contains the item that the user selected from the files list or typed into the selection text entry area. The value corresponds to the XmNdirSpec resource and it does not necessarily have to match an item in the directories or files lists.The mask field corresponds to the XmNdirMask resource; it represents a combination of the entire pathname specification in the filter. The dir and pattern fields represent the two components that make up the mask. All of these fields are compound strings; they can be converted to character strings using XmStringUnparse().*

File Searching

You can force a FileSelectionDialog to reinitialize the directory and file lists by calling XmFileSelectionDoSearch(). This routine reads the directory filter and scans the specified directory, which is useful if you set the mask directly. The function takes the following form:

```
void XmFileSelectionDoSearch ( XmFileSelectionBoxWidget   widget,
                               XmString                   dirmask)
```

When the routine is called, the widget invokes its directory search procedure and sets the text in the filter text entry area to the *dirmask* parameter. Calling

* XmStringGetLtoR() is deprecated from Motif 2.0 onwards.

`XmFileSelectionDoSearch()` has the same effect as setting the filter and selecting the *Filter* button.

By default, the FileSelectionDialog searches the directory specified in the mask according to its internal searching algorithm. You can replace this file searching procedure with your own routine by specifying a callback routine for the `XmNfileSearchProc` resource. This resource is not a callback list, so you do not install it by calling `XtAddCallback()`. Since the resource is just a single procedure, you specify it as a value like you would any other resource, as shown in the following code fragment:

```
extern void my_search_proc(Widget, XtPointer, XtPointer);
XtVaSetValues (file_selection_dialog,
                          XmNfileSearchProc, my_search_proc, NULL);
```

If you specify a search procedure, it is used to generate the list of filenames for the files list. A file search routine takes the following form:

```
void (* XmSearchProc) (Widget widget, XtPointer search_data)
```

The *`widget`* parameter is the actual FileSelectionBox widget and *`search_data`* is a pointer to a callback structure of type `XmFileSelectionBoxCallbackStruct`. This structure is just like the one used in the callback routines discussed in the previous section. Do not be concerned with the value of the `reason` field in this situation because none of the routines along the way use the value. The search function should scan the directory specified by the `dir` field of the *`search_data`* parameter. The `pattern` should be used to filter the files within the directory. You can get the complete filter from the `mask` field.

After the search procedure has determined the new list of files that it is going to use, it must set the `XmNfileListItems` and `XmNfileListItemCount` resources to store the list into the List widget used by the FileSelectionDialog. The routine must also set the `XmNlistUpdated` resource to `True` to indicate that it has indeed done something, whether or not any files are found. The function can also set the `XmNdirSpec` resource to reflect the full file specification in the selection text entry area, so that if the user selects the *OK* button, the specified file is used. Although this step is optional, we recommend doing it in case the old value is no longer valid.

To understand why it may be necessary to have your own file search procedure, consider how you would customize a FileSelectionDialog so that it only displays the writable files in an arbitrary directory. This customization might come in handy for a save operation in an electronic mail application, where the user invokes a *Save* action that displays a FileSelectionDialog that lists the files in which the user can save messages. Files that are not writable should not be displayed in the dialog. Example 6-4 shows an example of how a file search procedure can be used to implement this type of dialog.*

* `XtVaAppInitialize`() is deprecated in X11R6. `XmStringGetLtoR`() is deprecated in Motif 2.0: prefer `XmStringUnparse`().

Example 6-4The file_sel.c program

```
/* file_sel.c -- file selection dialog displays a list of all the writable
** files in the directory described by the XmNmask of the dialog.
** This program demonstrates how to use the XmNfileSearchProc for
** file selection dialog widgets.
*/
#include <stdio.h>
#include <Xm/Xm.h>
#include <Xm/FileSB.h>
#include <Xm/DialogS.h>
#include <Xm/PushBG.h>
#include <Xm/PushB.h>
#include <X11/Xos.h>
#include <sys/stat.h>

void do_search(Widget XtPointer, XtPointer);
void new_file_cb(Widget, XtPointer, XtPointer);
/* routine to determine if a file is accessible, a directory,
** or writable. Return -1 on all errors or if the file is not
** writable. Return 0 if it's a directory or 1 if it's a plain
** writable file.
*/
int is_writable (char *file)
{
    struct stat s_buf;
    /* if file can't be accessed (via stat()) return. */
    if (stat (file, &s_buf) == -1)
        return -1;
    else if ((s_buf.st_mode & S_IFMT) == S_IFDIR)
        return 0; /* a directory */
    else if (!(s_buf.st_mode & S_IFREG) || access (file, W_OK) == -1)
        /* not a normal file or it is not writable */
        return -1;
    /* legitimate file */
    return 1;
}

/* main() -- create a FileSelectionDialog
*/
main (int argc, char *argv[])
{
    Widget          toplevel, dialog;
    XtAppContext    app;
    extern void     exit(int);
    Arg             args[5];
    int             n = 0;
    XtSetLanguageProc (NULL, NULL, NULL);
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv, NULL,
                                    sessionShellWidgetClass, NULL);

    XtSetArg (args[n], XmNfileSearchProc, do_search); n++;
    dialog = XmCreateFileSelectionDialog (toplevel, "Files", args, n);
    XtSetSensitive (XtNameToWidget (dialog, “Help”), False);
```

```
    /* if user presses OK button, call new_file_cb() */
    XtAddCallback (dialog, XmNokCallback, new_file_cb, NULL);
    /* if user presses Cancel button, exit program */
    XtAddCallback (dialog, XmNcancelCallback, (void (*)()) exit, NULL);
    XtManageChild (dialog);
    XtAppMainLoop (app);
}

/* a new file was selected -- check to see if it's readable and not
** a directory. If it's not readable, report an error. If it's a
** directory, scan it just as though the user had typed it in the mask
** Text field and selected "Search".
*/
void new_file_cb (Widget widget, XtPointer client_data,
                  XtPointer call_data)
{
    char *file;
    XmFileSelectionBoxCallbackStruct *cbs = (XmFileSelectionBoxCallbackStruct
                                             *) call_data;
    /* get the string typed in the text field in char * format */
    if (!(file = (char *) XmStringUnparse (cbs->value,
                                           XmFONTLIST_DEFAULT_TAG,
                                           XmCHARSET_TEXT,
                                           XmCHARSET_TEXT,
                                           NULL, 0, XmOUTPUT_ALL)))
        return;
    if (*file != '/') {
        /* if it's not a directory, determine the full pathname
        ** of the selection by concatenating it to the "dir" part
        */
        char *dir, *newfile;
        if (dir = XmStringUnparse (cbs->dir,
                                   XmFONTLIST_DEFAULT_TAG,
                                   XmCHARSET_TEXT,
                                   XmCHARSET_TEXT,
                                   NULL, 0, XmOUTPUT_ALL)) {
            newfile = XtMalloc (strlen (dir) + 1 + strlen (file) + 1);
            sprintf (newfile, "%s/%s", dir, file);
            XtFree (file);
            XtFree (dir);
            file = newfile;
        }
    }
    switch (is_writable (file)) {
        case 1: puts (file); /* or do anything you want */ break;
        case 0: {
            /* a directory was selected, scan it */
            XmString str = XmStringCreateLocalized (file);
            XmFileSelectionDoSearch (widget, str);
            XmStringFree (str);
            break;
        }
        case -1:
            /* a system error on this file */
            perror (file);
```

```
    }
    XtFree (file);
}

/* do_search() -- scan a directory and report only those files that
** are writable. Here, we let the shell expand the (possible)
** wildcards and return a directory listing by using popen().
** A *real* application should -not- do this; it should use the
** system's directory routines: opendir(), readdir() and closedir().
*/
void do_search (Widget widget, /* file selection box widget */
                XtPointer search_data, XtPointer call_data)
{
    char          *mask, buf[BUFSIZ], *p;
    XmString       names[256]; /* maximum of 256 files in dir */
    int            i = 0;
    FILE          *pp, *popen();
    XmFileSelectionBoxCallbackStruct *cbs = (XmFileSelectionBoxCallbackStruct
                                             *) search_data;
    if (!(mask = (char *) XmStringUnparse (cbs->mask,
                                           XmFONTLIST_DEFAULT_TAG,
                                           XmCHARSET_TEXT,
                                           XmCHARSET_TEXT,
                                           NULL, 0, XmOUTPUT_ALL)))
        return; /* can't do anything */
    sprintf (buf, "/bin/ls %s", mask);
    XtFree (mask);
    /* let the shell read the directory and expand the filenames */
    if (!(pp = popen (buf, "r")))
        return;
    /* read output from popen() -- this will be the list of files */
    while (fgets (buf, sizeof buf, pp)) {
        if (p = index (buf, '\n'))
            *p = 0;
        /* only list files that are writable and not directories */
        if (is_writable (buf) == 1 &&
            (names[i] = XmStringCreateLocalized (buf))) i++;
    }
    pclose (pp);
    if (i) {
        XtVaSetValues (widget,
                       XmNfileListItems, names,
                       XmNfileListItemCount, i,
                       XmNdirSpec, names[0],
                       XmNlistUpdated, True,
                       NULL);
        while (i > 0)
            XmStringFree (names[--i]);
    } else
        XtVaSetValues (widget,
                       XmNfileListItems, NULL,
                       XmNfileListItemCount, 0,
                       XmNlistUpdated, True,
                       NULL);
}
```

The program simply displays a FileSelectionDialog that only lists the files that are writable by the user. The directories listed may or may not be writable. We are not testing that case here as it is handled by another routine that deals specifically with directories, which are discussed in the next section. The `XmNfileSearchProc` is set to `do_search()`, which is our own routine that creates the list of files for the files List widget. The function calls `is_writable()` to determine if a file is accessible and if it is a directory or a regular file that is writable.

The callback routine for the *OK* button is set to `new_file_cb()` through the `XmNokCallback` resource. This routine is called when a new file is selected in from the files list or new text is entered in the selection text entry area and the *OK* button is pressed. The specified file is evaluated using `is_writable()` and acted on accordingly. If it is a directory, the directory is scanned as if it had been entered in the filter text entry area. If the file cannot be read, an error message is printed. Otherwise, the file is a legitimate selection and, for demonstration purposes, the filename is printed to `stdout`.

Obviously, a real application would do something more appropriate in each case; errors would be reported using ErrorDialogs and legitimate values would be used by the application. An example of such a program is given in Chapter 18, *Text Widgets*, as *file_browser.c*. This program is an extension of Example 6-4 that takes a more realistic approach to using a FileSelectionDialog. Of course, the intent of that program is to show how Text widgets work, but its use of dialogs is consistent with the approach we are taking here.

Directory Searching

The FileSelectionDialog also provides a directory searching function that is analogous to the file searching function. While file searching may be necessary for some applications, it is less likely that customized directory searching will be as useful, since the default action taken by the toolkit should cover all common usages. However, since it is impossible to second-guess the requirements of all applications, Motif allows you to specify a directory searching function through the `XmNdirSearchProc` resource.

The procedure is used to create the list of directories. The method used by the procedure is virtually identical to the one used for files, except that the routine must set different resources. The routine must set the `XmNdirListItems` and `XmNdirListItemCount` resources to store the list of directories in the List widget. The value for `XmNlistUpdated` must be set just as it was for the file selection routine and `XmNdirectoryValid` must also be set to either `True` or `False`. If the directory cannot be read, `XmNdirectoryValid` is set to `False` to prevent the `XmNfileSearchProc` from being called. In this way, the file searching procedure is protected from getting invalid directories from the directory searching procedure.

The Search Process

In order to fully customize the directory and file searching functions in a FileSelectionDialog, it is important to understand exactly how the dialog works. This material is advanced and is intended for programmers who need to write advanced file and/or directory searching routines. When the user or the application invokes a directory search, the FileSelectionDialog performs the following tasks:

1. The List widgets are unmapped to give the user immediate feedback that something is happening. So, if a file and/or directory search takes along time, the user has a visual cue that the application is not waiting for input.
2. All of the items are deleted from the List widgets.
3. The widget calls its qualify search procedure to construct a proper directory mask, base directory, and file search pattern based on the text in the filter text entry area. The procedure creates a callback structure of the type `XmFileSelectionBoxCallbackStruct` for use by the directory and file search routines.
4. The `XmNdirSearchProc` function is called with the callback structure constructed by the qualify search procedure. The directory search routine checks to be sure that it can search the specified directory and if it can, it creates the list of directories for the dialog. If the directory cannot be searched, the routine sets `XmNdirectoryValid` to `False`.
5. The `XmNfileSearchProc` function is called if `XmNdirectoryValid` has been set to `True`. This routine creates the list of files for the dialog. If `XmNdirectoryValid` has been set to `False`, the file list remains empty.

Just as for the directory and file search routines, you can write your own qualify search procedure and install it as the value for the `XmNqualifySearchProc` resource. The routine takes the following form:

```
void (* XmQualifyProc) ( Widget       widget,
                         XtPointer    input_data,
                         XtPointer    output_data)
```

The *widget* parameter is the actual FileSelectionBox widget; *input_data* and *output_data* are pointers to callback structures of type `XmFileSelectionBoxCallbackStruct`. *input_data* contains the directory information that needs to be qualified. The routine uses this information to fill in the *output_data* callback structure that is then passed to the directory and file search procedures.

The `XmNfileTypeMask` resource indicates the types of files for which a particular search routine should be looking. The resource can be set to one of the following values:

```
XmFILE_REGULAR          XmFILE_DIRECTORY          XmFILE_ANY_TYPE
```

If you are using the same routine for both the `XmNdirSearchProc` and the `XmNfileSearchProc`, you can query this resource to determine the type of file to search for.

Summary

This chapter described the different types of selection dialogs provided by the Motif toolkit. These dialogs implement some common functionality that is needed by many different applications. This chapter builds on the material in Chapter 5, *Introduction to Dialogs*, which introduced the concept of dialogs and discussed the basic mechanisms that implement them. While the dialogs are designed to be used as single-entity abstractions, they can be customized to provide additional functionality as necessary. We describe how to customize the dialogs and how to create your own dialogs in Chapter 7.

In this chapter:

- *Modifying Motif Dialogs*
- *Designing New Dialogs*
- *Building a Dialog*
- *Generalizing the Action Area*
- *Using a TopLevelShell for a Dialog*
- *Positioning Dialogs*
- *Summary*

7

Custom Dialogs

This chapter describes how to create new types of dialogs, either by customizing Motif dialogs or by creating entirely new dialogs.

In this chapter we examine methods for creating your own dialogs. The need for such dialogs exists when those provided by Motif are too limited in functionality or are not specialized enough for your application. Sometimes it is not clear when you need to create your own dialog. In some situations, you may find that a Motif dialog would be just fine if only they did this one little thing. Fortunately, you can often make small adjustments to a predefined Motif dialog, rather than building an entirely new dialog box from scratch.

There are some issues to consider before you decide how you want to approach the problem of developing custom dialogs. For example, do you want to use your own widget layout or is the layout of one of the predefined dialogs sufficient? Do you have specialized user-interface appearance and functionality needs that go beyond what is provided by Motif? The answers to these questions affect the design of your dialogs. The discussion and examples provided in this chapter address both scenarios. We provide information on how to create dialogs that are based on the predefined Motif dialogs, as well as how to design completely new dialogs.

Before we get started, we should mention that creating your own dialogs makes heavy use of manager widgets, such as the Form, BulletinBoard, RowColumn, and PanedWindow widgets. While we use and describe the manager widgets in context, you may want to consult Chapter 8, *Manager Widgets*, for specific details about these widgets.

Modifying Motif Dialogs

We begin by discussing the simpler case of modifying existing Motif dialogs. In Chapter 5, *Introduction to Dialogs*, we showed you how to modify a dialog to some extent by changing the default labels on the buttons in the action area or by unmanaging or desensitizing certain components in the dialog. What we did not mention is that you can also add new components to a dialog box to expand its functionality. All of the predefined Motif dialog widgets let you add children. In this sense, you can treat a dialog as a manager

widget. Motif allows you to add multiple children to an existing dialog, so you can provide additional controls, action area buttons, and even a MenuBar.

Modifying MessageDialogs

At the end of Chapter 5, we described a scenario where an application might want to have more than three action area buttons in a MessageDialog. If the user has selected the *Quit* button in a text editor but has not saved her changes, an application might want to post a dialog that asks about saving the changes before exiting. The user could want to save the changes and exit, not save the changes and exit anyway, cancel the exit operation, or get help.

The MessageDialog supports three action area buttons, so creating a dialog with four buttons requires designing a custom dialog. The MessageDialog allows you to provide additional action area buttons. Example 7-1 demonstrates how to create a QuestionDialog with four action area buttons.*

Example 7-1 The question.c program

```
/* question.c - create a QuestionDialog with four action buttons */

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

main (int argc, char *argv[])
{
    XtAppContext app;
    Widget       toplevel, pb;
    void         pushed(Widget, XtPointer, XtPointer);

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

    pb = XmCreatePushButton (toplevel, "Button", NULL, 0);
    XtAddCallback (pb, XmNactivateCallback, pushed, NULL);
    XtManageChild (pb);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

void pushed (Widget w, XtPointer client_data, XtPointer call_data)
{
    Widget       dialog, no_button;
    void         dlg_callback(Widget, XtPointer, XtPointer);
    Arg          args[5];
    int          n = 0;
    XmString     m;
    XmString     yes = XmStringCreateLocalized ("Yes");
```

* `XtVaAppInitialize`() is deprecated in X11R6.

```
    XmString     no = XmStringCreateLocalized ("No");

    m = XmStringCreateLocalized ("Do you want to update your changes?");
    XtSetArg (args[n], XmNautoUnmanage, False); n++;
    XtSetArg (args[n], XmNmessageString, m); n++;
    XtSetArg (args[n], XmNokLabelString, yes); n++;
    dialog = XmCreateQuestionDialog (w, "notice", args, n);
    XtAddCallback (dialog, XmNokCallback, dlg_callback, NULL);
    XtAddCallback (dialog, XmNcancelCallback, dlg_callback, NULL);
    XtAddCallback (dialog, XmNhelpCallback, dlg_callback, NULL);
    XmStringFree (m);
    XmStringFree (yes);
    XtSetArg(args[0], XmNlabelString, no);
    no_button = XmCreatePushButton (dialog, "no", args, 1);
    XtAddCallback (no_button, XmNactivateCallback, dlg_callback, NULL);
    XtManageChild (no_button);
    XtManageChild (dialog);
}

void dlg_callback (Widget w, XtPointer client_data, XtPointer call_data)
{
    XmAnyCallbackStruct *cbs = (XmAnyCallbackStruct *) call_data;

    switch (cbs->reason) {
        case XmCR_OK       : /* FALLTHROUGH */
        case XmCR_CANCEL   : XtUnmanageChild (w); break;
        case XmCR_ACTIVATE : XtUnmanageChild (XtParent (w)); break;
        case XmCR_HELP     : puts ("Help selected"); break;
    }
}
```

The dialog box from the program is shown in Figure 7-1.

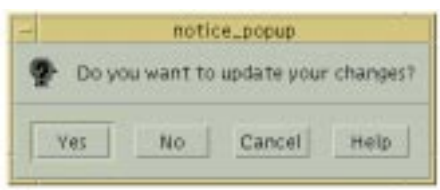


Figure 7-1 Output of the question program

The extra button is added to the dialog by creating a PushButton as a child of the dialog. We are treating the MessageDialog just like any other manager widget. The MessageDialog inserts any additional PushButton children into the action area after the *OK* button, which is why we added a *No* button. If you add more than one button, they are all put after the *OK* button, in the order that you create them. We have also changed the label of the *OK* button so that it is now the *Yes* button.

Since the *No* button is not part of the standard MessageDialog, we have to set the callback routine on its `XmNactivateCallback`. For the rest of the buttons, we use the callbacks

defined by the dialog. The dialog callback routine, `dlg_callback`(), has to handle the various callbacks in different ways. By checking the `reason` field of the callback structure, the routine can determine which button was selected. For the *Yes* and *Cancel* buttons, the routine unposts the dialog by unmanaging the MessageDialog. For the *No* button, we need to be a bit more careful about popping down the right widget. Since the `widget` in this case is the PushButton, we need to call `XtParent`() to get the MessageDialog.

The MessageDialog also supports the addition of other children besides PushButtons. If you add a MenuBar child, it is placed across the top of the dialog, although it is not clear why you would want a MenuBar in a MessageDialog. Any other type of widget child is considered the work area. The work area is placed below the message text if it exists. If there is a symbol, but no message, the work area is placed to the right of the symbol. The MessageDialog only supports the addition of one work area; the layout of multiple work area children is undefined.

The `XmNdialogType` resource can take the value `XmDIALOG_TEMPLATE`. This value creates a TemplateDialog, which is basically an empty MessageDialog that can be modified by the programmer. By default, the dialog only contains a Separator child*. By setting various resources on a TemplateDialog when it is created, you can cause the dialog to create other standard children. If you set a string or callback resource for an action area button, the button is created. If you set the `XmNmessageString` resource, the message is displayed in the standard location. If you set the `XmNsymbolPixmap` resource, the specified symbol appears in its normal location. If you don't set a particular resource, then that child is not created, which means that you cannot modify the resource later with `XtSetValues`(), set a callback for the child with `XtAddCallback`(), or retrieve the child through `XtNameToWidget`().

Modifying SelectionDialogs

The Motif SelectionDialog supports the same types of modifications as the MessageDialog. You can provide additional action area buttons, a work area child, and a MenuBar. Unlike the MessageDialog, the first widget that is added is taken as the workarea, regardless of whether it is a PushButton or a MenuBar.†If you want to add a PushButton to the action area of a SelectionDialog, you need to add a fake unmanaged work area widget first, so that the PushButton is placed in the action area, rather than used as the work area. After you add a work area, if you add a MenuBar, it is placed along the top of the dialog, and PushButton children are inserted after the *OK* button. The position of the work area child is controlled by the `XmNchildPlacement` resource, which can take the following values:

---

* There is a persistent bug such that attempting to unmanage the Separator (for whatever reason) before adding any other children to the TemplateDialog causes a segmentation fault.

† The fact that the first child is always taken to be the work area is considered a bug. As a result of the bug, you need to be careful about the order in which you add children to a SelectionDialog.

```
XmPLACE_ABOVE_SELECTION            XmPLACE_BELOW_SELECTION
XmPLACE_TOP
```

The SelectionDialog only supports the addition of one work area; the layout of multiple work area children is undefined.

Consider providing additional controls in a PromptDialog like the one used in the program *prompt_dlg* from Chapter 6, *Selection Dialogs*. In this program, the dialog prompts the user for a new label for the PushButton that activated the dialog. By adding another widget to the dialog, we can expand its functionality to prompt for either a label name or a button color. The user enters either value in the same text input area and the RadioBox controls how the text is evaluated. Example 7-2 shows the new program.*

Example 7-2 The modify_btn.c program

```
/* modify_btn.c -- demonstrate how a default Motif dialog can be
** modified to support additional items that extend the usability
** of the dialog itself. This is a modification of the prompt_dlg.c
** program.
*/

#include <Xm/SelectioB.h>
#include <Xm/RowColumn.h>
#include <Xm/PushB.h>

main (int argc, char *argv[])
{
    XtAppContext   app;
    Widget         toplevel, rc, button;
    void           pushed(Widget, XtPointer, XtPointer);

    XtSetLanguageProc (NULL, NULL, NULL);
    /* Initialize toolkit and create toplevel shell */
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv, NULL,
                                    sessionShellWidgetClass, NULL);
    /* RowColumn managed both PushButtons */
    rc = XmCreateRowColumn (toplevel, "rowcol", NULL, 0);
    /* Create two pushbuttons -- both have the same callback */
    button = XmCreatePushButton (rc, "PushMe 1", NULL, 0);
    XtAddCallback (button, XmNactivateCallback, pushed, NULL);
    XtManageChild (button);
    button = XmCreatePushButton (rc, "PushMe 2", NULL, 0);
    XtAddCallback (button, XmNactivateCallback, pushed, NULL);
    XtManageChild (button);
    XtManageChild (rc);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}
```

* XtVaAppInitialize() is considered deprecated in X11R6. XmSelectionBoxGetChild() is deprecated in Motif 2.0. The Toggle value in Motif 2.0 and later is an enumerated type, not a Boolean.XmStringGetLtoR() is deprecated from Motif 2.0: prefer XmStringUnparse().

```
/* pushed() --the callback routine for the main app's pushbuttons.
** Create a dialog that prompts for a new button name or color.
** A RadioBox is attached to the dialog. Which button is selected
** in this box is held as an int (0 or 1) in the XmNuserData resource
** of the dialog itself. This value is changed when selecting either
** of the buttons in the ToggleBox and is queried in the dialog's
** XmNokCallback function.
*/
void pushed (Widget pb, XtPointer client_data, XtPointer call_data)
{
    Widget       dialog, toggle_box;
    XmString     t, btn1, btn2;
    void         read_name(Widget, XtPointer, XtPointer);
    void         toggle_callback(Widget, XtPointer, XtPointer);
    void         destroy_callback(Widget, XtPointer, XtPointer);
    Arg          args[5];
    int          n = 0;

    /* Create the dialog -- the PushButton acts as the DialogShell's
    ** parent (not the parent of the PromptDialog). The "userData"
    ** is used to store the value
    */
    t = XmStringCreateLocalized ("Enter New Button Name:");
    XtSetArg (args[n], XmNselectionLabelString, t); n++;
    XtSetArg (args[n], XmNautoUnmanage, False); n++;
    XtSetArg (args[n], XmNuserData, 0); n++;
    dialog = XmCreatePromptDialog (pb, "notice_popup", args, n);
    XmStringFree (t); /* always destroy compound strings when done */
    /* When the user types the name, call read_name()... */
    XtAddCallback (dialog, XmNokCallback, read_name, pb);
    /* If the user selects cancel, just destroy the dialog */
    XtAddCallback (dialog, XmNcancelCallback, destroy_callback, NULL);
    /* No help is available... */
    XtUnmanageChild (XtNameToWidget (dialog, “Help”));
    /* Create a toggle box -- callback routine is toggle_callback() */
    btn1 = XmStringCreateLocalized ("Change Name");
    btn2 = XmStringCreateLocalized ("Change Color");
    toggle_box = XmVaCreateSimpleRadioBox (dialog,
                                           "radio_box", 0, toggle_callback,
                                       XmVaRADIOBUTTON, btn1, 0, NULL, NULL,
                                       XmVaRADIOBUTTON, btn2, 0, NULL, NULL,
                                         NULL);
    XtManageChild (toggle_box);
    XtManageChild (dialog);
}

/*
** This is passed the prompt when called. We destroy the shell parent.
*/
void destroy_callback (Widget prompt, XtPointer client_data,
                       XtPointer call_data)
{
    XtDestroyWidget ( XtParent (prompt));
```

```
}

/* callback for the items in the toggle box -- the "client data" i
** the item number selected. Since the function gets called whenever
** either of the buttons changes from true to false or back again,
** it will always be called in pairs -- ignore the "False" settings.
** When cbs->set is true, set the dialog's label string accordingly.
*/
void toggle_callback (Widget toggle_box, XtPointer client_data,
XtPointer call_data)
{
    Widget         dialog = XtParent (XtParent (toggle_box));
    XmString       str;
    int            n = (int) client_data;
    XmToggleButtonCallbackStruct  *cbs;

    cbs = (XmToggleButtonCallbackStruct *) call_data;

    if (cbs->set == XmUNSET)
        return; /* wait for the one that toggles "on" */
    if (n == 0)
        str = XmStringCreateLocalized ("Enter New Button Name:");
    else
        str = XmStringCreateLocalized ("Enter Text Color:");
    XtVaSetValues (dialog, XmNselectionLabelString, str,
                   /* reset the user data to reflect new value */
                   XmNuserData, n,
                   NULL);
    XmStringFree (str);
}

/* read_name() --the text field has been filled in. Get the userData
** from the dialog widget and set the PushButton's name or color.
*/
void read_name (Widget dialog, XtPointer client_data, XtPointer call_data)
{
    char       *text;
    int        n;
    Widget     push_button = (Widget) client_data;
    XmSelectionBoxCallbackStruct *cbs =
                              (XmSelectionBoxCallbackStruct *) call_data;
    /* userData: n == 0 -> Button Label, n == 1 -> Button Color */
    XtVaGetValues (dialog, XmNuserData, &n, NULL);
    if (n == 0)
         XtVaSetValues (push_button, XmNlabelString, cbs->value, NULL);
    else {
        /* convert compound string into regular text string */
        text = (char *) XmStringUnparse (cbs->value,
                                         XmFONTLIST_DEFAULT_TAG,
                                         XmCHARSET_TEXT,
                                         XmCHARSET_TEXT,
                                         NULL, 0, XmOUTPUT_ALL);
        XtVaSetValues (push_button,
                   XtVaTypedArg, XmNforeground, XmRString, text,
```

```
                    strlen (text) + 1, NULL);
            XtFree (text); /* must free text gotten from XmStringUnparse() */
        }
    }
```

The new dialog is shown in Figure 7-2.

Figure 7-2 Output of the modify_btn program

We add a RadioBox as the work area child of the PromptDialog. The ToggleButtons in the RadioBox indicate whether the input text is supposed to change the label of the PushButton or its text color. To determine which of these attributes to change, we use the callback routine `toggle_callback`().

Rather than storing the state of the RadioBox in a global variable, we store the value in the `XmNuserData` resource of the dialog widget. Using this technique, we can retrieve the value any time we wish and minimize the number of global variables in the program. The `XmNuserData` resource is available for all Motif widgets except shells, so it is a convenient storage area for arbitrary values. The type of value that `XmNuserData` takes is any type whose size is less than or equal to the size of an XtPointer, which is typically defined as a char pointer. As a result, storing an int works just fine. If you want to store a data structure in this resource, you need to store a pointer to the structure. The size or type of the structure is irrelevant, since pointers are the same size.*

When the user enters new text and presses RETURN or activates the *OK* button, `read_name`() is called. This callback routine gets the `XmNuserData` from the dialog widget. If the value is 0, the label of the PushButton is reset using the `XmNlabelString` resource. Since the callback routine provides the text in compound string format, it is already in the correct format for the label. If the `XmNuserData` is 1, then the text describes a color name for the PushButton.

* You might run into problems with unusual architectures where pointers of different types are not the same size, like DOS.

Rather than converting the string into a color explicitly, we use the XtVaTypedArg feature of XtVaSetValues() to do the conversion for us. This feature converts a value to the format needed by the specified resource. The XmNforeground resource takes a variable of type Pixel as a value. The conversion works provided there is an underlying conversion function to support it.* Motif does not supply a conversion function to change a compound string into a Pixel value, but there is one for converting a C string into a Pixel. We convert the compound string into a C string using XmStringUnparse() and then set the foreground color as follows:

```
XtVaSetValues (push_button, XtVaTypedArg, XmNforeground, XmRString, text,
                                                strlen (text) + 1, NULL);
```

So far, we've described the possibilities for both MessageDialogs and SelectionDialogs. If the layouts that are possible do not meet your needs, you should consider building your own dialogs from scratch.

Designing New Dialogs

In this section, we introduce the methods for building a dialog entirely from scratch. To create a new dialog, you need to follow basically the same steps that are used by the Motif convenience routines, which we described in Chapter 5, *Introduction to Dialogs*. We've modified the list a bit to reflect the flexibility that you have in controlling the kind of dialog that you make. Here are the steps that you need to follow:

1. Choose a shell widget that best fits the needs of your dialog. You may continue to use a DialogShell if you like.
2. Choose an appropriate manager widget to control the layout of the components of the dialog. This manager is a child of the shell widget. The manager widget you choose greatly affects how the dialog is laid out. You do not have to use a BulletinBoard or Form widget, but you can if you like.†
3. Create the control area, which may include any of the Motif primitive or manager widgets. This step is the one that gives you the most flexibility, as you have complete control over the contents and layout of the control area.
4. Create an action area with PushButtons such as *OK*, *Cancel*, and *Help*. Since you are creating the control area yourself, you cannot use XmNokCallback and the other resources specific to the predefined Motif dialogs. Instead, you use the callback resources appropriate for the widgets that you use in the dialog.

---

* For more information on conversion functions, how to write them, or how to install your own, see Volume 4, *X Toolkit Intrinsics Programming Manual*.

† If you do want to use a DialogShell with either a Form or a BulletinBoard widget as the manager, you can use one of the Motif convenience routines: XmCreateBulletinBoardDialog() or XmCreateFormDialog(). These routines give you a starting point for creating a custom dialog. However, in this chapter, we create each of the widgets explicitly, so that you have a complete sense of what goes into a dialog.

5. Pop up the shell created in the first step.

The Shell

In Chapter 4, *The Main Window*, we demonstrated the purpose of a main window in an application and the kinds of widgets that you use in a top-level window. Dialog boxes, as introduced in Chapter 5, are thought of as transient windows that act as satellites to a top-level shell. A transient dialog should use a DialogShell widget. However, not all dialogs are transient. A dialog may act as a secondary application window that remains on display for an extended period of time. This usage is especially common in large applications. The MainWindow widget can even be used in a dialog box*. For dialogs of this type, you may want to use a TopLevelShell, or a SessionShell.†

Choosing the appropriate shell widget for a dialog depends on the activities carried out in the dialog, so it is difficult to provide rules or even heuristics to guide you in your choice. As discussed in Chapter 5, a DialogShell cannot be iconified, it is always placed on top of the shell widget that owns the parent of the dialog, and it is always destroyed or withdrawn from the screen if its parent is destroyed or withdrawn. These three characteristics may influence your decision to use a DialogShell. A SessionShell or a TopLevelShell, on the other hand, is always independent of other windows, so you can change its stacking order, iconify it separately, and not worry about it being withdrawn because of another widget. The main difference between an SessionShell and a TopLevelShell is that a SessionShell is designed to start a completely new widget tree, as if it were a completely separate application. It is recommended that an application only have one SessionShell.‡

For some applications, you may want a shell with characteristics of several of the available shell classes. Unfortunately, it is difficult to intermix the capabilities of a DialogShell with those of a SessionShell or a TopLevelShell because it involves doing quite a bit of intricate window manager interaction. Having ultimate control over the activities of a shell widget requires setting up a number of event handlers on the shell and monitoring certain window property event state changes. Aside from being very complicated, you run the risk of breaking Motif compliance. See Chapter 20, *Interacting With the Window Manager*, for details on how you might handle this situation.

Once you have chosen the shell widget that you want to use, you need to decide how to create it. A DialogShell can be created using the routines `XtCreatePopupShell`() or

* Creating multiple MainWindow widgets in a single application has some problems associated with it, and has not found to be entirely robust.

† In X11R6, the simple ApplicationShell is considered deprecated: you should use the SessionShell in its place. SessionShell is derived from ApplicationShell. Programs using an ApplicationShell will still work, although their participation in X11R6 Session Management is limited.

‡ Multiple Application Shells (or derived classes) in a single application can be difficult to handle correctly: all sorts of focus issues can arise. There is poor guidance on the use of multiple ApplicationShell widgets in any case. Where the application shells occupy multiple screens, the issues are much less problematic.

XtVaCreatePopupShell(), or the Motif toolkit convenience routine, XmCreateDialogShell(). A SessionShell or a TopLevelShell can be created using either of the popup shell routines, XtAppCreateShell() or XtVaAppCreateShell(). The difference between the two types of routines involves whether the newly-created shell is treated like a popup shell or as a more permanent window on the desktop. If you create the shell as a popup shell, you need to select an adequate parent. The parent for a popup shell must be an initialized and realized widget. It can be any kind of widget, but it may not be a gadget because the parent must have a window. A dialog that uses a popup shell inherits certain attributes from its parent. For example, if the parent is insensitive (XmNsensitive is set to False), the entire dialog is insensitive as well.

The Manager Child

The manager widget that you choose for a dialog is the only managed child of the shell widget, which means that the widget must contain both the control area and the action area of the dialog and manage the relationship between them. Recall that the *Motif Style Guide* suggests that a dialog be composed of two main areas: the control area and the action area. Both of these areas extend to the left and right sides of a dialog and are stacked vertically, with the control area on the top. The action area usually does not fluctuate in size as the shell is resized, while the control area may be resized in any way. Figure 7-3 illustrates the general layout of a dialog.

Figure 7-3 Layout of a dialog

Motif dialog widgets handle this layout automatically. When you create your own dialog, you are responsible for managing the layout. You could consider using the PanedWindow widget as the manager widget for a dialog. The PanedWindow supports both vertically and horizontally stacked windows*, each of which may or may not be resizable, which allows you to create the suggested layout. If you use a PanedWindow as the manager widget for a dialog, it can manage other managers that act as the control and action areas. The control area can be resizable, while the action area is not. The PanedWindow also provides a

* Horizontal orientation for the PanedWindow was officially supported in Motif 2.0.

separator between the panes, which fulfills the *Style Guide* recommendation that there be a Separator widget between the control and action areas.

Of course you can use whatever manager widget you like for a dialog. If you use a BulletinBoard or a Form widget, you may be able to take advantage of the special interaction these widgets have with a DialogShell. The RowColumn widget can also lay out its children vertically, so you could use one to manage the control and action areas of a dialog. The difficulty with using a RowColumn involves resizing, in that there is no way to tell the widget to keep the bottom partition a constant height while allowing the top to fluctuate as necessary. The same problem can also arise with other manager widgets, so you need to be sure that the resizing behavior is appropriate.

The Control Area

The control area of a dialog box contains the widgets that provide the functionality of the dialog, such as Labels, ToggleButtons, and List widgets. Creating the control area of a dialog is entirely application-defined. You can use any of the Motif primitive and manager widgets in the control area to implement the functionality of the dialog. The ability to design your own control area is the main reason to create your own dialog as opposed to using one of the predefined Motif dialogs.

The Action Area

The action area of a dialog contains PushButtons whose callback routines actually perform the action of the dialog box. Constructing the action area for a dialog involves specifying labels and callback routines for the buttons and determining the best way to get information from the control area of the dialog. The *Motif Style Guide* defines a number of common dialog box actions. The common actions are designed to provide consistency between different Motif applications. You are not required to use the common actions, but you should consider them before creating your own arbitrary actions. The button labels and their corresponding actions are shown in the following list.

Yes
: Indicates an affirmative response and causes the dialog to be dismissed.

No
: Indicates a negative response and causes the dialog to be dismissed.

OK
: Applies any changes reflected in the control area, performs any related actions, and causes the dialog box to be dismissed.

Close
: Closes the dialog box without performing any action.

Apply
: Applies any changes reflected in the control area, performs any related actions, and leaves the dialog open for further interaction.

Retry
: Tries the task in progress again. This action is commonly found in dialog boxes that report errors.

Stop
: Stops the task in progress at the next possible breaking point. This action is often found in dialog boxes that indicate that the application is "busy."

Pause
: Pauses the task in progress. This action is used in combination with *Resume*.

Resume
: Resumes the task in progress. This action is used in combination with *Pause*.

Reset
: Resets the controls in the work area to the values they had at the time the dialog was originally opened.

Cancel
: Resets the controls in the work area and causes the dialog to be dismissed.

Help
: Provides help for the dialog box.

The following heuristics can help in designing the action area for a dialog box:

- Lay out the action area as a single horizontal row at the bottom of the dialog.
- Set the action area apart from the rest of the dialog using a Separator.
- Use single-word button labels.
- Choose command-style verbs over nouns when possible. Since some words can be interpreted in more than one way, be careful to avoid ambiguity.
- Affirmative actions should be placed farthest to the left (in a left-to-right language environment), followed by negative actions, followed by cancelling actions. For example, *Yes* should always be placed to the left of *No*.
- *Help*, if available, should always be placed farthest to the right (in a left-to-right language environment).

Depending on your application, you may want to create your own actions and overlook some of these guidelines. Figure 7-4 shows a custom dialog that demonstrates some of the issues involved in designing an action area.

Figure 7-4 A custom dialog

In this dialog, the *Help* and *Close* buttons are the only ones with a label recommended by Motif. Since the other common actions did not effectively represent the actions of the dialog, we chose our own labels. The *Find...* and *Search list...* buttons popup further dialogs without closing the window.

We do not use the *OK* action in the dialog because it doesn't work with the desired usage of the dialog. By definition, *OK* should perform the action and dismiss the dialog. Here we have in effect two actions: perform a search according to the current criteria, and popup the results of previous searches. In neither case is the current dialog dismissed. It was felt that neither *OK* nor *Apply* were appropriate in these circumstances.

Building a Dialog

Now that we've explained the design process for a dialog, let's create a real dialog and identify each of the steps in the process. Consider the problem of providing help. While the Motif InformationDialog is adequate for brief help messages, a customized dialog may be more appropriate for displaying large amounts of text. Our custom dialog displays the text in a scrolling region which is capable of handling arbitrarily large amounts of data.

Example 7-3 shows a program that uses a main application window as a generic backdrop. The MainWindow widget contains a MenuBar that has two menus: *File* and *Help*. The *Help* menu contains several items that, when selected, pop up a dialog window that displays the

associated help text. The text that we provide happens to be predefined in the program, but you could incorporate information from other sources, such as a database or an external file.*

Example 7-3 The help_text.c program

```
/* help_text.c:
** Create a simple main window that contains a sample (dummy) work
** area and a menubar. The menubar contains two items: File and Help.
** The items in the Help pulldown call help_cb(), which pops up a
** home-made dialog that displays predefined help texts. The purpose
** of this program is to demonstrate how one might approach the
** problem displaying a large amount of text in a dialog box.
*/
#include <stdio.h>
#include <ctype.h>
#include <Xm/DialogS.h>
#include <Xm/MainW.h>
#include <Xm/RowColumn.h>
#include <Xm/Form.h>
#include <Xm/Text.h>
#include <Xm/PushBG.h>
#include <Xm/LabelG.h>
#include <Xm/PanedW.h>

/* The following help text information is a continuous stream of characters
** that will all be placed in a single ScrolledText object. If a specific
** newline is desired, you must do that yourself. See "index_help" below.
*/
String context_help[] = {
    "This is context-sensitive help. Well, not really, but such",
    "help text could easily be generated by a real help system.",
    "All you really need to do is obtain information from the user",
    "about the widget from which he needs help, or perhaps prompt",
    "for other application-specific contexts.",
    NULL};

String window_help[] = {
    "Each of the windows in your application should have an",
    "XmNhelpCallback associated with it so you can monitor when",
    "the user presses the Help key over any particular widget.",
    "This is another way to provide context-sensitive help.",
    "The MenuBar should always have a Help entry at the far right",
    "that provides help for most aspects of the program, including",
    "the user interface. By providing different levels of help",
    "indexing, you can provide multiple stages of help, making the",
    "entire help system easier to use.",
    NULL};

String index_help[] = {
    "This is a small demonstration program, so there is very little",
```

* XtVaAppInitialize() is deprecated in X11R6.

```
    "material to provide an index. However, an index should contain",
    "a summary of the type of help available. For example, we have:\n",
    "   Help On Context\n",
    "   Help On Windows\n",
    "   This Index\n",
    "\n",
    "Higher-end applications might also provide a tutorial.",
    NULL};

String *help_texts[] = {context_help, window_help, index_help};

main (int argc, char *argv[])
{
    XtAppContext       app;
    Widget             toplevel, rc, main_w, menubar, w, label_g;
    void               help_cb(Widget, XtPointer, XtPointer);
    void               file_cb(Widget, XtPointer, XtPointer);
    XmString           str1, str2, str3;
    Widget             *cascade_btns;
    int                num_btns;
    Arg                args[2];

    XtSetLanguageProc (NULL, NULL, NULL);
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv, NULL,
                                    sessionShellWidgetClass, NULL);
    /* the main window contains the work area and the menubar */
    main_w = XmCreateMainWindow (toplevel, "main_w", NULL, 0);
    /* Create a simple MenuBar that contains two cascade buttons */
    str1 = XmStringCreateLocalized ("File");
    str2 = XmStringCreateLocalized ("Help");
    menubar = XmVaCreateSimpleMenuBar (main_w, "main_w",
                                       XmVaCASCADEBUTTON, str1, 'F',
                                       XmVaCASCADEBUTTON, str2, 'H',
                                       NULL);
    XmStringFree (str1);
    XmStringFree (str2);
    /* create the "File" pulldown menu -- callback is file_cb() */
    str1 = XmStringCreateLocalized ("New");
    str2 = XmStringCreateLocalized ("Open");
    str3 = XmStringCreateLocalized ("Quit");
    XmVaCreateSimplePulldownMenu (menubar, "file_menu", 0, file_cb,
                                  XmVaPUSHBUTTON, str1, 'N', NULL, NULL,
                                  XmVaPUSHBUTTON, str2, 'O', NULL, NULL,
                                  XmVaSEPARATOR,
                                  XmVaPUSHBUTTON, str3, 'Q', NULL, NULL,
                                  NULL);
    XmStringFree (str1);
    XmStringFree (str2);
    XmStringFree (str3);
    /* create the "Help" menu -- callback is help_cb() */
    str1 = XmStringCreateLocalized ("On Context");
    str2 = XmStringCreateLocalized ("On Window");
    str3 = XmStringCreateLocalized ("Index");
    w = XmVaCreateSimplePulldownMenu (menubar, "help_menu", 1, help_cb,
```

```
                                        XmVaPUSHBUTTON, str1, 'C', NULL, NULL,
                                        XmVaPUSHBUTTON, str2, 'W', NULL, NULL,
                                        XmVaPUSHBUTTON, str3, 'I', NULL, NULL,
                                        NULL);
    XmStringFree (str1);
    XmStringFree (str2);
    XmStringFree (str3);
    /* Identify the Help Menu for the MenuBar */
    XtVaGetValues (menubar, XmNchildren, &cascade_btns, XmNnumChildren,
                    &num_btns, NULL);
    XtVaSetValues (menubar, XmNmenuHelpWidget, cascade_btns[num_btns-1],
                   NULL);
    XtManageChild (menubar);
    /* the work area for the main window -- just create dummy stuff */
    rc = XmCreateRowColumn (main_w, "rc", NULL, 0);
    str1 = XmStringCreateLocalized (
                                  "\n This is an Empty\nSample Control Area\n");
    XtSetArg (args[0], XmNlabelString, str1);
    label_g = XmCreateLabelGadget (rc, "label", args, 1);
    XtManageChild (label_g);
    XmStringFree (str1);
    XtManageChild (rc);
    XtManageChild (main_w);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

/* callback for all the entries in the File pulldown menu. */
void file_cb (Widget w, XtPointer client_data, XtPointer call_data)
{
    int item_no = (int) client_data;
    if (item_no == 2)
        /* the Quit menu button */
        exit (0);
    printf ("Item %d (%s) selected\n", item_no + 1, XtName (w));
}

/* climb widget tree until we get to the top. Return the Shell */
Widget GetTopShell (Widget w)
{
    while (w && !XtIsWMShell (w))
    w = XtParent (w);
    return w;
}

#include "info.xbm"
/* bitmap data used by our dialog */
/* callback for all the entries in the Help pulldown menu.
** Create a dialog box that contains control and action areas.
*/
void help_cb (Widget w, XtPointer client_data, XtPointer call_data)
{
    Widget       help_dialog, pane, text_w, form, sep, widget, label;
    void         DestroyShell(Widget, XtPointer, XtPointer);
```

```
Pixmap       pixmap;
Pixel        fg, bg;
Arg          args[10];
int          n = 0;
int          i;
char         *p, buf[BUFSIZ];
int          item_no = (int) client_data;
Dimension    h;
/* Set up a DialogShell as a popup window. Set the delete
** window protocol response to XmDESTROY to make sure that
** the window goes away appropriately. Otherwise, it's XmUNMAP
** which means it'd be lost forever, since we're not storing
** the widget globally or statically to this function.
*/
i = 0;
XtSetArg (args[i], XmNdeleteResponse, XmDESTROY); i++
help_dialog = XmCreateDialogShell ( GetTopShell(w), "Help", args, i);

/* Create a PanedWindow to manage the stuff in this dialog. */
/* PanedWindow won't let us set these to 0! */
XtSetArg (args[0], XmNsashWidth, 1);
/* Make small so user doesn't try to resize */
XtSetArg (args[1], XmNsashHeight, 1);
pane = XmCreatePanedWindow (help_dialog, "pane", args, 2);
/* Create a RowColumn in the form for Label and Text widgets.
** This is the control area.
*/
form = XmCreateForm (pane, "form1", NULL, 0);
XtVaGetValues (form, /* once created, we can get its colors */
               XmNforeground, &fg,
               XmNbackground, &bg,
               NULL);
/* create the pixmap of the appropriate depth using the colors
** that will be used by the parent (form).
*/
pixmap = XCreatePixmapFromBitmapData (XtDisplay (form),
RootWindowOfScreen (XtScreen (form)),
                    (char *) info_bits, info_width, info_height,
                    fg, bg,
                    DefaultDepthOfScreen (XtScreen (form)));
/* Create a label gadget using this pixmap */
n = 0;
XtSetArg (args[n], XmNlabelType,        XmPIXMAP);      n++;
XtSetArg (args[n], XmNlabelPixmap,      pixmap);        n++;
XtSetArg (args[n], XmNleftAttachment,   XmATTACH_FORM); n++;
XtSetArg (args[n], XmNtopAttachment,    XmATTACH_FORM); n++;
XtSetArg (args[n], XmNbottomAttachment, XmATTACH_FORM); n++;
label = XmCreateLabelGadget (form, "label", args, n);
XtManageChild (label);
/* prepare the text for display in the ScrolledText object
** we are about to create.
*/
for (p = buf, i = 0; help_texts[item_no][i]; i++) {
    p += strlen (strcpy (p, help_texts[item_no][i]));
```

```
    if (!isspace (p[-1]))
    /* spaces, tabs and newlines are spaces. */
        *p++ = ' '; /* lines are concatenated together, insert space */
}
*--p = 0; /* get rid of trailing space... */
n = 0;
XtSetArg (args[n], XmNscrollVertical,        True);              n++;
XtSetArg (args[n], XmNscrollHorizontal,      False);             n++;
XtSetArg (args[n], XmNeditMode,              XmMULTI_LINE_EDIT); n++;
XtSetArg (args[n], XmNeditable,              False);             n++;
XtSetArg (args[n], XmNcursorPositionVisible, False);             n++;
XtSetArg (args[n], XmNwordWrap,              True);              n++;
XtSetArg (args[n], XmNvalue,                 buf);               n++;
XtSetArg (args[n], XmNrows,                  5);                 n++;
text_w = XmCreateScrolledText (form, "help_text", args, n);
/* Attachment values must be set on the Text widget's PARENT,
** the ScrolledWindow. This is the object that is positioned.
*/
XtVaSetValues (XtParent (text_w),
               XmNleftAttachment,   XmATTACH_WIDGET,
               XmNleftWidget,       label,
               XmNtopAttachment,    XmATTACH_FORM,
               XmNrightAttachment,  XmATTACH_FORM,
               XmNbottomAttachment, XmATTACH_FORM,
               NULL);
XtManageChild (text_w);
XtManageChild (form);
/* Create another form to act as the action area for the dialog */
XtSetArg (args[0], XmNfractionBase, 5);
form = XmCreateForm (pane, "form2", args, 1);
/* The OK button is under the pane's separator and is
** attached to the left edge of the form. It spreads from
** position 0 to 1 along the bottom (the form is split into
** 5 separate grids via XmNfractionBase upon creation).
*/
widget = XmCreatePushButtonGadget (form, "OK", NULL, 0);
XtVaSetValues (widget,
               XmNtopAttachment,                XmATTACH_FORM,
               XmNbottomAttachment,             XmATTACH_FORM,
               XmNleftAttachment,               XmATTACH_POSITION,
               XmNleftPosition,                 1,
               XmNrightAttachment,              XmATTACH_POSITION,
               XmNrightPosition,                2,
               XmNshowAsDefault,                True,
               XmNdefaultButtonShadowThickness, 1,
               NULL);
XtManageChild (widget);
XtAddCallback (widget, XmNactivateCallback, DestroyShell,
               (XtPointer) help_dialog);
/* This is created with its XmNsensitive resource set to False
** because we don't support "more" help. However, this is the
** place to attach it to if there were any more.
*/
widget = XmCreatePushButtonGadget (form, "More", NULL, 0);
```

```
        XtVaSetValues (widget,
                       XmNsensitive,                   False,
                       XmNtopAttachment,               XmATTACH_FORM,
                       XmNbottomAttachment,            XmATTACH_FORM,
                       XmNleftAttachment,              XmATTACH_POSITION,
                       XmNleftPosition,                3,
                       XmNrightAttachment,             XmATTACH_POSITION,
                       XmNrightPosition,               4,
                       XmNshowAsDefault,               False,
                       XmNdefaultButtonShadowThickness, 1,
                       NULL);
        XtManageChild (widget);
        /* Fix the action area pane to its current height -- never let it resize */
        XtManageChild (form);
        XtVaGetValues (widget, XmNheight, &h, NULL);
        XtVaSetValues (form, XmNpaneMaximum, h, XmNpaneMinimum, h, NULL);
        /* This also pops up the dialog, as it is the child of a DialogShell */
        XtManageChild (pane);
    }

    /* The callback function for the "OK" button. Since this is not a
    ** predefined Motif dialog, the "widget" parameter is not the dialog
    ** itself. That is only done by Motif dialog callbacks. Here in the
    ** real world, the callback routine is called directly by the widget
    ** that was invoked. Thus, we must pass the dialog as the client
    ** data to get its handle. (We could get it using GetTopShell(),
    ** but this way is quicker, since it's immediately available.)
    */
    void DestroyShell (Widget widget, XtPointer client_data,
                       XtPointer call_data)
    {
        Widget shell = (Widget) client_data;
        XtDestroyWidget (shell);
    }
```

The output of the program is shown in Figure 7-5.

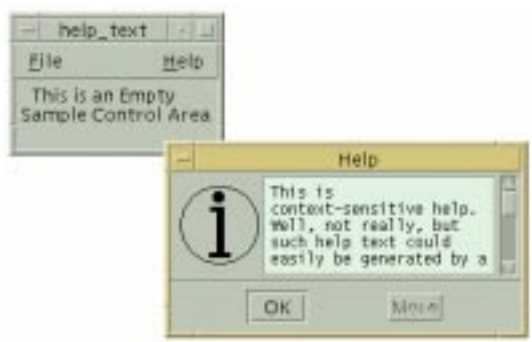


Figure 7-5 Output of help_text program

The function help_cb() is the callback routine that is invoked by all of the *Help* menu items. This routine follows the steps that we outlined earlier to create the dialog box.

The Shell

Since the dialog is a transient dialog, we use a DialogShell widget for the shell. We create the shell as follows:

```
i = 0;
XtSetArg (args[i], XmNdeleteResponse, XmDESTROY); i++;
help_dialog = XmCreateDialogShell (GetTopShell (w), "Help", args, i);
```

Instead of using the Motif convenience function, we could have used XtVaCreatePopupShell(), instead, as shown in the following code fragment:

```
help_dialog = XtVaCreatePopupShell ("Help", xmDialogShellWidgetClass,
                        GetTopShell (w), XmNdeleteResponse, XmDESTROY, NULL);
```

Both methods return a DialogShell. The XmNdeleteResponse resource is set to XmDESTROY because we want the *Close* item from the window menu in the window manager's titlebar for the shell to destroy the shell and its children. The default value for this resource is XmUNMAP; had we wanted to reuse the same dialog upon each invocation, we would have used XmUNMAP and retained a handle to the dialog widget.

The name of the dialog is *Help*, since that is the first parameter in the call to XtVaCreatePopupShell(). Resource specifications in a resource file that pertain to this dialog should use *Help* as the widget name, as shown below:

```
*Help*foreground: green
```

The string displayed in the title bar of a dialog defaults to the name of the dialog. Since the name of the dialog is *Help*, the title defaults to the same value. However, this method of setting the title does not prevent the value from being changed by the user in a resource file. For example, the following specification changes the title:

```
*Help.title: Help Dialog
```

The title can also be set using the XmNtitle resource, as shown in the following code fragments:*

```
help_dialog = XtVaCreatePopupShell ("Help", xmDialogShellWidgetClass, parent,
                                    XmNtitle, "Help Dialog", NULL);

i = 0;
XtSetArg (args[i], XmNtitle, "Help Dialog"); i++;
help_dialog = XmCreateDialogShell (parent, "Help", args, i);
```

* XmNtitle is defined identically to XtNtitle, which is an Xt resource, which means that the value is a regular character string, not a compound string.

When the title is hard-coded in the application, any resource specifications in a resource file are ignored.

The Manager Child

The next task is to create a manager widget that acts as the sole child of the DialogShell, since shell widgets can have only one managed child. This section deals heavily with manager widget issues, so if you have problems keeping up, you should look ahead to Chapter 8, *Manager Widgets*. However, the main point of the section is to provide enough context for you to understand Example 7-3. We are using a PanedWindow widget as the child of the DialogShell. The PanedWindow is created as follows:

```
...
XtSetArg (args[0], XmNsashWidth, 1);
XtSetArg (args[1], XmNsashHeight, 1);
pane = XmCreatePanedWindow (help_dialog, "pane", help_dialog, args, 2);
```

The PanedWindow manages two Form widget children, one each for the control area and the action area. These children are also called the PanedWindow's panes. Normally, in a PanedWindow, the user can resize the panes by moving the control sashes that are placed between the panes. Because the action area is not supposed to grow or shrink in size, we don't want to allow the user to adjust the sizes of the panes. There are really two issues involved here: the user might try to resize the panes individually or she might resize the entire dialog, which would cause the PanedWindow itself to resize them.

You can prevent the PanedWindow from resizing the action area when it is itself resized by setting the pane constraint resource `XmNskipAdjust` to True. However, this technique still allows the user to resize the individual panes, which means that you need to disable the control sashes. The best way to prevent both undesirable resize possibilities is to set the action area pane constraints maximum and minimum allowed heights to the same value. These settings should cause the PanedWindow to disable the sashes for that particular pane, but due to a bug in the PanedWindow widget class, the sashes are rarely disabled. To compensate, we try to make the sashes invisible by setting their sizes to a minimum value. Unfortunately, the PanedWindow won't let you set the size of a sash to 0 (a design error), so we set the values for `XmNsashWidth` and `XmNsashHeight` to 1.*

The PanedWindow widget is created unmanaged using `XmCreatePanedWindow`(). As pointed out in Chapter 8, manager widgets should not be managed until all of their children have been created and managed. Using this order allows the children's desired sizes and positions to be specified before the manager widget tries to negotiate other sizes and positions.

* The only other problem that might arise is that keyboard traversal still allows the user to reach the sashes, so you may want to remove them from the traversal list by setting their `XmNtraversalOn` resources to `False`. This issue is described in detail in Chapter 8, *Manager Widgets*.

The Control Area

The Form widget is the control area, so it is created as a child of the PanedWindow, as shown in the following fragment:

```
form = XmCreateForm (pane, "form1", NULL, 0);
```

As far as the PanedWindow is concerned, the Form widget is a single child whose width is stretched to the left and right edges of the shell. Within the Form, we add two widgets: a Label widget that contains the help pixmap and a ScrolledText for the help information.

In order to create the Label, we must first create the pixmap it is going to use. The following code fragment shows how we create the pixmap and then create the Label:

```
XtVaGetValues (form, XmNforeground, &fg, XmNbackground, &bg, NULL);
pixmap = XCreatePixmapFromBitmapData (XtDisplay (form),
                                      RootWindowOfScreen (XtScreen (form)),
                                      bitmap_bits,
                                      bitmap_width,
                                      bitmap_height,
                                      fg,
                                      bg,
                                     DefaultDepthOfScreen (XtScreen (form)));
n = 0;
XtSetArg (args[n], XmNlabelType, XmPIXMAP);             n++;
XtSetArg (args[n], XmNlabelPixmap, pixmap);             n++;
XtSetArg (args[n], XmNleftAttachment, XmATTACH_FORM);   n++;
XtSetArg (args[n], XmNtopAttachment, XmATTACH_FORM);    n++;
XtSetArg (args[n], XmNbottomAttachment, XmATTACH_FORM); n++;
label = XmCreateLabelGadget (form, "label", args, n);
XtManageChild (label);
```

We cannot create the pixmap until we know the foreground and background colors, so we retrieve these colors from the Form, since it has a valid window and colormap. This approach works for either monochrome or color screens. We use these values as the foreground and background for the pixmap we create in the call to `XCreatePixmapFromBitmapData`().* The bits for the bitmap, the width, and the height are predefined in the X bitmap file included earlier in the program (*info.xbm*). The Label uses the pixmap by setting the `XmNlabelType` and `XmNlabelPixmap` resources (see Chapter 12, *Labels and Buttons*, for more information on these resources).

The attachment resources we specified for the Label are constraint resources for the Form widget that describe how the Form should lay out its children. These constraint resources are ignored by the Label widget itself. See Chapter 8, *Manager Widgets*, for a complete description of how constraint resources are handled by widgets. In this case, the top,

* We could have used `XmGetPixmap`() to create a pixmap, but this routine does not allow us to load a pixmap directly from bitmap data, as we have done here. For us to use `XmGetPixmap`(), the file that contains the bitmap data would have to exist at run-time, or we would have to load the bitmap data directly into a static `XImage`. For more information on this technique, see Chapter 3, *Overview of the Motif Toolkit*

bottom, and left sides of the Label are all attached to the edge of the Form, which causes the Label to position itself relative to the Form.

Next, we create a ScrolledText compound object to display the help text, as shown in the following fragment:

```
n = 0;
XtSetArg (args[n], XmNscrollVertical,        True);              n++;
XtSetArg (args[n], XmNscrollHorizontal,      False);             n++;
XtSetArg (args[n], XmNeditMode,              XmMULTI_LINE_EDIT); n++;
XtSetArg (args[n], XmNeditable,              False);             n++;
XtSetArg (args[n], XmNcursorPositionVisible, False);             n++;
XtSetArg (args[n], XmNwordWrap,              True);              n++;
XtSetArg (args[n], XmNvalue,                 buf);               n++;
XtSetArg (args[n], XmNrows,                  5);                 n++;
text_w = XmCreateScrolledText (form, "help_text", args, n);
XtVaSetValues (XtParent (text_w),
               XmNleftAttachment,   XmATTACH_WIDGET,
               XmNleftWidget,       label,
               XmNtopAttachment,    XmATTACH_FORM,
               XmNrightAttachment,  XmATTACH_FORM,
               XmNbottomAttachment, XmATTACH_FORM,
               NULL);
XtManageChild (text_w);
```

In order to use `XmCreateScrolledText`(), we must use the old-style `XtSetArg`() method of setting the resources that are passed to the function. The routine actually creates two widgets that appear to be a single interface object. A ScrolledWindow widget and a Text widget are created so that the Text widget is a child of the ScrolledWindow. The toolkit returns a handle to the Text widget, but since the ScrolledWindow widget is the direct child of the Form, we set the constraint resources on the ScrolledWindow, not the Text widget. The top, right, and bottom sides of the ScrolledWindow are attached to the Form, while the left side is attached to the Label widget, so that the two widgets are always positioned next to each other.

We could have passed these resource/value pairs in the `args` list, but then the resources would have been set on both the ScrolledWindow widget and the Text widget. Since the attachment constraints would be ignored by the Text widget, there would be no real harm in setting them on both widgets. However, it is better programming style to set the resources directly on the ScrolledWindow. Details on the Text widget and the ScrolledText object can be found in Chapter 18, *Text Widgets*. Chapter 10, *ScrolledWindows and ScrollBars* discusses the ScrolledWindow widget and its resources.

The text for the widget is set using the `XmNvalue` resource. The value for this resource is the appropriate help text taken from the help_texts array declared at the beginning of the program. We set the `XmNeditable` resource to False so that the user cannot edit the help text.

The Text and Label widgets are the only two items in the Form widget. Once these children are created and managed, the Form can be managed using `XtManageChild`().

The Action Area

At this point, the control area of the dialog has been created, so it is time to create the action area. In our example, the action area is pretty simple, as the only action needed is to close the dialog. We use the *OK* button for this action. For completeness, we have also provided a *More* button to support additional or extended help. Since we don't provide any additional help, we set this button insensitive (although you can extend this example by providing it).

The action area does not have to be contained in a separate widget, although it is generally much easier to do so. We use a Form widget in order to position the buttons evenly across the width of the dialog. We create the Form as follows:

```
XtSetArg (args[0], XmNfractionBase, 5);
form = XmCreateForm (pane, "form2", args, 1);
```

The `XmNfractionBase` resource of the Form widget is set to five, so that the Form is broken down into five equal units, as shown in Figure 7-6.

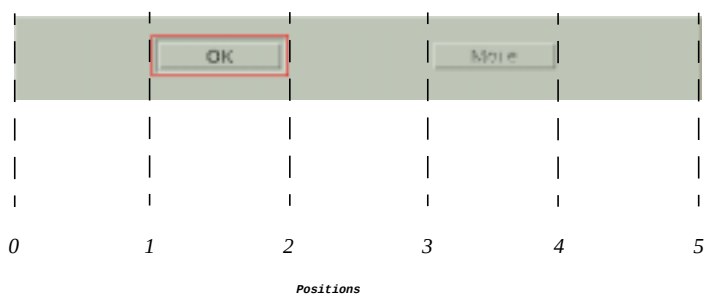


Figure 7-6 The XmNfractionBase resource divides the form into equal units

Position zero is the left edge of the form and position five is the right edge of the form. We chose five because it gave us the best layout aesthetically. The region is divided up equally, so you can think of the sections as percentages of the total width (or height) of the Form. By using this layout method, we don't have to be concerned with the width of the Form or of the DialogShell itself, since we know that the placement of the buttons will always be proportional. We create the *OK* button as shown in the following code fragment:

```
widget = XmCreatePushButtonGadget (form, "OK", NULL, 0);
XtVaSetValues ( widget,
                XmNtopAttachment,               XmATTACH_FORM,
                XmNbottomAttachment,            XmATTACH_FORM,
                XmNleftAttachment,              XmATTACH_POSITION,
                XmNleftPosition,                1,
                XmNrightAttachment,             XmATTACH_POSITION,
                XmNrightPosition,               2,
```

```
                XmNshowAsDefault,               True,
                XmNdefaultButtonShadowThickness, 1,
                NULL);
```

The left and right sides of the button are placed at positions one and two, respectively. Since this button is the default button for the dialog, we want the button to be displayed as such. We set XmNshowAsDefault to True, and XmNdefaultButtonShadowThickness to 1. The value for the XmNdefaultButtonShadowThickness resource is a pixel value that specifies the width of an extra three-dimensional border that is placed around the default button to distinguish it from the other buttons. An alternative to specifying the XmNshowAsDefault resource is to set the XmNdefaultButton resource on the containing Form to the value of the PushButton widget*. For example,

```
XtVaSetValues (form, XmNdefaultButton, push_button, NULL);
```

If XmNshowAsDefault is False, the button is not shown as the default, regardless of the value of any default shadow thickness.†

Because the dialog is not reused, we want the callback for the *OK* button to destroy the DialogShell. We use the XmNactivateCallback of the PushButton to implement the functionality. The callback routine is DestroyShell(), which is shown below:

```
void DestroyShell (Widget widget, XtPointer client_data, XtPointer call_data)
{
    Widget shell = (Widget) client_data;
    XtDestroyWidget (shell);
}
```

Since the dialog is not a predefined Motif dialog, the widget parameter to the callback routine is not the dialog, but the PushButton that caused the callback to be invoked. This difference is subtle and it is often overlooked by programmers who are breaking away from the predefined dialogs to build their own dialogs. We pass the DialogShell, help_dialog, as client data to the callback routine, so that the callback can destroy the widget.

The *More* button is not used in the application, since we do not provide any additional help for the dialog. We create the button as follows:

```
widget = XmCreatePushButtonGadget (form, "More", NULL, 0);
XtVaSetValues (widget,
               XmNsensitive,                    False,
               XmNtopAttachment,                XmATTACH_FORM,
               XmNbottomAttachment,             XmATTACH_FORM,
               XmNleftAttachment,               XmATTACH_POSITION,
               XmNleftPosition,                 3,
```

* Setting the XmNdefaultButton resource is generally to be preferred, although if the Form concerned is not the child of the containing Shell, the Button may not initially show the required visuals until it gains the keyboard focus.

† The XmNshowAsDefault resource can also take a numeric value that indicates the shadow thickness to use, but its value is only interpreted in this way if XmNdefaultButtonShadowThickness is set to 0. This functionality is for backwards compatibility with Motif 1.0 and should not be used.

```
XmNrightAttachment,             XmATTACH_POSITION,
XmNrightPosition,               4,
XmNshowAsDefault,               False,
XmNdefaultButtonShadowThickness, 1,
NULL);
```

In this case, the XmNshowAsDefault resource is set to False. We have also set XmNsensitive to False so that the widget is insensitive to user input.

Once the buttons in the action area have been created, we need to fix the size of the action area. We manage the Form and then we retrieve the height of one of the action area buttons, so that we can use the value as the minimum and maximum height of the pane. We set the XmNpaneMaximum and XmNpaneMinimum constraint resources on the Form, so that the PanedWindow sets the action area to a constant height.

Once the control area and the action area have been created and managed, the PanedWindow is managed using XtManageChild(), which has the side effect of popping up the parent DialogShell*. See Chapter 5, *Introduction to Dialogs* for a complete discussion of the posting of dialogs.

Generalizing the Action Area

While dialogs can vary in many respects, the structure of the action area usually remains consistent for all dialogs. Most large programs are going to make use of many customized dialogs. In the general case, you do not want to rewrite the code to generate an action area for each special case. It is much easier and more efficient to write a generalized routine that creates an action area for any dialog.

Whenever we generalize any procedure, we first identify how the situation may change from one case to the next. For example, not all action areas have only two buttons; you may have any number from one to, say, ten. As a result, you need to be able to change the number of partitions in the Form widget to an arbitrary value depending on the number of actions in the dialog. The positions to which the left and right sides of each action button are attached also need to be adjusted.

Some known quantities in this equation are that the action area must be at the bottom of a dialog and it must contains PushButtons. While the PushButtons may be either widgets or gadgets, you should probably choose one or the other and use them consistently throughout your application. In general, all of the buttons in the action area should be from the same class, and all of the action areas in an application should be consistent with one another.

* In Motif 1.2, the special behavior to automatically popup a DialogShell relied upon the child being a BulletinBoard or derivative for proper operation. This is no longer the case in Motif 2.x: the ChangeManaged() method of the DialogShell is less sensitive to a particular child class.

Each button in an action area has its own label, its own callback routine, and its own associated client data. To create a general action area, we need a data structure that abstracts this information. The ActionAreaItem structure is defined as follows:

```
typedef struct {
    char         *label;          /* PushButton's Label */
    void         (*callback)();   /* pointer to a callback routine */
    XtPointer    data;            /* client data for the callback routine */
} ActionAreaItem;
```

This data structure contains all of the information that we need to know in order to create an action area; the rest of the information is known or it can be derived.

Now we can write a routine that creates an action area. The purpose of the function is to create and return a composite widget that contains the specified number of PushButtons, where the buttons are arranged horizontally and evenly spaced. The `CreateActionArea`() routine is used in Example 7-4. This program does not do anything substantial, but it does present a generalized architecture for creating dialogs for an application.*

Example 7-4 The action_area.c program

```
/* action_area.c -- demonstrate how CreateActionArea() can be used
** in a real application. Create what would otherwise be identified
** as a PromptDialog, only this is of our own creation. As such,
** we provide a TextField widget for input. When the user presses
** Return, the OK button is activated.
*/
#include <Xm/DialogS.h>
#include <Xm/PushBG.h>
#include <Xm/PushB.h>
#include <Xm/LabelG.h>
#include <Xm/PanedW.h>
#include <Xm/Form.h>
#include <Xm/RowColumn.h>
#include <Xm/TextF.h>

typedef struct {
    char       *label;
    void       (*callback)();
    XtPointer  data;
} ActionAreaItem;

static void do_dialog(Widget, XtPointer, XtPointer);
static void close_dialog(Widget, XtPointer, XtPointer);
static void activate_cb(Widget, XtPointer, XtPointer);
static void ok_pushed(Widget, XtPointer, XtPointer);
static void clear_pushed(Widget, XtPointer, XtPointer);
static void help(Widget, XtPointer, XtPointer);
```

* `XtVaAppInitialize`() is considered deprecated in X11R6.

```
main (int argc, char *argv[])
{
    Widget        toplevel, button;
    XtAppContext  app;

    XtSetLanguageProc (NULL, NULL, NULL);
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv, NULL,
                                    sessionShellWidgetClass, NULL);
    button = XmCreatePushButton (toplevel, "Push Me", NULL, 0);
    XtManageChild (button);
    XtAddCallback (button, XmNactivateCallback, do_dialog, NULL);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

/* callback routine for "Push Me" button. Actually, this represents
** a function that could be invoked by any arbitrary callback. Here,
** we demonstrate how one can build a standard customized dialog box.
** The control area is created here and the action area is created in
** a separate, generic routine: CreateActionArea().
*/
static void do_dialog (Widget w, XtPointer client_data, XtPointer call_data)
{
    Widget             dialog, pane, rc, text_w, action_a, label_g;
    XmString           string;
    Arg                args[6];
    int                n;
    Widget             CreateActionArea(Widget, ActionAreaItem *, int);
    static ActionAreaItem   action_items[] = {
        {"OK",ok_pushed,NULL},
        {"Clear", clear_pushed, NULL},
        {"Cancel", close_dialog, NULL},
        {"Help",help, "Help Button"},
    };
    /* The DialogShell is the Shell for this dialog. Set it up so
    ** that the "Close" button in the window manager's system menu
    ** destroys the shell (it only unmaps it by default).
    */
    n = 0;
    /* give arbitrary title in wm */
    XtSetArg (args[n], XmNtitle, “Dialog Shell”); n++;
    /* system menu "Close" action */
    XtSetArg (args[n], XmNdeleteResponse, XmDESTROY); n++;
    dialog = XmCreateDialogShell (XtParent (w), "dialog", args, n);
    /* now that the dialog is created, set the Cancel button's
    ** client data, so close_dialog() will know what to destroy.
    */
    action_items[2].data = (XtPointer) dialog;
    /* Create the paned window as a child of the dialog. This will
    ** contain the control area and the action area
    ** (created by CreateActionArea() using the action_items above).
    */
    n = 0;
    XtSetArg (args[n], XmNsashWidth, 1); n++;
```

```
    XtSetArg (args[n], XmNsashHeight, 1); n++;
    pane = XmCreatePanedWindow (dialog, "pane", args, n);
    /* create the control area which contains a
    ** Label gadget and a TextField widget.
    */
    rc = XmCreateRowColumn (pane, "control_area", NULL, 0);
    string = XmStringCreateLocalized ("Type Something:");
    n = 0;
    XtSetArg (args[n], XmNlabelString, string); n++;
    label_g = XmCreateLabelGadget (rc, "label", args, n);
    XmStringFree (string);
    XtManageChild (label_g);
    text_w = XmCreateTextField (rc, "text-field", NULL, 0);
    XtManageChild (text_w);
    /* RowColumn is full -- now manage */
    XtManageChild (rc);
    /* Set the client data for the "OK" and "Cancel" buttons */
    action_items[0].data = (XtPointer) text_w;
    action_items[1].data = (XtPointer) text_w;
    /* Create the action area. */
    action_a = CreateActionArea (pane, action_items,
    XtNumber (action_items));
    /* callback for Return in TextField. Use action_a as client data */
    XtAddCallback (text_w, XmNactivateCallback, activate_cb,
                   (XtPointer) action_a);
                   XtManageChild (pane);
}

/* The next four functions are the callback routines for the buttons
** in the action area for the dialog created above. Again, they are
** simple examples, yet they demonstrate the fundamental design approach.
*/
static void close_dialog (Widget w, XtPointer client_data, XtPointer call_data)
{
    Widget shell = (Widget) client_data;
    XtDestroyWidget (shell);
}

/* The "ok" button was pushed or the user pressed Return */
static void ok_pushed (Widget w, XtPointer client_data, XtPointer call_data)
{
    Widget               text_w = (Widget) client_data;
    XmAnyCallbackStruct  *cbs = (XmAnyCallbackStruct *) call_data;
    char                 *text = XmTextFieldGetString (text_w);
    printf ("String = %s\n", text);
    XtFree (text);
}

static void clear_pushed (Widget w, XtPointer client_data, XtPointer call_data)
{
    Widget              text_w = (Widget) client_data;
    XmAnyCallbackStruct *cbs = (XmAnyCallbackStruct *) call_data;
    /* cancel the whole operation; reset to NULL. */
    XmTextFieldSetString (text_w, "");
```

```
}

static void help (Widget w, XtPointer client_data, XtPointer call_data)
{
    String string = (String) client_data;
    puts (string);
}

/* When Return is pressed in TextField widget, respond by getting
** the designated "default button" in the action area and activate
** it as if the user had selected it.
*/
static void activate_cb (Widget text_w, XtPointer client_data,
                         XtPointer call_data)
{
    XmAnyCallbackStruct *cbs = (XmAnyCallbackStruct *) call_data;
    Widget              dflt, action_area = (Widget) client_data;

    /* get the "default button" from the action area... */
    XtVaGetValues (action_area, XmNdefaultButton, &dflt, NULL);
    if (dflt) /* sanity check -- this better work */
        /* make the default button think it got pushed using
        ** XtCallActionProc(). This function causes the button
        ** to appear to be activated as if the user pressed it.
        */
        XtCallActionProc (dflt, "ArmAndActivate", cbs->event, NULL, 0);
}

#define TIGHTNESS 20
Widget CreateActionArea (Widget parent, ActionAreaItem *actions,
                         int num_actions)
{
    Widget       action_area, widget;
    int          i;
    action_area = XmCreateForm (parent, "action_area", NULL, 0);
    XtVaSetValues (action_area,
                   XmNfractionBase, TIGHTNESS*num_actions - 1,
                   XmNleftOffset,   10,
                   XmNrightOffset,  10,
                   NULL);
    for (i = 0; i < num_actions; i++) {
        widget = XmCreatePushButton (action_area, actions[i].label,
                                     NULL, 0);
        XtVaSetValues (widget,
                       XmNleftAttachment,
                           i? XmATTACH_POSITION: XmATTACH_FORM,
                       XmNleftPosition,TIGHTNESS*i,
                       XmNtopAttachment,XmATTACH_FORM,
                       XmNbottomAttachment,XmATTACH_FORM,
                       XmNrightAttachment,
                           i != num_actions - 1 ?
                               XmATTACH_POSITION :
                                   XmATTACH_FORM,
                       XmNrightPosition, TIGHTNESS * i + (TIGHTNESS - 1),
```

```
                        XmNshowAsDefault,i == 0,
                        XmNdefaultButtonShadowThickness, 1,
                        NULL);
            if (actions[i].callback)
            XtAddCallback (widget, XmNactivateCallback, actions[i].callback,
                        (XtPointer) actions[i].data);
            XtManageChild (widget);
            if (i == 0) {
                /* Set the action_area's default button to the first widget
                ** created (or, make the index a parameter to the function
                ** or have it be part of the data structure). Also, set the
                ** pane window constraint for max and min heights so this
                ** particular pane in the PanedWindow is not resizable.
                */
                Dimension height, h;
                XtVaGetValues (action_area, XmNmarginHeight, &h, NULL);
                XtVaGetValues (widget, XmNheight, &height, NULL);
                height += 2 * h;
                XtVaSetValues (action_area,
                               XmNdefaultButton, widget,
                               XmNpaneMaximum,   height,
                               XmNpaneMinimum,   height,
                               NULL);
            }
        }
        XtManageChild (action_area);
        return action_area;
    }
```

The application uses a PushButton to create and pop up a customized dialog. The control area is composed of a RowColumn widget that contains a Label gadget and a TextField widget. The action area is created using `CreateActionArea`(). The actions and the number of actions are specified in the `actions` and `num_actions` parameters. We use a Form widget to lay out the actions. We give the Form the name *action_area*, since it is descriptive and it makes it easy for the user to specify the area in a resource file. The output of the program in shown in Figure 7-7.

Figure 7-7 Output of the action_area program

In order to distribute the PushButtons evenly across the action area, we use the `XmNfractionBase` resource of the Form widget to segment the widget into equal

portions. The value of the resource is based on the value of the TIGHTNESS definition, which controls the spacing between buttons. A higher value causes the PushButtons to be closer together, while a lower value spaces them further apart. We use the value 20 for purely aesthetic reasons. As each button is created, its attachments are set. The left side of the first button and right side of the last button are attached to the left and right edges of the Form, respectively, while all of the other left and right edges are attached to positions.

The callback routine and associated client data for each button are added using XtAddCallback(). The first button in the action area is specified as the default button for the dialog. The XmNdefaultButton resource indicates which button is designated as the default button for certain actions that take place in the control area of the dialog. The XmNactivateCallback of the TextField widget in the control area uses the resource to activate the default button when the user presses the RETURN key in the TextField.

The CreateActionArea() function also sets XmNpaneMaximum and XmNpaneMinimum constraint resources on the action area. These are PanedWindow constraint resources that are used to specify the height of the action area. The assumption, of course, is that the parent of the action area is a PanedWindow. If that is not true, these resource specifications have no effect.

Using a TopLevelShell for a Dialog

You don't have to use a DialogShell widget to implement a dialog. In fact, it is quite common to use a TopLevelShell or even a SessionShell* in cases where the particular functionality is an important part of a larger application. For example, an e-mail application has a variety of functions that range from reading messages to composing new ones. As shown in Figure 7-8, you can have a separate TopLevelShell, complete with a MenuBar, that looks and acts like a separate application, but is still considered a dialog, since it is only a sub-part of the whole application.

As you can see, this dialog uses the same elements as other dialogs. The control area is complete with a ScrolledText region and other controls, while the action area contains action buttons. The principal difference between this dialog (which uses a TopLevelShell) and a dialog implemented with a DialogShell is that this dialog that uses a TopLevelShell may be iconified separately from the other windows in the program.

* The ApplicationShell is considered deprecated in X11R6.

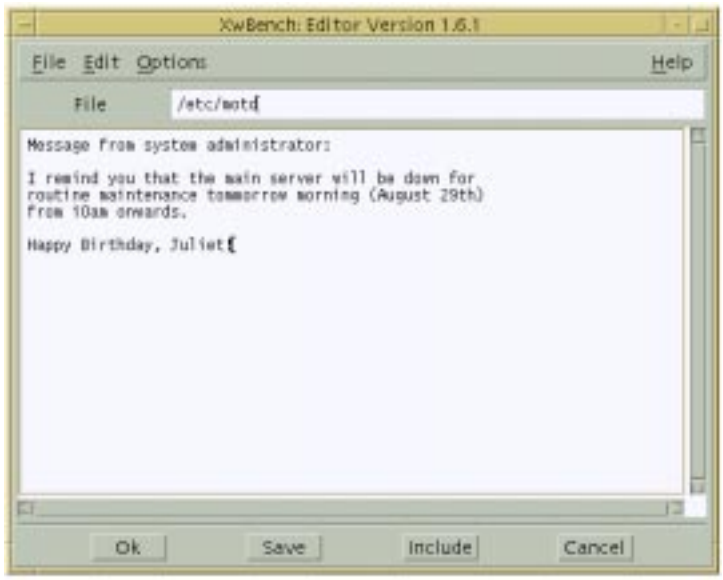


Figure 7-8 An editor dialog from a workbench application

When you need to implement a dialog with a TopLevelShell, you should not regard or implement it as a popup dialog. But for the most part, there is little difference from this approach and the method discussed for regular dialogs. You may still use BulletinBoards, Forms, and RowColumns to manage the inner controls. You still need an action area (provided you want to look and act like a dialog), and you still need to handle the cases where the dialog is popped up and down. You can create the TopLevelShell with `XtVaAppCreateShell`(). The shell is automatically mapped onto the screen when you call `XtPopup`(). You may also want to call `XMapRaised`() on the shell, in case it is already popped up but is not at the top of the window hierarchy.

In direct contrast to the DialogShell widget, managing the immediate child of a TopLevelShell does not cause the dialog to pop up automatically. Even if that child is subclassed from the BulletinBoard widget, this type of behavior only happens if the shell is a DialogShell widget. Because you are using a TopLevelShell, you cannot rely on the special communication that happens between a DialogShell and child widgets.

If you want to use one of the standard Motif dialogs, such as a MessageDialog or a FileSelectionDialog, in a shell widget that can be iconified separately from its primary window shell, you can put the dialog in a TopLevelShell. Create the shell using `XtVaAppCreateShell`() and then use one of the Motif convenience routines to create a MessageBox or a FileSelectionBox, rather than the corresponding dialog widget. The following code fragment shows an example of this usage:

```
shell = XtVaAppCreateShell (NULL, "Class", topLevelShellWidgetClass, dpy,
                            XtNtitle, "Dialog Shell Title", NULL);
dialog = XmCreateMessageBox (shell, "MessageDialog", NULL, 0);
XtAddCallback (dialog, XmNokCallback, callback_func, NULL);
XtAddCallback (dialog, XmNcancelCallback, callback_func, NULL);
XtAddCallback (dialog, XmNhelpCallback, help_func, NULL);
```

Positioning Dialogs

In all of the dialog examples that you have seen so far, the toolkit has handled the positioning of the dialog. For dialogs that use the DialogShell widget with a subclass of BulletinBoard as the immediate child, the `XmNdefaultPosition` resource controls this behavior. If the resource is `True`, the dialog is centered relative to the parent of the DialogShell and placed on top of the parent. If the resource is set to `False`, the application is responsible for positioning the dialog. It is easy to position a dialog using the `XmNmapCallback` resource that is supported by all of the Motif manager widgets, as shown in Example 7-5.*

Example 7-5 The map_dlg.c program

```
/* map_dlg.c -- Use the XmNmapCallback to automatically position
** a dialog on the screen. Each time the dialog is displayed, it
** is mapped down and to the right by 200 pixels in each direction.
*/
#include <Xm/MessageB.h>
#include <Xm/PushB.h>
/* main() --create a pushbutton whose callback pops up a dialog box */
main (int argc, char *argv[])
{
    Widget         toplevel, button;
    XtAppContext   app;
    void           pushed(Widget, XtPointer, XtPointer);

    XtSetLanguageProc (NULL, NULL, NULL);

    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv, NULL,
                                    sessionShellWidgetClass, NULL);
    button = XmCreatePushButton (toplevel, "Push Me", NULL, 0);
    XtAddCallback (button, XmNactivateCallback, pushed,
                   (XtPointer) "Hello World");
    XtManageChild (button);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

/* callback function for XmNmapCallback.  Position dialog in 200 pixel
** "steps".  When the edge of the screen is hit, start over.
*/
```

* `XtVaAppInitialize`() is deprecated in X11R6.

```
static void map_dialog (Widget dialog, XtPointer client_data,
XtPointer call_data)
{
    static Position x, y;
    Dimension       w, h;
    XtVaGetValues(dialog, XmNwidth, &w, XmNheight, &h, NULL);
    if ((x + w) >= WidthOfScreen (XtScreen (dialog)))
        x = 0;
    if ((y + h) >= HeightOfScreen (XtScreen (dialog)))
        y = 0;
    XtVaSetValues (dialog, XmNx, x, XmNy, y, NULL);
    x += 200;
    y += 200;
}
/* pushed() --the callback routine for the main app's pushbutton.
** Create and popup a dialog box that has callback functions for
** the Ok, Cancel and Help buttons.
*/
void pushed (Widget w, XtPointer client_data, XtPointer call_data)
{
    Widget       dialog;
    Arg          arg[5];
    int          n = 0;
    char         *message = (char *) client_data;

    XmString t = XmStringCreateLocalized (message);
    XtSetArg (arg[n], XmNmessageString, t); n++;
    XtSetArg (arg[n], XmNdefaultPosition, False); n++;
    dialog = XmCreateMessageDialog (w, "notice", arg, n);
    XmStringFree (t);
    XtAddCallback (dialog, XmNmapCallback, map_dialog, NULL);
    XtManageChild (dialog);
}
```

Each time the dialog is mapped to the screen, the `map_dialog`() routine is invoked. The routine merely places the dialog at a location that is 200 pixels from its previous position. Obviously, this example is meant to demonstrate the technique of positioning a dialog, rather than providing any useful functionality. The `XmNwidth`, `XmNheight`, `XmNx`, and `XmNy` resources are retrieved from the DialogShell widget since the dialog is a predefined Motif dialog. Similarly, the position of the DialogShell is set by calling `XtVaSetValues`() using the same resources.

If you are using a SessionShell* or a TopLevelShell, rather than a DialogShell, the position of the dialog is subject to various resources that are controlled by the user and/or the window manager. For example, if the user is using *mwm*, she can set the resource `interactivePlacement`, which allows her to position the shell interactively. While it is acceptable for an application to control the placement of a DialogShell, it should not try to control the placement of a TopLevelShell or a SessionShell because that is the user's

* The ApplicationShell is considered deprecated in X11R6.

domain. However, if you feel you must, you can position any shell widget directly by setting its XmNx and XmNy resources to the desired position when the shell is created or later using XtVaSetValues(). The Motif toolkit passes the coordinate values to the window manager and allows it to position the dialog at the intended location.

This issue is an important dilemma in user-interface design. If you are going to hard-code the position of a dialog on the screen, you probably do not want to position the dialog at that location each time that it is popped up. Imagine that you pop up a dialog, move it to an uncluttered area on your screen, interact with it for a while, and then pop it down. If you use the dialog again, you would probably like it to reappear in the location where you put it previously. The best way to handle this dilemma is to avoid doing any of your own dialog placement, with the possible exception of the first time that a dialog is popped up.

Whether or not you want to position a dialog when it is displayed, you may still find it useful to be informed about when a dialog is popped up or down. The XmNmapCallback is not the best tool for this purpose, since it is not called each time the popped-up state of the dialog changes. The XmNpopupCallback and XmNpopdownCallback callbacks are meant for this purpose. These resources are defined and implemented by X Toolkit Intrinsics for all shell widgets. The XmNpopupCallback is invoked each time XtPopup() is called on a shell widget, while the XmNpopdownCallback is called for XtPopdown().

People often get confused by the terminology of a dialog being popped down and a shell being iconified. Remember that whether or not a shell is popped up is independent of its iconic state. Although a DialogShell cannot be iconified separately, other shells can. These shells may also be popped up and down using XtPopup() and XtPopdown() independent of their iconic state. XtPopup() causes a shell to be deiconified, while XtPopdown() causes the dialog and its icon to be withdrawn from the screen, regardless of its iconic state. The subject of window iconification is discussed in Chapter 20, *Interacting With the Window Manager*.

Summary

Obviously, it is impossible to cover all of the possible scenarios of how dialogs can and should be used in an application. If nothing else, you should come away from the chapters on dialogs with a general feeling for the design approach that we encourage. You should also understand the steps that are necessary to create and use both predefined Motif dialogs and customized dialogs. For a final look at some particularly thorny issues in using dialogs, see Chapter 27, *Advanced Dialog Programming*.

In this chapter:

- *Types of Manager Widgets*
- *Creating Manager Widgets*
- *The BulletinBoard Widget*
- *The Form Widget*
- *The RowColumn Widget*
- *The Frame Widget*
- *The PanedWindow Widget*
- *Keyboard Traversal*
- *Summary*

8

Manager Widgets

This chapter provides detailed descriptions of the various classes of Motif manager widgets. Examples explore the various methods of positioning children within the BulletinBoard, Form, and RowColumn widgets.

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

While managers are used and explained in different contexts throughout this book, this chapter discusses the details of the different manager widget classes. Chapter 3, *Overview of the Motif Toolkit*, discusses the general concepts behind manager widgets and how they fit into the broader application model. You are encouraged to review the material in this and other chapters for a wider range of examples, since it is impossible to deal with all of the possibilities here. For an in-depth discussion of the X Toolkit Composite and Constraint widget classes, from which managers are subclassed, see Volume 4, *X Toolkit Intrinsics Programming Manual*.

Types of Manager Widgets

The Manager widget class is a metaclass for a number of functional subclasses. The Manager widget class is never instantiated; the functionality it provides is inherited by each of its subclasses. In this chapter, we describe the general-purpose Motif manager widgets, which are introduced below:

BulletinBoard
: The BulletinBoard is the most basic of the manager widgets. The geometry management is, as the class name implies, like a bulletin board. A child is pinned up on the BulletinBoard in a particular location and remains there until it moves itself or some-

one else moves it. The BulletinBoard widget does not impose any layout policy on its children, but it does support keyboard traversal. The BulletinBoard is a superclass for more sophisticated and useful managers. The BulletinBoard is also designed to be used as the container for dialog boxes, so it has translation tables and callback routines for this purpose. The predefined Motif dialogs use the BulletinBoard widget class to handle all of their input mechanisms; each dialog widget class handles its own geometry management. See Chapter 5, *Introduction to Dialogs*, for a complete discussion of dialogs.

Form

The Form widget is subclassed from the BulletinBoard. The Form extends the capabilities of the BulletinBoard by introducing a sophisticated geometry management policy that involves both absolute and relative positioning and sizing of its children. For example, a Form may lay out its children in a grid-like manner, anchoring the edges of each child to specific positions on the grid, or it may attach the children to one another in a chain-like fashion.

RowColumn

The RowColumn widget lays out its children in rows and columns. Resources control the number of rows or columns and the packing of widgets into those rows and columns. The Motif toolkit uses the RowColumn internally to implement many objects that are not implemented as individual widgets, such as PopupMenus, PulldownMenus, MenuBars, RadioBoxes, and CheckBoxes. There are a number of RowColumn resources that are specific to these objects.

Frame

The purpose of the Frame widget is to provide a visible, three-dimensional border for objects such as RowColumns or Labels that do not provide a border for themselves. The Frame widget may have two children: a work area child and a label child. The Frame sizes itself just big enough to contain its children.

PanedWindow

The PanedWindow manages its children in a vertically or horizontally* tiled format. In a vertical orientation, the widget takes its width from the widest widget in its list of children. When horizontally oriented, the PanedWindow takes its height from the height of the tallest child. The PanedWindow also provides control sashes or grips that enable the user to adjust the individual heights of the PanedWindow's children. Constraint resources for the PanedWindow allow each child to specify its desired maximum and minimum height and whether it may be resized.

DrawingArea

Although the DrawingArea widget is subclassed from the Manager widget class, it is not generally used in the way that conventional managers are used. The widget does

* Horizontal layout was introduced in Motif 2.0.

not do any drawing itself, and it doesn't define any keyboard or mouse behavior, although it does provide callbacks for user input. It is basically a free-form widget that can be used for application-specific purposes. The widget provides callback resources to handle keyboard, mouse, exposure, and resize events. While the DrawingArea widget can have children, it does not manage them in any defined way. Since the DrawingArea widget is typically used for application drawing, rather than for managing other widgets, it is discussed separately in Chapter 11, *The DrawingArea Widget*.

ScrolledWindow

The ScrolledWindow widget provides a viewing area into another widget. The user can adjust the viewing area using ScrollBars that are attached to the ScrolledWindow. The ScrolledWindow can handle scrolling automatically, so that the application does not have to do any work. The widget also has an application-defined mode, which allows an application to control all of the aspects of scrolling. Since the operation of the ScrolledWindow is tied to the operation of ScrollBars, the two widgets are discussed together in Chapter 10, *ScrolledWindows and ScrollBars*.

MainWindow

The MainWindow widget is subclassed from the ScrolledWindow widget. The MainWindow is the standard layout manager for the main application window in a Motif application. The widget is designed to lay out a MenuBar, a work area, ScrollBars, a command area, and a message area. Since the MainWindow is central to many Motif applications, it is discussed separately in Chapter 4, *The Main Window*.

Scale

The Scale widget displays a slider object that has a specific value in a range of values. The user can adjust the value of the widget by moving the slider. The Scale creates and manages its own widgets. In Motif 1.2, the only sensible children that you could add to a Scale were Label widgets that represent tick marks, and these would have to be laid out by the programmer. However, in Motif 2.0, the function `XmScaleSetTicks`() was introduced which automatically places marks at calculated positions along the Scale edge. The widget class is therefore not meant to be a general-purpose manager, so it is described separately in Chapter 16, *The Scale Widget*.

The MessageBox, SelectionBox, FileSelectionBox, and Command widgets are also Motif manager widgets. These widgets are used for predefined Motif dialogs and are discussed in Chapter 5, *Introduction to Dialogs*, Chapter 6, *Selection Dialogs*, and Chapter 7, *Custom Dialogs*.

Creating Manager Widgets

A manager widget may be created and destroyed like any other widget. The main difference between using a manager and other widgets involves when the widget is declared to be managed in the creation process. We normally suggest that you create manager widgets

using the appropriate convenience function or `XtVaCreateWidget()`, rather than using `XtVaCreateManagedWidget()`, and then manage it later using `XtManageChild()`. To understand why this technique can be important, you need to understand how a manager widget manages its children.

A manager widget manages its children by controlling the sizes and positions of the children. The process of widget layout only happens when the child and the parent are both in the managed state. If a child is created as an unmanaged widget, the parent skips over that widget when it is determining the layout until such time as the child is managed. However, if a manager widget is not itself managed, it does not perform geometry management on any of its children regardless of whether those children are managed.*

To demonstrate the problems that you are trying to avoid, consider creating a manager as a managed widget before any of its children are created. The manager is going to have a set of PushButtons as its children. When the first child is added using `XtVaCreateManagedWidget()`, the manager widget negotiates the size and position of the PushButton. Depending on the type of manager widget being used, the parent either changes its size to accommodate the new child or it changes the size of the child to its own size. In either case, these calculations are not necessary because the geometry needs to change as more buttons are added. The problem becomes complicated by the fact that when the manager's size changes, it must also negotiate its new size with its own parent, which causes that parent to negotiate with its parent all the way up to the highest-level shell. If the new size is accepted, the result goes back down the widget tree with each manager widget resizing itself on the way down. Repeating this process each time a child is added almost certainly affects performance.

Because of the different geometry management methods used by the different manager widgets, there is the possibility that all of this premature negotiation can result in a different layout than you intended. For example, as children are added to a RowColumn widget, the RowColumn checks to see if there is enough room to place the new child on the same row or column. If there isn't, then a new row or column is created. This behavior depends heavily on whether the RowColumn is managed and also on whether its size has been established by being realized. If the manager parent is not managed when the children are added, the whole process can be avoided, yet you still have the convenience of using `XtVaCreateManagedWidget()` for all of the widget children should you so wish. When

* To be precise, a manager does not actually manage its children until it is both managed and realized. If you realize all of your widgets at once, by calling `XtRealizeWidget()` on the top-level widget of the application, as described in Chapter 2, *The Motif Programming Model*, it should not make a difference whether a manager is managed before or after its children are created. However, if you are adding widgets to a tree of already-realized widgets, the principles set forth in this section are important. If you are adding children to an already-realized parent, the child is automatically realized when it is managed. If you are adding a manager widget as a child of a realized widget, you should explicitly manage all children before managing the parent. The performance implications can be quite severe otherwise, and can be exponential to the number of already managed children. The code examples all explicitly manage the child before the parent to demonstrate the correct technique, even though the application shell may not as yet be realized.

the manager is itself managed, it queries its children for their size and position requests, calculates its own size requirements, and communicates that size back up the widget tree.

For best results, you should use the appropriate Motif convenience function, `XtCreateWidget`() or `XtVaCreateWidget()` to create manager widgets, reserving `XtVaCreateManagedWidget()` for primitive widgets. Creating a primitive widget as an unmanaged widget serves no purpose, unless you explicitly want the widget's parent to ignore it for some reason. If you are adding another manager as a child, the same principle applies; you should also create it as an unmanaged widget until all its children are added as well. The idea is to descend as deeply into the widget tree and create as many children as possible before managing the manager parents as you ascend back up. Once all the children have been added, `XtManageChild()` can be called for the managers so that they only have to negotiate with their parents once, thus saving time, improving performance, and probably producing better results.

Despite all we've just said, realize that the entire motivating factor behind this principle is to optimize the method by which managers negotiate sizes and positions of their children. If a manager only has one child, it does not matter if you create the manager widget as managed or not. Also, the geometry management constraints of some widgets are such that no negotiation is required between the parent and the children. In these situations, it is not necessary to create the manager as an unmanaged widget, even though it has children. We will explain these cases as they arise.

In the rest of this chapter, we examine the basic manager widget classes and present examples of how they can be used. While geometry management is the most obvious and widely used aspect of the widget class, managers are also responsible for keyboard traversal, gadget display, and gadget event handling. Many of the resources of the Manager metaclass are inherited by each of its subclasses for handling these tasks.

The BulletinBoard Widget

The BulletinBoard is the most basic of the manager widget subclasses. The BulletinBoard widget does not enforce position or size policies on its children, so it is rarely used by applications as a general geometry manager for widgets. The BulletinBoard is the superclass for the Form widget and all of the predefined Motif dialog widgets. To support these roles, the BulletinBoard has a number of resources that are used specifically for communicating with DialogShells.

The BulletinBoard has callback resources for `FocusIn`, `FocusOut`, and `MapNotify` events. These callbacks are invoked when the user moves the mouse or uses the TAB key to traverse the widget hierarchy. The events do not require much visual feedback and they only require application-specific callback routines when an application needs to set internal states based on the events. The `XmNfocusCallback` and `XmNmapCallback resources` are used extensively by DialogShells.

Despite the low profile of the BulletinBoard as a manager widget, there is a lot to be learned from it, since the principles also apply to most other manager widgets. In this spirit, let's take a closer look at the BulletinBoard widget and examine the different things that can be done with it as a manager widget. If you want to use a BulletinBoard directly in an application, you must include the file *<Xm/BulletinB.h>*. The following code fragment shows the two recommended ways to create a BulletinBoard:

```
Widget bboard = XtVaCreateWidget ("name", xmBulletinBoardWidgetClass, parent,
                                  resource-value-list, NULL);
/* Create children */
...
XtManageChild (bboard);

Widget bboard = XmCreateBulletinBoard (parent, "name",
                                       resource-value-array,
                                       resource-value-count);
/* Create children */
...
XtManageChild (bboard);
```

The `parent` parameter is the parent of the BulletinBoard, which may be another manager widget or a shell widget. You can specify any of the resources that are specific to the BulletinBoard, but unless you are using the widget as a dialog box, your choices are quite limited.

Resources

Of the few BulletinBoard resources not tied to DialogShells, the only visual one is `XmNshadowType`. When used in conjunction with the `XmNshadowThickness` resource, you can control the three-dimensional appearance of the widget. There are four possible values for `XmNshadowType`:

`XmSHADOW_IN`	`XmSHADOW_OUT`
`XmSHADOW_ETCHED_IN`	`XmSHADOW_ETCHED_OUT`

The default value for `XmNshadowThickness` is `0`, except when the BulletinBoard is the child of a DialogShell, in which case the default value is `1`. In either case, the value can be changed by the application or by the user.

The `XmNbuttonRenderTable`* resource may be set to a render table as described in Chapter 24, *Render Tables*. This render table is used for each of the button children of the BulletinBoard, when the button does not specify its own render table. If the resource is not specified, its value is taken from the `XmNbuttonRenderTable` of the nearest ancestor which holds the `XmQTspecifyRenderTable` Trait. BulletinBoard, VendorShell, and

* The `XmFontList` is obsolete as of Motif 2.0, and is replaced by the `XmRenderTable`. `XmNbuttonFontList`, `XmNlabelFontList`, and `XmNtextFontList` are deprecated, and the resources `XmNbuttonRenderTable`, `XmNlabelRenderTable`, `XmNtextRenderTable` are preferred respectively.

MenuShell hold this Trait. Similarly, the XmNlabelRenderTable and XmNtextRenderTable resources can be set for Label and Text widgets, respectively, that are direct children of the BulletinBoard.

Geometry Management

Since the BulletinBoard does not provide any geometry management by default, you must be prepared to manage the positions and sizes of the widgets within a BulletinBoard. As a result, you must set the XmNx and XmNy resources for each child. You may also have to set the XmNwidth and XmNheight resources if you need consistent or predetermined sizes for the children. In order to maintain the layout, you must add an event handler for resize (ConfigureNotify) events, so that the new sizes and positions of the children can be calculated. Example 8-1 shows the use of an event handler with the BulletinBoard.*

Example 8-1 The corners.c program

```
/* corners.c -- demonstrate widget layout management for a
** BulletinBoard widget. There are four widgets each labelled
** top-left, top-right, bottom-left and bottom-right. Their
** positions in the bulletin board correspond to their names.
** Only when the widget is resized does the geometry management
** kick in and position the children in their correct locations.
*/
#include <Xm/BulletinB.h>
#include <Xm/PushB.h>

char *corners[] = { "Top Left", "Top Right",
                    "Bottom Left", "Bottom Right" };
static void resize(Widget, XEvent *, String *, Cardinal *);

main (int argc, char *argv[])
{
    Widget         toplevel, bboard, button;
    XtAppContext   app;
    XtActionsRec   rec;
    int            i;

    XtSetLanguageProc (NULL, NULL, NULL);
    /* Initialize toolkit and create toplevel shell */
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0,  &argc, argv, NULL,
                                    sessionShellWidgetClass, NULL);
    /* Create your standard BulletinBoard widget */
    bboard = XmCreateBulletinBoard (toplevel, "bboard", NULL, 0);
    /* Set up a translation table that captures "Resize" events
    ** (also called ConfigureNotify or Configure events). If the
    ** event is generated, call the function resize().
    */
    rec.string = "resize";
```

* XtVaAppInitialize() is considered deprecated in X11R6.

```
    rec.proc = resize;
    XtAppAddActions (app, &rec, 1);
    XtOverrideTranslations (bboard,
                            XtParseTranslationTable ("<Configure>: resize()"));
    /* Create children of the dialog -- a PushButton in each corner. */
    for (i = 0; i < XtNumber (corners); i++) {
        button = XmCreatePushButton (bboard, corners[i], NULL, 0);
        XtManageChild (button);
    }
    XtManageChild (bboard);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}
/* resize(), the routine that is automatically called by Xt upon the
** delivery of a Configure event. This happens whenever the widget
** gets resized.
*/
static void resize ( Widget       w,           /* Widget that resized */
                     XEvent       *event,
                     String       args[],      /* unused */
                     Cardinal     *num_args)   /* unused */
{
    WidgetList         children;
    Dimension          w_width, w_height;
    short              margin_w, margin_h;
    XConfigureEvent    *cevent = (XConfigureEvent *) event;
    int                width = cevent->width;
    int                height = cevent->height;

    /* get handle to BulletinBoard's children and marginal spacing */
    XtVaGetValues (w, XmNchildren, &children, XmNmarginWidth, &margin_w,
                   XmNmarginHeight, &margin_h, NULL);
    /* place the top left widget */
    XtVaSetValues (children[0], XmNx, margin_w, XmNy, margin_h, NULL);
    /* top right */
    XtVaGetValues (children[1], XmNwidth, &w_width, NULL);
    XtVaSetValues (children[1], XmNx, width - margin_w - w_width,
                   XmNy, margin_h, NULL);
    /* bottom left */
    XtVaGetValues (children[2], XmNheight, &w_height, NULL);
    XtVaSetValues (children[2], XmNx, margin_w,
                   XmNy, height - margin_h - w_height,
                   NULL);
    /* bottom right */
    XtVaGetValues (children[3], XmNheight, &w_height,
                   XmNwidth, &w_width, NULL);
    XtVaSetValues (children[3], XmNx, width - margin_w - w_width,
                   XmNy, height - margin_h - w_height,
                   NULL);
}
```

The program uses four widgets, labelled *Top Left*, *Top Right*, *Bottom Left*, and *Bottom Right*. The positions of the buttons in the BulletinBoard correspond to their names. Since the widgets are not positioned when they are created, the geometry management only

happens when the widget is resized. Figure 8-1 shows the application before and after a resize event.

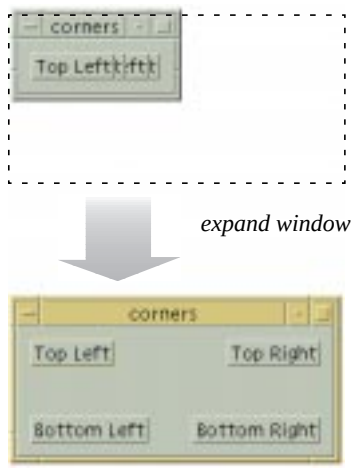


Figure 8-1 Output of the corners program before and after a resize event

When a resize event occurs, X generates a `ConfigureNotify` event. This event is interpreted by Xt and the translation table of the widget corresponding to the resized window is searched to see if the application is interested in being notified of the event. We have indicated interest in this event by calling `XtAppAddActions()` and `XtOverrideTranslations()`, as shown below:

```
XtActionsRec rec;
...
rec.string = "resize";
rec.proc = resize;
XtAppAddActions (app, &rec, 1);
XtOverrideTranslations (bboard,
                    XtParseTranslationTable ("<Configure>: resize()"));
```

As described in Volume 4, *X Toolkit Intrinsics Programming Manual*, a translation table pairs a sequence of one or more events with a sequence of one or more functions that are called when the event sequence occurs. In this case, the event is a `ConfigureNotify` event and the function is `resize()`. Translations are specified as strings and then parsed into an internal format with the function `XtParseTranslationTable()`. The routine creates an internal structure of events and the functions to which they correspond. Xt provides the table for translating event strings such as `<Configure>` to the actual `ConfigureNotify` event, but Xt cannot convert the string `resize()` to an actual function unless we provide a lookup table.The `XtActionsRec` type performs this task. The structure is defined as follows:

```
typedef struct {
    String       string;
```

```
    XtActionProc proc;
} XtActionsRec;
```

The action list is initialized to map the string `resize` to the actual function `resize()` using `XtAppAddActions()`. We install the translation table on the widget using `XtOverrideTranslations()` so that when a `ConfigureNotify event` occurs, the `resize()` function is called.

The `resize()` function takes four arguments. The first two arguments are a pointer to the widget in which the event occurred and the event structure. The `args` and `num_args` parameters are ignored because we did not specify any extra parameters to be passed to the function when we installed it. Since the function is called as a result of the event happening on the BulletinBoard widget, we know that we are dealing with a composite widget. We also know that there is only one event type that could have caused the function to be called, so we cast the `event` parameter accordingly.

The task of the function is to position the children so that there is one per corner in the BulletinBoard. We get a handle to all of the children of the BulletinBoard. Since we are going to place the children around the perimeter of the widget, we also need to know how far from the edge to place them. This distance is taken from the values for `XmNmarginWidth` and `XmNmarginHeight`. All three resource values are retrieved in the following call:

```
XtVaGetValues (w, XmNchildren,     &children,
                  XmNmarginWidth,  &margin_w,
                  XmNmarginHeight, &margin_h,
                  NULL);
```

The remainder of the function simply places the children at the appropriate positions within the BulletinBoard. The routine uses a very simple method for geometry management, but it does demonstrate the process.

The general issue of geometry management for composite widgets is not trivial. If you plan on doing your own geometry management for a BulletinBoard or any other composite widget, you should be very careful to consider all the resources that could possibly affect layout. In our example, we considered the margin width and height, but there is also `XmNallowOverlap`, `XmNborderWidth` (which is a general Core widget resource), `XmNshadowThickness` (a general manager widget resource) and the same values associated with the children of the BulletinBoard.

There are also issues about what to do if a child decides to resize itself, such as if a Label widget gets wider. In this case, you must first evaluate what the geometry layout of the widgets would be if you were to grant the Label permission to resize itself as it wants. This evaluation is done by asking each of the children how big they want to be and calculating the hypothetical layout. The BulletinBoard either accepts or rejects the new layout. Of course, the BulletinBoard may have to make itself bigger too, which requires asking its parent for a new size, and so on. If the BulletinBoard cannot resize itself, then you have to

decide whether to force other children to be certain sizes or to reject the resize request of the child that started all the negotiation. Geometry management is by no means a simple task; it is explained more completely in Volume 4, *X Toolkit Intrinsics Programming Manual*.

The Form Widget

The Form widget is subclassed from the BulletinBoard class, so it inherits all of the resources that the BulletinBoard has to offer. Accordingly, the children of a Form can be placed at specific x, y coordinates and geometry management can be performed as in Example 8-1. However, the Form provides additional geometry management features that allow its children to be positioned relative to one another and relative to specific locations in the Form.

In order to use a Form, you must include the file *<Xm/Form.h>*. A Form is created in a similar way to other manager widgets, either through a convenience routine or using the general purpose Xt mechanisms, as shown below:

```
Widget form = XtVaCreateWidget ("name",  xmFormWidgetClass, parent,
                                resource-value-list, NULL);
/* create children */
XtManageChild (form);

Widget form = XmCreateForm ( parent, "name", resource-value-array,
                            resource-value-count);
/* create children */
XtManageChild (form);
```

Form Attachments

Geometry management in a Form is done using attachment resources. These resources are constraint resources, which means that they are specified for the children of the Form. The resources provide various ways of specifying the position of a child of a Form by attaching each of the four sides of the child to another entity. The side of a widget can be attached to another widget, to a fixed position in the Form, to a flexible position in the Form, to the

Form itself, or to nothing at all. These attachments can be considered hooks, rods, and anchor points, as shown in Figure 8-2.

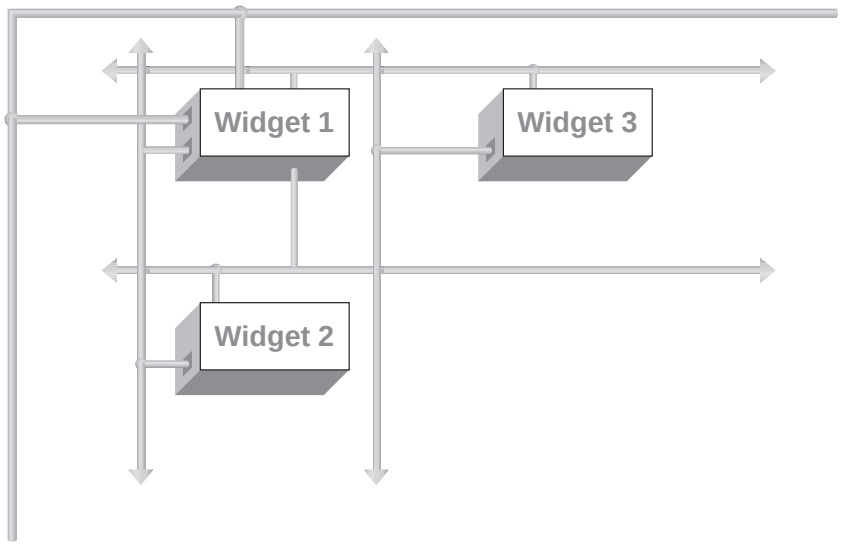


Figure 8-2 Attachments in a Form

In this figure, there are three widgets. The sizes and types of the widgets are not important. What is important is the relationship between the widgets with respect to their positions in the Form. *Widget 1* is attached to the top and left sides of the Form by creating two attachments. The top side of the widget is hooked to the top of the Form. It can slide from side to side, but it cannot be moved up or down (just like a shower curtain). The left side can slide up and down, but not to the right or to the left. Given these two attachment constraints, the top and left sides of the widget are fixed. The right and bottom edges of the widget are not attached to anything, but other widgets are attached to those edges.

The left side of *Widget 2* is attached to the right side of *Widget 1*. Similarly, the top side of *Widget 2* is attached to the top side of *Widget 1*. As a result, the top and left sides of the widget cannot be moved unless *Widget 1* moves. The same kind of attachments hold for *Widget 3*. The top side of this widget is attached to the bottom of *Widget 1* and its left side is attached to the left side of *Widget 1*. Given these constraints, no matter how large each of the widgets may be, or how the Form may be resized, the positional relationship of the widgets is maintained.

In general, you must attach at least two adjacent edges of a widget to keep it from moving unpredictably. If you only attach one edge of a widget, you are only specifying relative position: both opposing sides must be attached for potential resize behavior.

The following resources represent the four sides of a widget:

```
XmNtopAttachment            XmNbottomAttachment
XmNrightAttachment          XmNleftAttachment
```

For example, if we want to specify that the top of a widget is attached to something, we use the XmNtopAttachment resource. Each of the four resources can be set to one of the following values:

```
XmATTACH_FORM               XmATTACH_OPPOSITE_FORM
XmATTACH_WIDGET             XmATTACH_OPPOSITE_WIDGET
XmATTACH_NONE
XmATTACH_SELF               XmATTACH_POSITION
```

XmATTACH_FORM

When an attachment is set to XmATTACH_FORM, the specified side is attached to the Form as shown in Figure 8-3. If the resource that has this value is XmNtopAttachment, then the top side of the widget is attached to the top of the Form. The top attachment does not guarantee that the widget will not move from side to side. If XmNbottomAttachment is also set to XmATTACH_FORM, the bottom of the widget is attached to the bottom side of the Form. With both of these attachments, the widget is resized to the height of the Form itself. The same would be true for the right and left edges of the widget if they were attached to the Form.

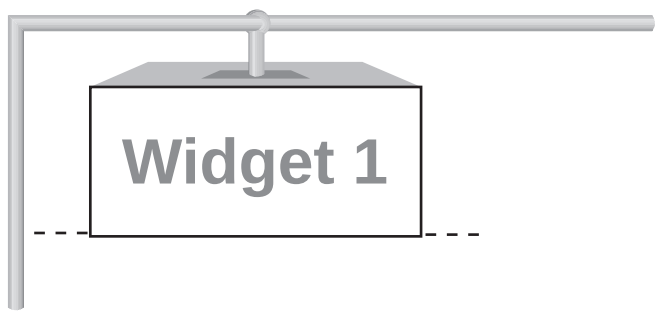


Figure 8-3 XmNtopAttachment set to XmATTACH_FORM

XmATTACH_OPPOSITE_FORM

When an attachment is set to XmATTACH_OPPOSITE_FORM, the specified side of the widget is attached to the opposite side of the Form. For example, if XmNtopAttachment is set to XmATTACH_OPPOSITE_FORM, the top side of the widget is attached to the bottom side of the Form. This value must be used with a negative offset value (discussed in the next section) or the widget is placed off of the edge of the Form and it is not visible. While it may seem confusing, this value is the only one that can be applied to an attachment resource that allows you to specify a constant offset from the edge of a Form.

XmATTACH_WIDGET

The XmATTACH_WIDGET value indicates that the side of a widget is attached to another widget. The other widget must be specified using the appropriate resource from the following list:

```
XmNtopWidget                XmNbottomWidget
XmNleftWidget               XmNrightWidget
```

The value for one of these resources must be the widget ID. For example, Figure 8-4 shows how to attach the right side of *Widget 1* to the left side of *Widget 2*. This attachment method is commonly used to chain together a series of adjacent widgets. Chaining widgets horizontally does not guarantee that the widgets will be aligned vertically, or vice versa.

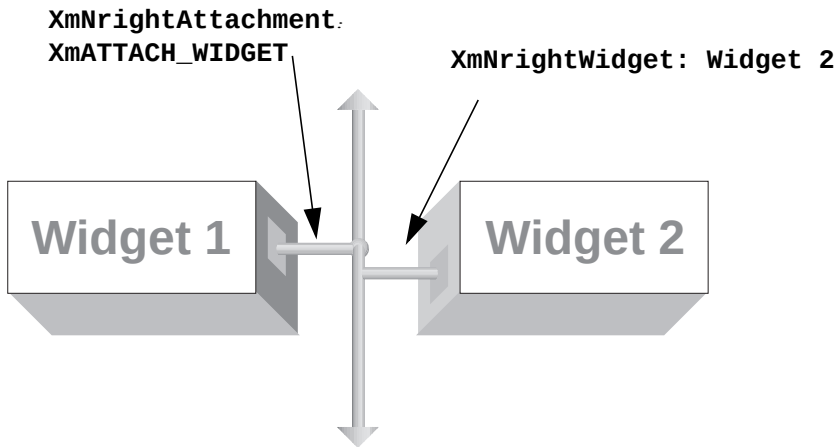


Figure 8-4 XmNrightAttachment set to XmATTACH_WIDGET

XmATTACH_OPPOSITE_WIDGET

The XmATTACH_OPPOSITE_WIDGET value is just like XmATTACH_WIDGET, except that the widget is attached to the same edge of the specified widget, as shown in Figure 8-5. In this case, the right side of *Widget 1* is attached to the right side of *Widget 3*. This attachment method allows you to align the edges of a group of widgets. As with XmATTACH_WIDGET, the other widget must be specified using XmNtopWidget, XmNbottomWidget, XmNleftWidget, or XmNrightWidget.

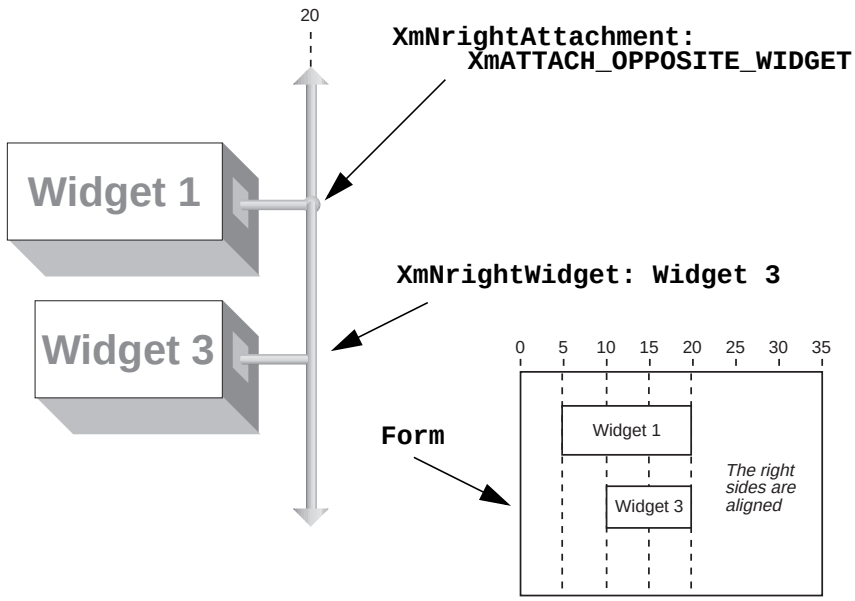


Figure 8-5 XmNrightAttachment set to XmATTACH_OPPOSITE_WIDGET

XmATTACH_NONE

XmATTACH_NONE specifies that the side of a widget is not attached to anything, which is the default value*. This case could be represented by a dangling hook that is not attached to anything. If the entire widget moves because another side is attached to something, then this side gets dragged along with it so that the widget does not need resizing. Unless a particular side of a widget is attached to something, that side of the widget is free-floating and moves proportionally with the other parts of the widget.

XmATTACH_POSITION

When the side of a widget is attached using XmATTACH_POSITION, the side is anchored to a relative position in the Form. This value works by segmenting the Form into a fixed number of equally-spaced horizontal and vertical positions, based on the value of the XmNfractionBase resource. The position of the side must be specified using the appropriate resource from the following list:

XmNtopPosition	XmNbottomPosition
XmNleftPosition	XmNrightPosition

Later in this chapter there is a complete discussion of position attachments.

* This is true for any individual edge. However, a widget which has no attachments specified on *any* side will have an implicit XmATTACH_FORM attachment on the top and left edges assigned by the containing Form.

XmATTACH_SELF

When an attachment is set to XmATTACH_SELF, the side of the widget is attached to its initial position in the Form. You position the widget initially by specifying its x,y location in the Form. After the widget has been placed in the Form, the attachment for the side reverts to XmATTACH_POSITION, with the corresponding position resource set to the relative position of the x,y coordinate in the Form.

Some Examples

Now that we have explained the concept of Form attachments, we can reimplement the four corners example from the previous section. Unlike in the previous version, we no longer need a resize procedure to calculate the positions of the widgets. By specifying the correct attachments, as shown in Example 8-2, the widgets are placed and managed correctly by the Form when it is resized.*

Example 8-2 The form_corners.c program

```
/* form_corners.c -- demonstrate form layout management. Just as
** in corners.c, there are four widgets each labelled top-left,
** top-right, bottom-left and bottom-right. Their positions in the
** form correspond to their names. As opposed to the BulletinBoard
** widget, the Form manages this layout management automatically by
** specifying attachment types for each of the widgets.
*/
#include <Xm/PushB.h>
#include <Xm/Form.h>

char *corners[] = {"Top Left", "Top Right", "Bottom Left", "Bottom Right"};

main (int argc, char *argv[])
{
    Widget         toplevel, form, button;
    XtAppContext   app;

    XtSetLanguageProc (NULL, NULL, NULL);
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0,  &argc, argv,
                                    NULL,sessionShellWidgetClass, NULL);
    form = XmCreateForm (toplevel, "form", NULL, 0);
    /* Attach the edges of the widgets to the Form. Which edge of
    ** the widget that's attached is relative to where the widget is
    ** positioned in the Form. Edges not attached default to having
    ** an attachment type of XmATTACH_NONE.
    */
    button = XmCreatePushButton (form, corners[0], NULL, 0);
    XtVaSetValues (button, XmNtopAttachment, XmATTACH_FORM, XmNleftAttachment,
                   XmATTACH_FORM, NULL);
    XtManageChild (button);
    button = XmCreatePushButton (form, corners[1], NULL, 0);
```

* XtVaAppInitialize() is considered deprecated in X11R6.

```
    XtVaSetValues (button, XmNtopAttachment, XmATTACH_FORM,
                   XmNrightAttachment, XmATTACH_FORM, NULL);
    XtManageChild (button);
    button = XmCreatePushButton (form, corners[2], NULL, 0);
    XtVaSetValues (button, XmNbottomAttachment, XmATTACH_FORM,
                   XmNleftAttachment, XmATTACH_FORM, NULL);
    XtManageChild (button);
    button = XmCreatePushButton (form, corners[3], NULL, 0);
    XtVaSetValues (button, XmNbottomAttachment, XmATTACH_FORM,
                   XmNrightAttachment, XmATTACH_FORM, NULL);
    XtManageChild (button);
    XtManageChild (form);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}
```

In this example, two sides of each widget are attached to the Form. It is not necessary to attach the other sides of the widgets to anything else. If we attach the other sides to each other, the widgets would have to be resized so that they could stretch to meet each other. With the specified attachments, the output of the program looks just like the output in Figure 8-1.

A more complex example of Form attachments is shown in Example 8-3. This example implements the layout shown in Figure 8-2*

Example 8-3 The attach.c program

```
/* attach.c -- demonstrate how attachments work in Form widgets. */
#include <Xm/PushB.h>
#include <Xm/Form.h>

main (int argc, char *argv[])
{
    Widget        toplevel, parent, one, two, three;
    XtAppContext  app;

    XtSetLanguageProc (NULL, NULL, NULL);
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv, NULL,
                                    sessionShellWidgetClass, NULL);
    parent = XmCreateForm (toplevel, "form",  NULL, 0);
    one = XmCreatePushButton (parent, "One", NULL, 0);
    two = XmCreatePushButton (parent, “Two”, NULL, 0);
    three = XmCreatePushButton (parent, “Three”, NULL, 0);
    XtVaSetValues (one,
                   XmNtopAttachment, XmATTACH_FORM,
                   XmNleftAttachment, XmATTACH_FORM,
                   NULL);
    XtVaSetValues (two,
                   XmNleftAttachment, XmATTACH_WIDGET,
                   /* attach top to same y coordinate as top of "one" */
```

* XtVaAppInitialize() is considered deprecated in X11R6.

```
                    XmNleftWidget, one,
                    XmNtopAttachment, XmATTACH_OPPOSITE_WIDGET,
                    XmNtopWidget, one,
                    NULL);
    XtVaSetValues (three,
                    XmNtopAttachment, XmATTACH_WIDGET,
                    /* attach left to same x coordinate as left of "one" */
                    XmNtopWidget, one,
                    XmNleftAttachment, XmATTACH_OPPOSITE_WIDGET,
                    XmNleftWidget, one,
                    NULL);
    XtManageChild (one);
    XtManageChild (two);
    XtManageChild (three);
    XtManageChild (parent);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}
```

The example uses three PushButton gadgets inside of a Form widget. The output of the program is shown in Figure 8-6.

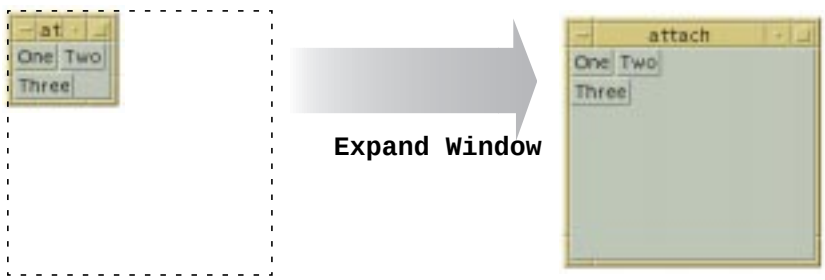


Figure 8-6 Output of the attach program

You should notice that the widgets are packed together quite tightly, which might not be how you expected them to appear. In order to space the widgets more reasonably, we need to specify some distance between them using attachment offsets.

Attachment Offsets

Attachment offsets control the spacing between widgets and the objects to which they are attached. The following resources represent the attachment offsets for the four sides of a widget:

```
XmNleftOffset              XmNrightOffset
XmNtopOffset               XmNbottomOffset
```

Figure 8-7 shows the graphic representation of attachment offsets.

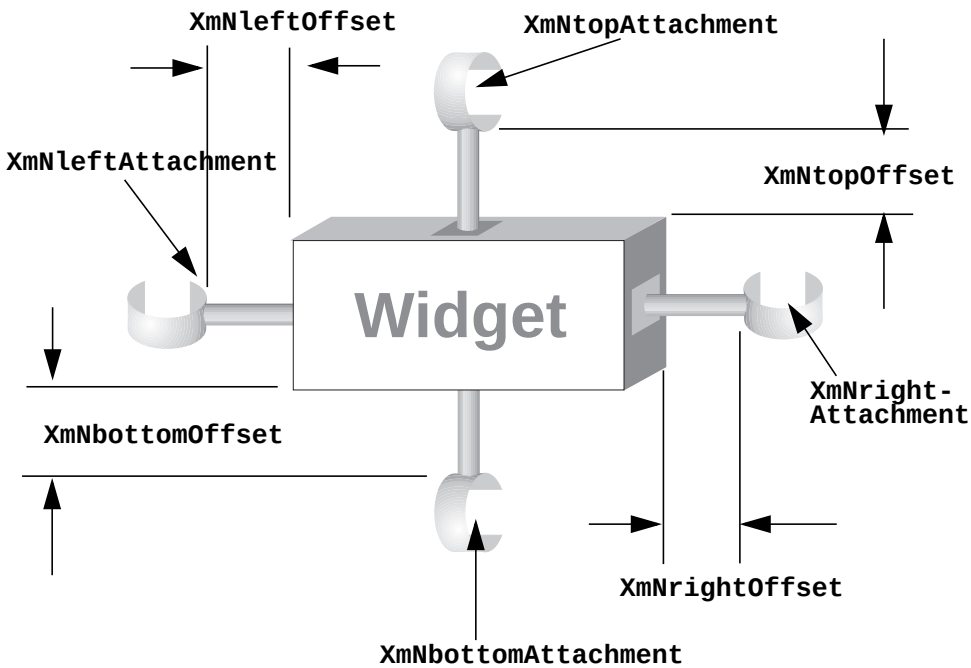


Figure 8-7 Attachment offsets

By default, offsets are set to `0` (zero), which means that there is no offset*, as shown in the output forExample 8-3. To make the output more reasonable, we need only to set the left offset between widgets *One* and *Two* and the top offset to between widgets *One* and *Three*. The resources values can be hard-coded in the application or set in a resource file, using the following specification:

```
*form.One.leftOffset:   10
*form.One.topOffset:    10
*form.Two.leftOffset:   10
*form.Three.topOffset:  10
```

Our choice of the value `10` was arbitrary. The widgets are now spaced more appropriately, as shown in Figure 8-8.

* Not strictly true: if no offset is specified, the value depends upon the Form `XmNhorizontalSpacing` and `XmNverticalSpacing` resources, which are zero by default. An explicit offset of zero overrides any Form spacing resources.

Figure 8-8 Output of the attach program with offset resources set to 10

While the layout of the widgets can be improved by setting offset resources, it is also possible to disrupt the layout. Consider the following resource specifications:

```
*form*leftOffset: 10
*form*topOffset:  10
```

While it might seem that these resource values are simply a terser way to specify the offsets shown earlier, Figure 8-9 makes it clear that these specifications do not produce the desired effect.

Figure 8-9 Output of the attach program with inappropriate offset resources

An application should hard-code whatever resources may be necessary to prevent the user from setting values that would make the application non-functional or aesthetically unappealing. Offset resource values can be tricky because they apply individually to each side of each widget in a Form. The problem with the resource specifications used to produce Figure 8-9 is that the offsets are being applied to each side of every widget, when some of the alignments need to be precise. In order to prevent this problem, we need to hard-code the offsets for particular attachments, as shown in the following code fragment:

```
two = XmCreatePushButton (parent, "Two",  NULL, 0);
XtVaSetValues (two, XmNleftAttachment, XmATTACH_WIDGET,
                    XmNleftWidget, one,
                    XmNtopAttachment,XmATTACH_OPPOSITE_WIDGET,
                    XmNtopWidget, one,
                    XmNtopOffset, 0,
                    NULL);
three = XmCreatePushButton (parent, "Three",  NULL, 0);
XtVaSetValues (three, XmNtopAttachment, XmATTACH_WIDGET,
                      XmNtopWidget, one,
                      XmNleftAttachment, XmATTACH_OPPOSITE_WIDGET,
                      XmNleftWidget, one,
                      XmNleftOffset, 0,
                      NULL);
```

The use of zero-length offsets guarantees that the widgets they are associated with are aligned exactly with the widgets to which they are attached, regardless of any resource specifications made by the user. A general rule of thumb is that whenever you use XmATTACH_OPPOSITE_WIDGET, you should also set the appropriate offset to zero so that the alignment remains consistent.

In some situations it is necessary to use negative offsets to properly arrange widgets in a Form. The most common example of this situation occurs when using the XmATTACH_OPPOSITE_FORM attachment. Unless you use a negative offset, as shown in Figure 8-10, the widgets are placed off the edge of the Form and are not visible.

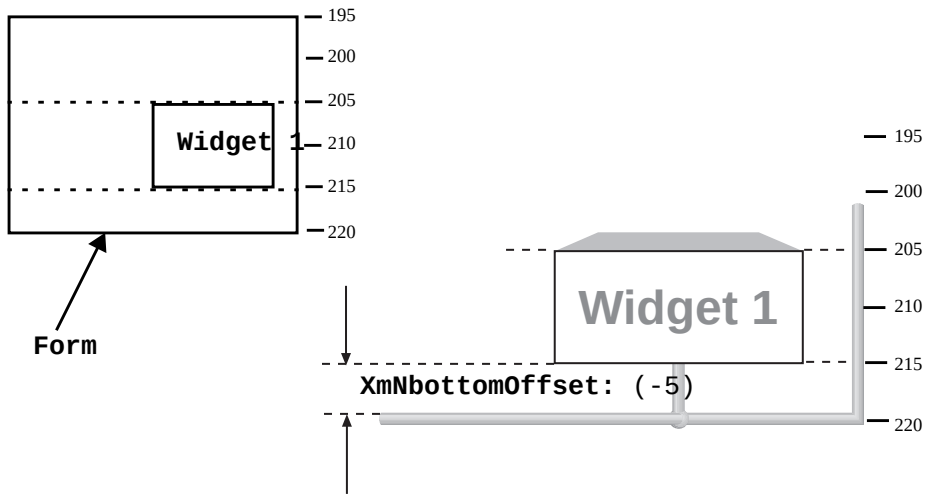


Figure 8-10 XmNleftAttachment of XmATTACH_OPPOSITE_WIDGET with negative offset

Position Attachments

Form positions provide another way to position widgets within a Form. The concept is similar to the hook and rod principle described earlier, but in this case the widgets are anchored on at positions that are based on imaginary longitude and latitude lines that are used to segment the Form into equal pieces. The resource used to partition the Form into segments is XmNfractionBase. Although the name of this resource may suggest complicated calculations, you just need to know that the Form is divided horizontally and vertically into the number of partitions represented by its value. For example, Figure 8-11 shows how a Form is partitioned if XmNfractionBase is set to 5.

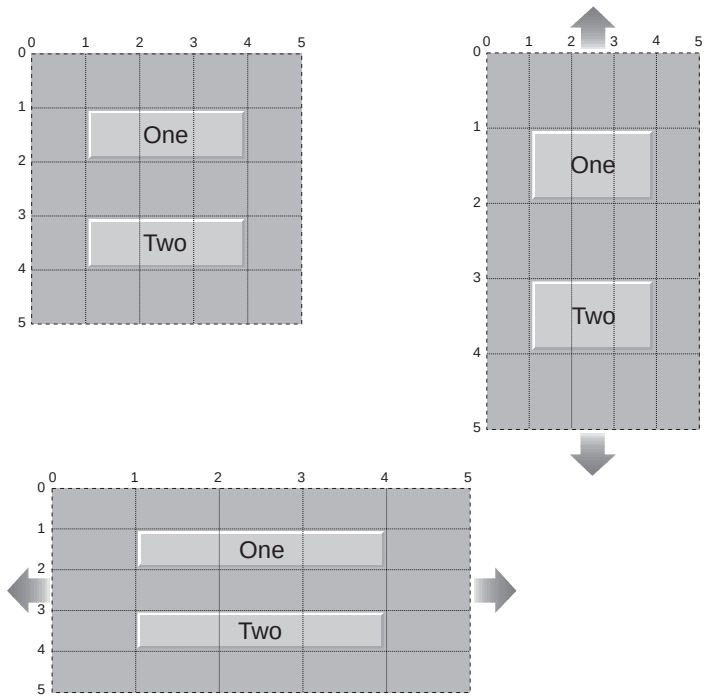


Figure 8-11 Form with XmNfractionBase set to 5

As you can see, there are an equal number of horizontal and vertical partitions, but the size of the horizontal partitions is not the same as the size of the vertical partitions. It is currently not possible to set the number of horizontal partitions separately from the number of vertical ones, although it is possible to work around this shortcoming, as we will describe shortly.

Widgets are placed at the coordinates that represent the partitions by specifying XmATTACH_POSITION for the attachment resource and by specifying a coordinate value for the corresponding position resource. The position resources are XmNtopPosition, XmNbottomPosition, XmNleftPosition, and XmNrightPosition. For example, if we wanted to attach the top and left sides of a PushButton to position 1, we could use the following code fragment:

```
button = XmCreatePushButton (form, "name",  NULL, 0);
XtVaSetValues (button, XmNtopAttachment, XmATTACH_POSITION,
                       XmNtopPosition, 1,
                       XmNleftAttachment, XmATTACH_POSITION,
                       XmNleftPosition, 1,
                       NULL);
```

The right and bottom attachments are left unspecified, so those edges of the widget are not explicitly positioned by the Form. If attachments had been specified for these edges, the widget would have to be resized by the Form in order to satisfy all the attachment constraints.

One obvious example of using position attachments is to create a tic-tac-toe board layout, as is done in Example 8-4.*

Example 8-4 The ticktactoe.c program

```
/* tictactoe.c -- demonstrate how fractionBase and XmATTACH_POSITIONs
** work in Form widgets.
*/
#include <Xm/PushB.h>
#include <Xm/Form.h>

main (int argc, char *argv[])
{
    XtAppContext   app;
    Widget         toplevel, parent, w;
    int            x, y, n;
    Arg            args[10];
    /* callback for each PushButton */
    extern void    pushed(Widget, XtPointer, XtPointer);

    XtSetLanguageProc (NULL, NULL, NULL);
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv, NULL,
                                    sessionShellWidgetClass, NULL);
    n = 0;
    XtSetArg (args[n], XmNfractionBase, 3); n++;
    parent = XmCreateForm (toplevel, "form", args, n);

    for (x = 0; x < 3; x++)
        for (y = 0; y < 3; y++) {
            n = 0;
            XtSetArg (args[n], XmNtopAttachment, XmATTACH_POSITION);    n++;
            XtSetArg (args[n], XmNtopPosition, y);                      n++;
            XtSetArg (args[n], XmNleftAttachment, XmATTACH_POSITION);   n++;
            XtSetArg (args[n], XmNleftPosition, x);                     n++;
            XtSetArg (args[n], XmNrightAttachment, XmATTACH_POSITION);  n++;
            XtSetArg (args[n], XmNrightPosition, x+1);                  n++;
            XtSetArg (args[n], XmNbottomAttachment, XmATTACH_POSITION); n++;
            XtSetArg (args[n], XmNbottomPosition, y+1);                 n++;
            w = XmCreatePushButton (parent, "   ",  args, n);
            XtAddCallback (w, XmNactivateCallback, pushed, NULL);
            XtManageChild (w);
        }

    tManageChild (parent);
    XtRealizeWidget (toplevel);
```

* XtVaAppInitialize() is considered deprecated in X11R6.

```
    XtAppMainLoop (app);
}

void pushed (Widget         w,            /* PushButton that got activated */
             XtPointer    client_data,    /* unused */
             XtPointer    call_data)
{
    char      buf[2];
    XmString  str;
    XmPushButtonCallbackStruct  *cbs =
                          (XmPushButtonCallbackStruct *) call_data;

    /* Shift key gets an O. (xbutton and xkey happen to be similar) */
    if (cbs->event->xbutton.state & ShiftMask)
        buf[0] = '0';
    else
        buf[0] = 'X';
    buf[1] = 0;
    str = XmStringCreateLocalized (buf);
    XtVaSetValues (w, XmNlabelString, str, NULL);
    XmStringFree (str);
}
```

The output of this program is shown in Figure 8-12.

Figure 8-12 Output of the tictactoe program

As you can see, the children of the Form are equally sized because their attachment positions are segmented equally. If the user resizes the Form, all of the children maintain their relationship to one another. The PushButtons simply grow or shrink to fill the form.

One common use of positional attachments is to lay out a number of widgets that need to be of equal size and equal spacing. For example, you might use this technique to arrange the buttons in the action area of a dialog. Chapter 7, *Custom Dialogs*, provides a detailed discussion of how to arrange buttons in this manner.

There may be situations where you would like to attach widgets to horizontal positions that do not match up with how you'd like to attach their vertical positions. Since the fraction base cannot be set differently for the horizontal and vertical orientations, you have to use the least common multiple as the fraction base value. For example, say you want to position the tops and bottoms of all of your widgets to the 2nd and 4th positions, as if the Form were segmented vertically into 5 parts. But, you also want to position the left and right edges of those same widgets to the 3rd, 5th, 7th, and 9th positions, as if it were segmented into 11

parts. You would have to apply some simple arithmetic and set the value for XmNfractionBase to 55 (5x11). The top and bottom edges would be set to the 22nd (2x11) and 44th (4x11) positions and the left and right edges would be set to the 15th (3x5), 25th (5x5), 35th (7x5), and 45th (9x5) positions.

Additional Resources

There are a few other useful Form resources that we have not covered so far. The XmNhorizontalSpacing resource can be used to specify the distance between horizontally adjacent widgets, while XmNverticalSpacing specifies the distance between vertically adjacent widgets. These values only apply when the left and right offset values are not specified, so they are intended to be used as global offset values global for a Form. The following resource specification:

```
*horizontalSpacing: 10
```

is equivalent to:

```
*leftOffset:   10
*rightOffset:  10
```

The XmNrubberPositioning resource specifies the default attachments for widgets in the Form. The default value of False indicates that the top and left edges are attached to the form by default. If XmNrubberPositioning is set to True, the top and left attachments are set to XmATTACH_POSITION by default. If the XmNtopAttachment or XmNleftAttachment resource is explicitly set for a widget, then the default attachment has no effect.

The XmNresizable resource is another constraint resource that can be set on the children of a Form widget. This resource indicates whether or not the Form tries to grant resize requests from the child.

Nested Forms

Some widget layouts are difficult to create using a single Form widget. Since a manager widget can contain other managers, it is often possible to generate the desired layout by using a Form within a Form. One common problem is that there are no Form attachments available to align two widgets horizontally if they have different heights. We need a middle attachment resource, but one doesn't exist. For example, if you have a series of Labels and Text widgets that you want to pair off and stack vertically, it would be nice to align each pair of widgets at their mid sections.

To solve this problem, we can place each Label-Text widget pair in a separate Form. If the top and bottom edges of the widgets are attached to the Form, the widgets are stretched to satisfy the constraints, which means that they are aligned horizontally. All of these smaller

Form widgets can be placed inside of a larger Form widget. Example 8-5 shows an implementation of this idea.*

Example 8-5 The text_form.c program

```
/* text_form.c -- demonstrate how attachments work in Form widgets
* by creating a text-entry form type application.
*/
#include <Xm/LabelG.h>
#include <Xm/Text.h>
#include <Xm/Form.h>

char *prompts[] = {"Name:", "Phone:", "Address:",
                   "City:", "State:", "Zip Code:"};

main (int argc, char *argv[])
{
    Widget         toplevel, mainform, subform, last_subform, label, text;
    XtAppContext   app;
    char           buf[32];
    int            i;

    XtSetLanguageProc (NULL, NULL, NULL);
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv, NULL,
                                    sessionShellWidgetClass, NULL);
    mainform = XmCreateForm (toplevel, "mainform", NULL, 0);
    last_subform = NULL;
    for (i = 0; i < XtNumber (prompts); i++) {
        subform = XmCreateForm (mainform, "subform", NULL, 0);
        XtVaSetValues (subform,
            /* first one should be attached for form */
            XmNtopAttachment,   last_subform ? XmATTACH_WIDGET : XmATTACH_FORM,
            /* others are attached to the previous subform */
            XmNtopWidget,       last_subform,
            XmNleftAttachment,  XmATTACH_FORM,
            XmNrightAttachment, XmATTACH_FORM,
            NULL);
        /* Note that the label here contains a colon from the prompts
        ** array above.  This makes it impossible for external resources
        ** to be set on these widgets.  Here, that is intentional, but
        ** be careful in the general case.
        */
        label = XmCreateLabelGadget (subform, prompts[i], NULL, 0);
        XtVaSetValues (label,
                       XmNtopAttachment,     XmATTACH_FORM,
                       XmNbottomAttachment,  XmATTACH_FORM,
                       XmNleftAttachment,    XmATTACH_FORM,
                       XmNalignment,         XmALIGNMENT_BEGINNING,
                       NULL);
        XtManageChild (label);
        sprintf (buf, "text_%d", i);
```

* XtVaAppInitialize() is considered deprecated in X11R6.

```
            text = XmCreateText (subform, buf, NULL, 0);
            XtVaSetValues (text,
                           XmNtopAttachment,    XmATTACH_FORM,
                           XmNbottomAttachment, XmATTACH_FORM,
                           XmNrightAttachment,  XmATTACH_FORM,
                           XmNleftAttachment,   XmATTACH_WIDGET,
                           XmNleftWidget,       label,
                           NULL);
            XtManageChild (text);
            XtManageChild (subform);
            last_subform = subform;
        }
        /* Now that all the forms are added, manage the main form */
        XtManageChild (mainform);
        XtRealizeWidget (toplevel);
        XtAppMainLoop (app);
    }
```

The output of the program is shown in Example 8-5.

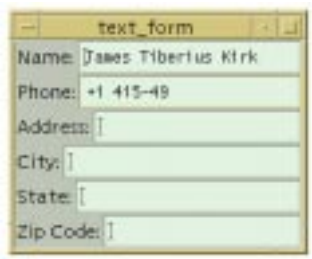


Figure 8-13 Output of the text_form program

Notice that the Labels are centered vertically with respect to their corresponding Text widgets. This arrangement happened because each Label was stretched vertically in order to attach it to the top and bottom of the respective Form. Of course, if the Labels were higher than the Text widgets, the Text widgets would be stretched instead.

Later, we'll show another version of this program that gives better results. As you can imagine, there are many different ways for a Form, or any other manager widget, to manage the geometry of its children to produce the same layout. Later, when we discuss the RowColumn widget, we will show you another solution to the problem of horizontal alignment. It is important to remember that there is no right or wrong way to create a layout, as long as it works for your application. However, you should be very careful to experiment with resizing issues as well as with resources that can beset by the user that might affect widget layout, such as fonts and strings.

Common Problems

With a Form widget, you can specify a virtually unlimited number of attachments for its children. The dependencies inherent in these attachments can lead to various errors in the layout of the widgets. One common problem involves circular dependencies. The following code fragment shows a very simple example of a circular dependency:

```
w1 = XmCreateLabel (form, "w1", NULL, 0);
w2 = XmCreateLabel (form, "w2", NULL, 0);
XtVaSetValues (w1, XmNrightAttachment, XmATTACH_WIDGET,
                   XmNrightWidget, w2,
                   NULL);
XtVaSetValues (w2, XmNleftAttachment, XmATTACH_WIDGET,
                   XmNleftWidget, w1,
                   NULL);
```

In this example, the left widget is attached to the right widget and the right widget is attached to the left one. If you do mistakenly specify a circular dependency, it is unlikely that it will be as obvious as this example. Fortunately, in most cases, the Motif toolkit catches circular dependencies and displays an error message if one is found. When this situation occurs, you need to reconsider your widget layout and try to arrange things such that the relationship between widgets is less complex. One rule to remember is that adjacent widgets should generally only be attached in one direction.

When you attach the side of a widget to another widget in a Form, you need to be careful about how you specify the attached widget. If you specify this widget in the application code, you need to make sure that the widget has been created before you specify it as a resource value. Alternatively, the toolkit provides a name-to-widget converter, so you can also specify widget IDs in a resource file. (See Volume 4, *X Toolkit Intrinsics Programming Manual* for information about resource converters.)

Another common problem arises with certain Motif compound objects, such as ScrolledList and ScrolledTextobjects. `XmCreateScrolledText()` and `XmCreateScrolledList()` return the corresponding Text or List widget, but it is the parent of this widget that needs to be positioned within a Form. The following code fragment shows an example of positioning a ScrolledList incorrectly:

```
form = XmCreateForm (parent, "form", NULL, 0);
list = XmCreateScrolledList (form, "scrolled_list", NULL, 0);
XtVaSetValues ( list,  /* <- WRONG */
                XmNleftAttachment, XmATTACH_FORM,
                XmNtopAttachment,  XmATTACH_FORM,
                NULL);
```

Since the List is a child of the ScrolledWindow, not the Form, specifying attachments for the List has no effect on the position of the List in the Form. The attachments need to be specified on the ScrolledWindow, as shown in the following code fragment:

```
XtVaSetValues ( XtParent (list),
                XmNleftAttachment, XmATTACH_FORM,
                XmNtopAttachment,  XmATTACH_FORM,
                NULL);
```

If you specify attachments for two opposing sides of a widget, the Form resizes the widget as needed, so that the default size of the widget is ignored. In most cases, the Form can resize the widget without a problem. However, one particular case that can cause a problem is a List widget that has its `XmNvisibleItemCount` resource set. This resource implies a specific size requirement, so that when the List is laid out in the Form widget, the negotiation process between the Form and the List may not be resolved. See Chapter 13, *The List Widget*, for a complete discussion of the List widget.

Attachments in Form widgets can be delicate specifications, which means that you must be specific and, above all, complete in your descriptions of how widgets should be aligned and positioned. Since resources can be set from many different places, the only way to guarantee that you get the layout you want is to hard-code these resource values explicitly. Even though it is important to allow the user to specify as many resources as possible, you do not want to compromise the integrity of your application. Attachments and attachment offsets are probably not in the set of resources that should be user-definable.

Although attachments can be delicate, they also provide a powerful, convenient, and flexible way to lay out widgets within a Form, especially when the widgets are grouped together in some abstract way. Attachments make it easy to chain widgets together, to bind them to the edge of a Form, and to allow them to be fixed on specific locations. You do not need to use a single attachment type exclusively; it is perfectly reasonable, and in most cases necessary, to use a variety of different types of attachments to achieve a particular layout. If you specify too few attachments, you may end up with misplaced widgets or widgets that drift when the Form is resized, while too many attachments may cause the Form to be too inflexible. In order to determine the best way to attach widgets to one another, you may find it helpful to a draw picture first, with all of the hooks and offset values considered.

The RowColumn Widget

The RowColumn widget is a manager widget that, as its name implies, lays out its children in a row and/or column format. The widget is also used internally by the Motif toolkit to implement a number of special objects, such as the Motif menus, including PopupMenus, PulldownMenus, MenuBars, and OptionMenus. Many of the resources for the RowColumn widget are used to control different aspects of these objects. The Motif convenience functions for creating these objects set most of these resources automatically, so they are generally hidden from the programmer. The resources are not useful when you are using the RowColumn as a simple manager widget anyway, so we do not discuss them here. The header file <*Xm/RowColumn.h*> is required if you are using this widget.

The XmNrowColumnType resource controls how a particular instance of the RowColumn is used. The resource can be set to the following values:

XmWORK_AREA	XmMENU_BAR	XmPULLDOWN
XmMENU_POPUP	XmMENU_OPTION	

The default value is XmWORK_AREA; this value is also the one that you should use whenever you want to use a RowColumn widget as a manager. The rest of the values are for the different types of Motif menus. If you want to create a particular menu object, you should use the appropriate convenience function, rather than try to create the menu yourself using a RowColumn directly. We discuss menu creation in Chapter 4, *The Main Window*, and Chapter 20, *Interacting With the Window Manager*. The RowColumn widget is also used to implement RadioBoxes and CheckBoxes, which are collections of ToggleButtons. See Chapter 12, *Labels and Buttons*, for more information on these objects.

The RowColumn is useful for generic geometry management because it requires less fine tuning than is necessary for a Form or a BulletinBoard widget. Although the RowColumn has a number of resources, you can create a usable layout without specifying any resources. In this case, the children of the RowColumn are automatically laid out vertically. In Example 8-6, we create several PushButtons as children of a RowColumn, without specifying any RowColumn resources.*

Example 8-6 The rowcol.c program

```
/* rowcol.c -- demonstrate a simple RowColumn widget. Create one
** with 3 pushbutton gadgets.  Once created, resize the thing in
** all sorts of contortions to get a feel for what RowColumns can
** do with its children.
*/
#include <Xm/PushB.h>
#include <Xm/RowColumn.h>

main (int argc, char *argv[])
{
    Widget         toplevel, rowcol, button;
    XtAppContext   app;

    XtSetLanguageProc (NULL, NULL, NULL);
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv, NULL,
                                    sessionShellWidgetClass, NULL);
    rowcol = XmCreateRowColumn (toplevel, "rowcolumn", NULL, 0);
    button = XmCreatePushButton (rowcol, "One", NULL, 0);
    XtManageChild (button);
    button = XmCreatePushButton (rowcol, "Two", NULL, 0);
    XtManageChild (button);
    button = XmCreatePushButton (rowcol, "Three", NULL, 0);
    XtManageChild (button);
    XtManageChild (rowcol);
```

* XtVaAppInitialize() is considered deprecated in X11R6.

```
        XtRealizeWidget (toplevel);
        XtAppMainLoop (app);
    }
```

What makes the RowColumn widget unique is that it automates much of the process of widget layout and management. If you display the application and resize it in a number of ways, you can get a better feel for how the RowColumn works. Figure 8-14 shows a few configurations of the application; the first configuration is the initial layout of the application. As you can see, if the application is resized just so, the widgets are oriented horizontally rather than vertically.

Figure 8-14 Output of the rowcol program

The orientation of the widgets in a RowColumn is controlled by the `XmNorientation` resource. The default value of the resource is `XmVERTICAL`. If we want to arrange the widgets horizontally, we can set the resource to `XmHORIZONTAL`. The orientation can be hard-coded in the application, or we can specify the value of the resource in a resource file. The following resource specification sets the orientation to horizontal:

```
*RowColumn.orientation: horizontal
```

Alternatively, we can specify the resource on the command line as follows:

```
% rowcol -xrm "*orientation: horizontal"
```

Figure 8-15 shows the output of Example 8-6 with a horizontal orientation. As before, the figure shows a few different configurations of the application, with the first configuration being the initial one.

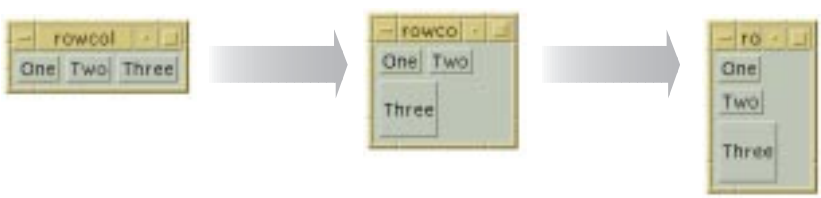


Figure 8-15 Output of the rowcol program with a horizontal orientation

If you use a RowColumn widget to manage more objects than can be arranged in a single row or column, you can specify that the widgets should be arranged in both rows and columns. You can also specify whether the widgets should be packed together tightly, so that the rows and columns are not necessarily the same size, or whether the objects should

be placed in identically-sized boxes. As with the Form and BulletinBoard widgets, objects can also be placed at specific x,y locations in a RowColumn widget. The RowColumn widget does not provide a three-dimensional border, so if you want to provide a visual border for the widget, you should create it as a child of a Frame widget.

Rows and Columns

The RowColumn widget can be quite flexible in terms of how it lays out its children. The advantage of this flexibility is that all of its child widgets are arranged in an organized fashion, regardless of their widget types. The widgets remain organized when the RowColumn is resized and in spite of constraints imposed by other widgets or by resources. One disadvantage of the flexibility is that sometimes the children need to be arranged in a specific layout so that the user interface is intuitive.

Example 8-7 shows how to lay out widgets in a spreadsheet-style format using a RowColumn. This layout requires that each of the widgets be the same size and be spaced equally in a predetermined number of rows and columns.*

Example 8-7 The spreadsheet.c program

```
/* spreadsheet.c -- This demo shows the most basic use of the RowColumn
** It displays a table of widgets in a row-column format similar to a
** spreadsheet. This is accomplished by setting the number ROWS and
** COLS and setting the appropriate resources correctly.
*/
#include <Xm/LabelG.h>
#include <Xm/PushB.h>
#include <Xm/RowColumn.h>

#define ROWS  8
#define COLS 10

main (int argc, char *argv[])
{
    Widget         toplevel, parent, child;
    XtAppContext   app;
    char           buf[16];
    int            i, j;

    XtSetLanguageProc (NULL, NULL, NULL);
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv, NULL,
                                    sessionShellWidgetClass, NULL);
    parent = XmCreateRowColumn (toplevel, "rowcolumn", NULL, 0);
    XtVaSetValues (parent, XmNpacking,     XmPACK_COLUMN,
                           XmNnumColumns,  COLS,
                           XmNorientation, XmVERTICAL,
                           NULL);
    /* simply loop through the strings creating a widget for each one */
```

* XtVaAppInitialize() is considered deprecated in X11R6.

```
        for (i = 0; i < COLS; i++)
            for (j = 0; j < ROWS; j++) {
                sprintf (buf, "%d-%d", i+1, j+1);
                if (i == 0 || j == 0)
                    child = XmCreateLabelGadget (parent, buf, NULL, 0);
                else
                    child = XmCreatePushButton (parent, "", NULL, 0);
                XtManageChild (child);
            }

        XtManageChild (parent);
        XtRealizeWidget (toplevel);
        XtAppMainLoop (app);
    }
```

The output of this example is shown in Figure 8-16.

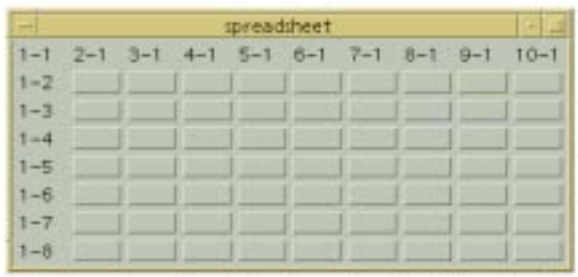


Figure 8-16 Output of the spreadsheet program

The number of rows is specified by the ROWS definition and the number of columns is specified by COLS. In order to force the RowColumn to lay out its children in the spreadsheet format, we set the XmNpacking, XmNnumColumns, and XmNorientation resources.

The value for XmNpacking is set to XmPACK_COLUMN, which specifies that each of the cells should be the same size. The heights and widths of the widgets are evaluated and the largest height and width are used to determine the size of the rows and columns. All of the widgets are resized to this size. If you are mixing different widget types in a RowColumn, you may not want to use XmPACK_COLUMN because of size variations. XmPACK_COLUMN is typically used when the widgets are exactly the same, or at least similar in nature. The default value of XmPACK_TIGHT for XmNpacking allows each widget to keep its specified size and packs the widgets into rows and columns based on the size of the RowColumn widget.

Since we are packing the widgets in a row/column format, we need to specify how many columns (or rows) we are using by setting the value of XmNnumColumns to the number of columns. In this case, the program defines COLS to be 10, which indicates that the

RowColumn should pack its children such that there are 10 columns. The widget creates as many rows as necessary to provide enough space for all of the child widgets.

Whether XmNnumColumns specifies the number of columns or the number of rows depends on the orientation of the RowColumn. In this program, XmNorientation is set to XmVERTICAL to indicate that the value of XmNnumColumns specifies the number of columns to use. If XmNorientation is set to XmHORIZONTAL, XmNnumColumns indicates the number of rows. If we wanted to use a horizontal orientation in our example, we would set XmNnumColumns to ROWS and XmNorientation to XmHORIZONTAL. The orientation also dictates how children are added to the RowColumn; when the orientation is vertical, children are added vertically so that each column is filled up before the next one is started.*

In our example, we explicitly set the value of XmNorientation to the default value of XmVERTICAL. If we do not hard-code this resource, an external resource specification can reset it. Since the orientation and the value for XmNnumColumns need to be consistent, you should always specify these resources together. Whether you choose to hard-code the resources, to use the fallback mechanism, or to use a specification in a resource file, you should be sure that both of the resources are specified in the same place.

In the spreadsheet example, we can use either a horizontal or vertical orientation. However, orientation may be significant in other situations, since it affects how the RowColumn adds its children. For example, if we want to implement the text-entry form from Example 8-5 using a RowColumn, the order of the widgets is important. In this case, there are two columns and the number of rows depends on the number of text entry fields provided by the application. We specify the orientation of the RowColumn as XmHORIZONTAL and set XmNnumColumns to the number of entries provided by the application, as shown in Example 8-8.†

Example 8-8 The text_entry.c program

```
/* text_entry.c -- This demo shows how the RowColumn widget can be
** configured to build a text entry form.  It displays a table of
** right-justified Labels and Text widgets that extend to the right
** edge of the Form.
*/
#include <Xm/LabelG.h>
#include <Xm/RowColumn.h>
#include <Xm/Text.h>

char *text_labels[] = {"Name:", "Phone:", "Address:",
                       "City:", "State:", "Zip Code:"};
```

* If you need to insert a child in the middle of an existing RowColumn layout, you can use the XmNpositionIndex constraint resource to specify the position of the child. Since this resource is used most often with menus, it is discussed in Chapter 19, *Menus*.

† XtVaAppInitialize() is considered deprecated in X11R6.

```
main (int argc, char *argv[])
{
    Widget       toplevel, rowcol, child;
    XtAppContext app;
    char         buf[8];
    int          i;

    XtSetLanguageProc (NULL, NULL, NULL);
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv,
                                NULL,sessionShellWidgetClass, NULL);
    rowcol = XmCreateRowColumn (toplevel, "rowcolumn",  NULL, 0);
    XtVaSetValues (rowcol, XmNpacking,        XmPACK_COLUMN,
                           XmNnumColumns,     XtNumber (text_labels),
                           XmNorientation,    XmHORIZONTAL,
                           XmNisAligned,      True,
                           XmNentryAlignment, XmALIGNMENT_END,
                           NULL);
    /* simply loop through the strings creating a widget for each one */
    for (i = 0; i < XtNumber (text_labels); i++) {
        child = XmCreateLabelGadget (rowcol, text_labels[i],NULL, 0);
        XtManageChild (child);
        sprintf (buf, "text_%d", i);
        child = XmCreateText (rowcol, buf, NULL, 0);
        XtManageChild (child);
    }
    XtManageChild (rowcol);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}
```

The output of this example is shown in Figure 8-17.

Figure 8-17 Output of the text_entry program

The labels for the text fields are initialized by the `text_labels` string array. When the RowColumn is created, it is set to a horizontal orientation and the number of rows is set to the number of items in `text_labels`. As you can see, the output of this program is slightly different from the output for the `text_form` example.

The example uses the `XmNisAligned` and `XmNentryAlignment` resources to control the positioning of the Labels in the RowColumn.These resources control the alignment of

widgets that are subclasses of Label and LabelGadget. When XmNisAligned is True (the default), the alignment is taken from the XmNentryAlignment resource. The possible alignment values are the same as those that can be set for the Label's XmNalignment resource:

```
XmALIGNMENT_BEGINNING        XmALIGNMENT_CENTER        XmALIGNMENT_END
```

By default, the text is left justified. While the alignment of the Labels could also be specified using the XmNalignment resource for each widget, it is convenient to be able to set the alignment for the RowColumn and have it propagate automatically to its children. In our example, we use XmALIGNMENT_END to right justify the Labels so that they appear to be attached to the Text widgets.

There is an additional resource for controlling the alignment of various children. The XmNentryVerticalAlignment resource controls the vertical positioning of children that are subclasses of Label, LabelGadget, and Text. The possible values for this resource are:

```
XmALIGNMENT_BASELINE_BOTTOM          XmALIGNMENT_BASELINE_TOP
XmALIGNMENT_CENTER                   XmALIGNMENT_CONTENTS_BOTTOM
XmALIGNMENT_CONTENTS_TOP
```

In the example, we do not specify this resource because the default value, XmALIGNMENT_CENTER, produces the layout that we want.

Homogeneous Children

The RowColumn can be set up so that it only manages one particular type of widget or gadget. In many cases, this feature facilitates layout and callback management. For example, a MenuBar consists entirely of CascadeButtons that all act the same way and a RadioBox contains only ToggleButtons. The XmNisHomogeneous resource indicates whether or not the RowColumn should only allow one type of widget child. The widget class that is allowed to be managed is specified by the XmNentryClass resource. XmNisHomogeneous can be set at creation-time only. Once a RowColumn is created, you cannot reset this resource, although you can always get its value. These resources are useful for ensuring consistency; if you attempt to add a widget as a child of a RowColumn that does not permit that widget class, an error message is printed and the widget is not accepted.

The Motif toolkit uses these mechanisms to ensure consistency in certain compound objects, to prevent you from doing something like adding a List widget to a MenuBar, for example. In this case, the XmNentryClass is set to xmCascadeButtonWidgetClass. As another example, when XmNradioBehavior is set, the RowColumn only allows ToggleButton widgets and gadgets to be added. The XmCreateRadioBox() convenience function creates a RowColumn widget with the appropriate resources set automatically. (See Chapter 12, *Labels and Buttons*.)

You probably do not need to use XmNisHomogeneous unless you are providing a mechanism that is exported to other programmers. If you are writing an interactive user-interface builder or a program that creates widgets by scanning text files, you may want to ensure that new widgets are of a particular type before they are added to a RowColumn widget. In such cases, you may want to use XmNisHomogeneous and XmNentryClass. Unless there is some way for a user to dynamically create widgets while an application is running, these resources are not particularly useful.

Callbacks

The RowColumn does not provide any specific callback routines that react to user input. While there are no callbacks for FocusIn and FocusOut events, the widget does have XmNmapCallback and XmNunmapCallback callback resources. These callbacks are invoked when the window for the RowColumn is mapped and unmapped. The callbacks are similar to those for the BulletinBoard, but since the RowColumn is not designed specifically to be a child of a DialogShell, the routines are invoked regardless of whether the parent of the RowColumn is a DialogShell.

The XmNentryCallback is the only other callback that is associated specifically with the RowColumn widget. This callback resource makes it possible to install a single callback function that acts as the activation callback for each of the children of a RowColumn widget. The routine specified for the XmNentryCallback overrides the XmNactivateCallback functions for any PushButton or CascadeButton children and the XmNvalueChangedCallback functions for ToggleButtons. The XmNentryCallback is a convenience to the programmer; if you use it, you don't have to install separate callbacks for each widget in the RowColumn. XmNentryCallback functions must be installed before children are added to the RowColumn, so be sure you call XtAddCallback() before you create any child widgets.

The callback procedure takes the standard form of an XtCallbackProc. The call_data parameter is an XmRowColumnCallbackStruct, which is defined as follows:

```
typedef struct {
    int        reason;
    XEvent     *event;
    Widget     widget;
    char       *data;
    char       *callbackstruct;
} XmRowColumnCallbackStruct;
```

The reason field of this data structure is set to XmCR_ACTIVATE when the XmNentryCallback is invoked. The event indicates the event that caused the notification. The entry callback function is called regardless of which widget within the RowColumn was activated. Since an entry callback overrides any previously-set callback lists for PushButtons, CascadeButtons, and ToggleButtons, the parameters that would have been passed to these callback routines are provided in the RowColumn callback structure.

The `widget` field specifies the child that was activated, the widget-specific callback structure is placed in the `callbackstruct` field, and the client data that was set for the widget is passed in the `data` field.

Example 8-9shows the installation of an entry callback and demonstrates how the normal callback functions are overridden.*

Example 8-9 The entry_cb.c program

```
/* entry_cb.c -- demonstrate how the XmNentryCallback resource works
** in RowColumn widgets. When a callback function is set for this
** resource, all the callbacks for the RowColumn's children are reset
** to point to this function.  Their original functions are no longer
** called had they been set in favor of the entry-callback function.
*/
#include <Xm/PushBG.h>
#include <Xm/RowColumn.h>

char *strings[] = {"One", "Two", "Three", "Four", "Five", "Six", "Seven",
                   "Eight", "Nine", "Ten"};

void called (Widget widget, XtPointer client_data, XtPointer call_data)
{
    XmRowColumnCallbackStruct *cbs =
                            (XmRowColumnCallbackStruct *) call_data;
    Widget pb = cbs->widget;
    printf ("%s: %d\n", XtName (pb), cbs->data);
}

static void never_called (Widget widget, XtPointer client_data,
                          XtPointer call_data)
{
    puts ("This function is never called");
}

main (int argc, char *argv[])
{
    Widget         toplevel, parent, w;
    XtAppContext   app;
    int            i;

    XtSetLanguageProc (NULL, NULL, NULL);
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv, NULL,
                                    sessionShellWidgetClass, NULL);
    parent = XmCreateRowColumn (toplevel, "rowcolumn", NULL, 0);
    XtAddCallback (parent, XmNentryCallback, called, NULL);

    /* simply loop through the strings creating a widget for each one */
    for (i = 0; i < XtNumber (strings); i++) {
        w = XmCreatePushButtonGadget (parent, strings[i], NULL, 0);
        /* Call XtAddCallback() to install client_data only! */
```

* `XtVaAppInitialize`() is considered deprecated in X11R6.

```
            XtAddCallback (w, XmNactivateCallback, never_called,
                           (XtPointer) (i+1));
            XtManageChild (w);
        }
        XtManageChild (parent);
        XtRealizeWidget (toplevel);
        XtAppMainLoop (app);
    }
```

The RowColumn is created and its `XmNentryCallback` is set to `called()`. This routine ignores the `client_data` parameter, as none is provided. However, we do use the `data` field of the callback structure because this is the data that is specified in the call to `XtAddCallback()` for each of the children. We install the `never_called()` routine for each PushButton and pass the position of the button in the RowColumn as the `client_data`. Even though the entry callback overrides the activate callback, the `client_data` is preserved.

Our example is a bit contrived, so it may seem pointless to call `XtAddCallback()` for each PushButton and specify an `XmNentryCallback` as well. The most compelling reason for using an entry callback is that you may want to provide client data for the RowColumn as a whole, as well as for each child widget.

Remember that the RowColumn widget is also used for a number of objects implemented internally by the Motif toolkit, such as the Motif menu system, RadioBoxes, and CheckBoxes. Many of the resources for the widget are specific to these objects, so they are not discussed here. For more information on menus, see Chapter 4, *The Main Window*, and Chapter 19, *Menus*; for information on RadioBoxes and CheckBoxes, see Chapter 12, *Labels and Buttons*.

The Frame Widget

The Frame is a simple manager widget; the purpose of the Frame is to draw a three-dimensional border around its child. The widget can have two children: a work area child and a title child. The Frame shrink wraps itself around its work area child, adding space for a title if one is specified. The children are responsible for setting the size of the Frame.

The Frame is useful for grouping related control elements, so that they are separated visually from other elements in a window. The Frame is commonly used as the parent of RadioBoxes and CheckBoxes, since the RowColumn widget does not provide a three-dimensional border. Figure 8-18 shows a portion of a dialog box that uses Frames to segregate three groups of ToggleButtons.

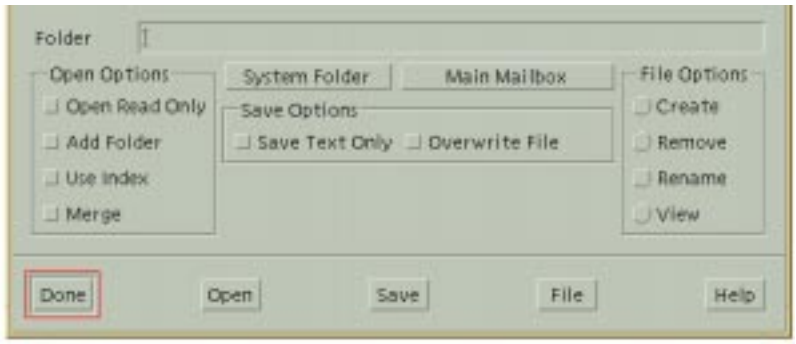


Figure 8-18 Frame widgets used to provide borders

To use Frame widgets in an application, you must include the file <*Xm/Frame.h*>. Creating a Frame widget is just like creating any other manager widget, as shown in the following code fragment:

```
Widget frame = XmCreateFrame (parent, "name", resource-value-array,
                              resource-value-count);
```

Since the Frame performs only simple geometry management, you can also create a Frame widget as managed using `XtVaCreateManagedWidget()` and not worry too much about a performance loss. The Frame widget is an exception to the guidelines about creating manager widgets that we presented earlier in the chapter.

The principal resource used by the Frame widget is `XmNshadowType`. This resource specifies the style of the three-dimensional border that is placed around the work area child of the Frame. The value may be any of the following:

`XmSHADOW_IN`	`XmSHADOW_OUT`
`XmSHADOW_ETCHED_IN`	`XmSHADOW_ETCHED_OUT`

If the parent of the Frame is a shell widget, the default value for XmNshadowType is set to `XmSHADOW_OUT` and the value for `XmNshadowThickness` is set to 1. Otherwise, the default shadow type is `XmSHADOW_ETCHED_IN` and the thickness is 2. Of course, these values may be overridden by the application or the user.

The Frame provides some constraint resources that can be specified for its children. The `XmNframeChildType`* resource indicates whether the child is the work area or the title child for the Frame. The default value is `XmFRAME_WORKAREA_CHILD`. To specify that a child is the title child, use the value `XmFRAME_TITLE_CHILD`.

* `XmNchildType` is deprecated as of Motif 2.0.

The XmNchildHorizontalAlignment and XmNchildHorizontalSpacing resources control the horizontal positioning of the title. The possible values for horizontal alignment are:

```
XmALIGNMENT_BEGINNING        XmALIGNMENT_END          XmALIGNMENT_CENTER
```

The XmNchildVerticalAlignment resource specifies the vertical positioning of the title child relative to the top shadow of the Frame. The possible values for this resource are:

```
XmALIGNMENT_BASELINE_BOTTOM              XmALIGNMENT_BASELINE_TOP
XmALIGNMENT_CENTER                       XmALIGNMENT_WIDGET_TOP
XmALIGNMENT_WIDGET_BOTTOM
```

Example 8-10 demonstrates many of the different shadow and alignment styles that are possible with the Frame widget.*

Example 8-10 The frame.c program

```
/* frame.c -- demonstrate the Frame widget by creating
** four Labels with Frame widget parents.
*/
#include <Xm/LabelG.h>
#include <Xm/RowColumn.h>
#include <Xm/Frame.h>

main (int argc, char *argv[])
{
    Widget         toplevel, rowcol, frame, label;
    XtAppContext   app;
    Arg            args[10];
    int            n;

    XtSetLanguageProc (NULL, NULL, NULL);
    /* Initialize toolkit and create TopLevel shell widget */
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv, NULL,
                                    sessionShellWidgetClass, NULL);
    /* Make a RowColumn to contain all the Frames */
    n = 0;
    XtSetArg (args[n], XmNspacing, 5); n++;
    rowcol = XmCreateRowColumn (toplevel, "rowcolumn",  args, n);

    /* Some informative labelling */
    label = XmCreateLabelGadget (rowcol, "Frame Types:", NULL, 0);
    XtManageChild (label);

    /* Create different Frames each containing a unique shadow type */
    /* First frame: Shadow in */
    n = 0;
    XtSetArg (args[n], XmNshadowType, XmSHADOW_IN); n++;
    frame = XmCreateFrame (rowcol, "frame1",   args, n);
```

* XtVaAppInitialize() s considered deprecated in X11R6. XmNchildType is deprecated as of Motif 2.0, and XmNframeChildType is preferred.

```
label = XmCreateLabelGadget (frame, "XmSHADOW_IN", NULL, 0);
XtManageChild (label);
n = 0;
XtSetArg (args[n], XmNframeChildType, XmFRAME_TITLE_CHILD); n++;
XtSetArg (args[n], XmNchildVerticalAlignment, XmALIGNMENT_CENTER); n++;
label = XmCreateLabelGadget (frame, "XmALIGNMENT_CENTER", args, n);
XtManageChild (label);
XtManageChild (frame);

/* Second frame: Shadow out */
n = 0;
XtSetArg (args[n], XmNshadowType, XmSHADOW_OUT); n++;
frame = XmCreateFrame (rowcol, "frame2",  args, n);
label = XmCreateLabelGadget (frame, "XmSHADOW_OUT", NULL, 0);
XtManageChild (label);

n = 0;
XtSetArg (args[n], XmNframeChildType, XmFRAME_TITLE_CHILD); n++;
XtSetArg (args[n], XmNchildVerticalAlignment,
                                XmALIGNMENT_BASELINE_TOP); n++;
label = XmCreateLabelGadget (frame, "XmALIGNMENT_BASELINE_TOP",
                            args, n);
XtManageChild (label);
XtManageChild (frame);

/* Third frame: Etched in */
n = 0;
XtSetArg (args[n], XmNshadowType, XmSHADOW_ETCHED_IN); n++;
frame = XmCreateFrame (rowcol, "frame3", args, n);
label = XmCreateLabelGadget (frame, "XmSHADOW_ETCHED_IN",  NULL, 0);
XtManageChild (label);

n = 0;
XtSetArg (args[n], XmNframeChildType, XmFRAME_TITLE_CHILD); n++;
XtSetArg (args[n], XmNchildVerticalAlignment, XmALIGNMENT_WIDGET_TOP);
n++;
label = XmCreateLabelGadget (frame, "XmALIGNMENT_WIDGET_TOP", args, n);
XtManageChild (label);
XtManageChild (frame);

/* Fourth frame: Etched out */
n = 0;
XtSetArg (args[n], XmNshadowType, XmSHADOW_ETCHED_OUT); n++;
frame = XmCreateFrame (rowcol, "frame4",  args, n);
label = XmCreateLabelGadget (frame, "XmSHADOW_ETCHED_OUT", NULL, 0);
XtManageChild (label);

n = 0;
XtSetArg (args[n], XmNframeChildType, XmFRAME_TITLE_CHILD); n++;
XtSetArg (args[n], XmNchildVerticalAlignment,
                                XmALIGNMENT_WIDGET_BOTTOM); n++;
label = XmCreateLabelGadget (frame, "XmALIGNMENT_WIDGET_BOTTOM",
                            args, n);
XtManageChild (label);
```

```
        XtManageChild (frame);

        XtManageChild (rowcol);
        XtRealizeWidget (toplevel);
        XtAppMainLoop (app);
    }
```

The output of this example is shown in Figure 8-19.

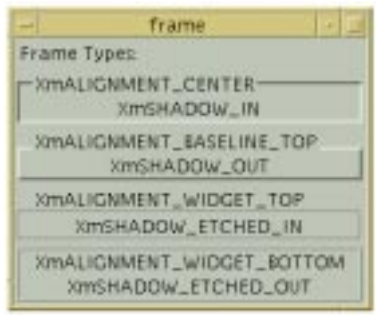


Figure 8-19 Output of the frame program

The program creates four Frame widgets. Each Frame has two Label children, one for the work area and one for the title. Each Frame uses a different value for the XmNshadowType and XmNchildVerticalPlacement resources, where these values are indicated by the text of the Labels. Although we have used a Label as the work area child of a Frame in this example, it is not a good idea to put a border around a Label. The shadow border implies selectability, which can confuse the user.

The PanedWindow Widget

The PanedWindow widget lays out its children in a horizontal or vertically-tiled format.* The idea behind the PanedWindow is that the user can adjust the individual panes to provide more or less space as needed on a per-child basis. For example, if the user wants to see more text in a Text widget, he can use the control sashes (sometimes called grips) to resize the area for the Text widget. For a horizontal PanedWindow, set the XmNorientation resource to XmHORIZONTAL; a vertical pane has XmNorientation set to XmVERTICAL, which is the default. When the user moves the sash, the widget above or below (or, for horizontal layout, to the left or right) the one being resized is resized smaller to compensate for the size change.

In a vertical layout, the width of the widget expands to that of its widest managed child and all of the other children are resized to match that width. The height of the PanedWindow is

* Horizontal orientation is available from Motif 2.0.

set to the sum of the heights of all of its children, plus the spacing between them and the size of the top and bottom margins.

In a horizontal layout, the height of the widget depends on the height of its tallest managed child. Similarly to the vertical case, the width of the PanedWindow depends upon the sum of the managed child widths, plus spacing and left/right margins.

By default, children are arranged in the PanedWindow in the order in which they are added. You can set the XmNpositionIndex constraint resource to control the position of a child in a PanedWindow if you do not want to use the default order.

An application that wants to use the PanedWindow widget must include the header file *<Xm/PanedW.h>*. An instance of the widget may be created as usual for manager widgets, as shown in the following code fragment:

```
Widget paned_w = XmCreatePanedWindow (parent, "name", resource-value-array,
                                      resource-value-count);
...
XtManageChild (paned_w);
```

The PanedWindow widget provides constraint resources that allow its children to indicate their preferred maximum and minimum sizes. Example 8-11 shows a pair of PanedWindows, each containing three widgets. One PanedWindow is in a vertical orientation, the other is horizontal.*

Example 8-11 The paned_win1.c program

```
/* paned_wind1.c -- two possible orientations of a PanedWindow.
** In the vertical PanedWindow, there are two Label widgets that are
** positioned above and below a Text widget.
** The Labels' minimum and maximum
** sizes are set to 25 and 45 respectively, preventing those
** panes from growing beyond those bounds.  The Text widget has its
** minimum size set to 35 preventing it from becoming so small that
** its text cannot be read.
** In the horizontal orientation, the Label’s minimum are set to 60
** to prevent the label from being truncated
*/
#include <Xm/Label.h>
#include <Xm/PanedW.h>
#include <Xm/Text.h>

Widget CreatePaneGroup (Widget parent, unsigned char orientation)
{
    Widget     pane, child;
    XmString   xms;
    Arg        args[6];
    int        n = 0;
```

* XtVaAppInitialize() is considered deprecated in X11R6. The PanedWindow resource XmNorientation is only properly supported from Motif 2.0 onwards.

```
    XtSetArg (args[n], XmNorientation, orientation); n++;
    pane = XmCreatePanedWindow (parent, "pane", args, n);

    n = 0;
    if (orientation == XmVERTICAL) {
        XtSetArg (args[n], XmNpaneMinimum, 25); n++;
        XtSetArg (args[n], XmNpaneMaximum, 45); n++;
    }
    else {
        XtSetArg (args[n], XmNpaneMinimum, 60); n++;
    }
    child = XmCreateLabel (pane, "Hello", args, n);
    XtManageChild (child);

    n = 0;
    XtSetArg (args[n], XmNpaneMinimum, 35); n++;
    XtSetArg (args[n], XmNrows, 5); n++;
    XtSetArg (args[n], XmNcolumns, 80); n++;
    XtSetArg (args[n], XmNvalue,
                       "This is a test of the PanedWindow widget"); n++;
    XtSetArg (args[n], XmNeditMode, XmMULTI_LINE_EDIT); n++;
                           child = XmCreateText (pane, "text", args, n);
    XtManageChild (child);

    n = 0;
    if (orientation == XmVERTICAL) {
        XtSetArg (args[n], XmNpaneMinimum, 25); n++;
        XtSetArg (args[n], XmNpaneMaximum, 45); n++;
    }
    else {
        XtSetArg (args[n], XmNpaneMinimum, 60); n++;
    }
    child = XmCreateLabel (pane, "Goodbye", args, n);
    XtManageChild (child);

    return pane;
}
main (int argc, char *argv[])
{
    Widget        toplevel, pane;
    XtAppContext  app;

    XtSetLanguageProc (NULL, NULL, NULL);
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv, NULL,
                                    sessionShellWidgetClass, NULL);
    pane = CreatePaneGroup (toplevel, (argc > 1 ? XmHORIZONTAL : XmVERTICAL));
    XtManageChild (pane);
    XtRealizeWidget (toplevel);
    XtAppMainLoop(app);
}
```

In the vertical layout, the two Label widgets are positioned above and below a Text widget. The minimum and maximum sizes of the Labels are set to 25 and 45 pixels respectively,

using the resources XmNpaneMinimum and XmNpaneMaximum. No matter how the PanedWindow or any of the other widgets are resized, the two Labels cannot grow or shrink vertically beyond these bounds. The Text widget, however, only has a minimum size restriction, so it maybe resized as tall or as short as the user prefers, provided that it does not get smaller than the 35-pixel minimum. Similar constraints are placed on the Labels and Text in the horizontally oriented PanedWindow: the XmNpaneMinimum of each Label is set to 60 to prevent the user truncating the label string. The program creates the PanedWindow in the horizontal layout if the program parameter list is arbitrarily not empty (*argc* > 1). Figure 8-20 shows the horizontal and vertical configurations of this application.

Figure 8-20 Horizontal and vertical PanedWindows

Figure 8-21 shows how the PanedWindow behaves when a child is resized using the Sash.

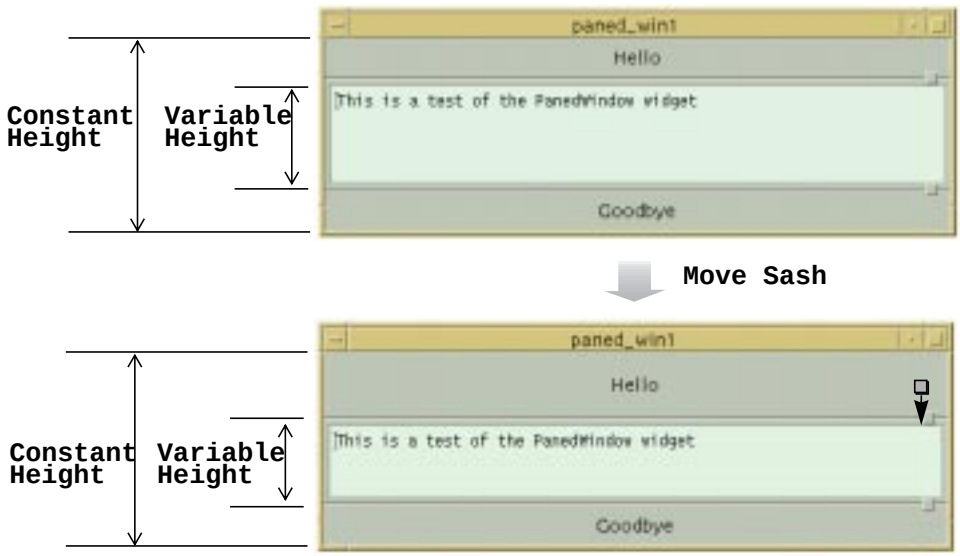


Figure 8-21 Output of paned_win1 program

Pane Constraints

One problem with setting the maximum and minimum resources for a widget involves determining exactly what those extents should be. The maximum size of 45 for the Label widgets in Example 8-11 is an arbitrary value that was selected for demonstration purposes only. If other resources had been set on one of the Labels such that the widget needed to be larger, the application would definitely look unbalanced. For example, an extremely high resolution monitor might require the use of unusually large fonts in order for text to appear normal. There are two choices available at this point. One is to specify the maximum and minimum values in a resolution-independent way and the other is to ask the Label widget itself what height it wants to be.

Specifying resolution-independent dimensions requires you to carefully consider the type of application you are creating. When you specify resolution-independent values, you must specify the values in either millimeters, inches, points, or font units. The value of the XmNunitType Manager resource controls the type of units that are used. Example 8-12 demonstrates the use of resolution-independent dimensions.*

Example 8-12 The unit_types.c program

```
/* unit_types.c --the same as paned_win1.c except that the
** Labels' minimum and maximum sizes are set to 1/4 inch and
```

* XtVaAppInitialize() is considered deprecated in X11R6.

```
** 1/2 inch respectively.  These measurements are retained
** regardless of the pixels-per-inch resolution of the user's
** display.
*/
#include <Xm/Label.h>
#include <Xm/PanedW.h>
#include <Xm/Text.h>

main (int argc, char *argv[])
{
    Widget         toplevel, pane, child;
    XtAppContext   app;
    Arg            args[6];
    int            n;

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

    n = 0;
    XtSetArg (args[n], XmNunitType, Xm1000TH_INCHES); n++;
    pane = XmCreatePanedWindow (toplevel, "pane", args, n);

    n= 0;
    XtSetArg (args[n], XmNpaneMinimum, 250); n++; /* quarter inch */
    XtSetArg (args[n], XmNpaneMaximum, 500); n++; /* half inch */
    child = XmCreateLabel (pane, "Hello",args, n);
    XtManageChild (child);

    n = 0;
    XtSetArg (args[n], XmNrows, 5); n++;
    XtSetArg (args[n], XmNcolumns, 80); n++;
    XtSetArg (args[n], XmNpaneMinimum, 250); n++; /* quarter inch */
    XtSetArg (args[n], XmNeditMode, XmMULTI_LINE_EDIT); n++;
    XtSetArg (args[n], XmNvalue,
    "This is a test of the paned window widget."); n++;
    child = XmCreateText (pane, "text", args, n);
    XtManageChild (child);

    n = 0;
    XtSetArg (args[n], XmNpaneMinimum, 250); n++; /* quarter inch */
    XtSetArg (args[n], XmNpaneMaximum, 500); n++; /* half inch */
    child = XmCreateLabel (pane, "Goodbye", args, n);
    XtManageChild (child);

    XtManageChild (pane);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}
```

The second technique that we can use is to query the Label widgets about their heights. This technique requires the use of the Xt function `XtQueryGeometry()`, as shown in Example 8-13.*

Example 8-13 The paned_win2.c program

```
/* paned_wind2.c --there are two label widgets that are positioned
** above and below a Text widget.  The labels' desired heights are
** queried using XtQueryGeometry() and their corresponding maximum
** and minimum sizes are set to the same value.  This effectively
** prevents those panes from being resized.  The Text widget has its
** minimum size set to 35 preventing it from becoming so small that
** its text cannot be read.
*/
#include <Xm/Label.h>
#include <Xm/PanedW.h>
#include <Xm/Text.h>

main (int argc, char *argv[])
{
    Widget              toplevel, pane, label, text;
    XtWidgetGeometry    size;
    XtAppContext        app;
    Arg                 args[8];
    int                 n;

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

    label = XmCreateLabel (pane, "Hello",  NULL, 0);
    XtManageChild (label);
    size.request_mode = CWHeight;
    XtQueryGeometry (label, NULL, &size);
    XtVaSetValues (label, XmNpaneMaximum, size.height,
                                    XmNpaneMinimum, size.height, NULL);
    printf ("hello's height: %d\n", size.height);

    n = 0;
    XtSetArg (args[n], XmNrows, 5); n++;
    XtSetArg (args[n], XmNcolumns, 80); n++;
    XtSetArg (args[n], XmNresizeWidth, False); n++;
    XtSetArg (args[n], XmNresizeHeight, False); n++;
    XtSetArg (args[n], XmNpaneMinimum, 35); n++;
    XtSetArg (args[n], XmNeditMode, XmMULTI_LINE_EDIT); n++;
    XtSetArg (args[n], XmNvalue, "This is a test of the paned widget."); n++;
    text = XmCreateText (pane, "text", args, n);
    XtManageChild (text);

    label = XmCreateLabel (pane, "Goodbye", NULL, 0);
    XtManageChild (label);
    size.request_mode = CWHeight;
    XtQueryGeometry (label, NULL, &size);
    XtVaSetValues (label, XmNpaneMaximum, size.height,  XmNpaneMinimum, size.
                   height, NULL);
```

* `XtVaAppInitialize`() is considered deprecated in X11R6.

```
    printf ("goodbye's height: %d\n", size.height);

    XtManageChild (pane);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}
```

`XtQueryGeometry()` asks a widget what size it would like to be. This routine takes the following form:

```
XtGeometryResult XtQueryGeometry ( Widget widget,
                                   XtWidgetGeometry *intended,
                                   XtWidgetGeometry *preferred_return)
```

Since we do not want to resize the widget, we pass NULL for the *intended* parameter. We are not interested in the return value of the function, since the information that we want is returned in the `preferred_return` parameter. This parameter is of type `XtWidgetGeometry`, which is defined as follows:

```
typedef struct {
    XtGeometryMask     request_mode;
    Position           x, y;
    Dimension          width, height, border_width;
    Widget             sibling;
    int                stack_mode;
} XtWidgetGeometry;
```

We tell the widget what we want to know by setting the `request_mode` field of the `size` variable that we pass to the routine. The `request_mode` field is checked by the `query_geometry` function within the called widget. Depending on which bits that are specified, the appropriate fields are set within the returned data structure. InExample 8-13, we set `request_mode` to `CWHeight`, which tells the Label widget's `query_geometry` method to return the desired height in the height field of the data structure. If we had wanted to know the width as well, we could have set `request_mode` as follows:

```
size.request_mode = (CWHeight | CWWidth);
```

In this case, the `width` and `height` fields would be filled in by the Label widget.

Once we have the Label's desired height, we can set the constraint resources `XmNpaneMaximum` and `XmNpaneMinimum` to the height of the Label. By making these two values the same, the pane associated with the Label cannot be resized. In most cases, the `XtQueryGeometry()` method can be used reliably to determine proper values for minimum and maximum pane extents. Setting extents is useful, since without them, the user can adjust a PanedWindow so that the size of a widget is unreasonable or unaesthetic. If you are setting the extents for a scrolled object (ScrolledText or ScrolledList), you do not need to be as concerned about the maximum extent, since these objects handle larger sizes appropriately. Minimum states are certainly legitimate though. For example, you could use the height of a font as a minimum extent for Text or a List.

The PanedWindow widget can be useful for building your own dialogs because you can control the size of the action area. The action area is always at the bottom of the dialog and its size should never be changed. See Chapter 7, *Custom Dialogs*, for a complete discussion of how a PanedWindow can be used in in this manner.

Sashes

The Sashes in a PanedWindow widget are in fact widgets, even though they are not described or defined publicly. While the *Motif Style Guide* says that the Sash is part of the PanedWindow widget, the Motif toolkit defines the object privately, which means that technically the Sash is not supported and it may change in the future. However, it is possible to get a handle to a Sash if you absolutely need one. In order to retrieve a Sash, you need to include the header file *<Xm/SashP.h>*. The fact that the file ends in an upper case *P* indicates that it is a private header file, which means that an application program should not include it. However, there is no public header file for the Sash widget, so unless you include the private header file, you cannot access the Sashes in a PanedWindow.

If you retrieve all of the children from a PanedWindow using `XtVaGetValues()` on the `XmNchildren` resource, you can use the `XmIsSash()` macro to locate the Sash children. This macro is defined as follows:

```
#define XmIsSash (w)    XtIsSubclass (w, xmSashWidgetClass)
```

Although `XtIsSubclass()` is a public function, `xmSashWidgetClass` is not declared publicly. One reason that you might want to get handles to the Sashes in a PanedWindow is to turn off keyboard traversal to the Sashes, as described in the next section.

Keyboard Traversal

The *Motif Style Guide* specifies methods by which the user can interact with an application without using the mouse. These methods provide a way for the user to navigate through an application and activate user-interface elements on the desktop using only the keyboard. Such activity is known as *keyboard traversal* and is based on the Common User Access (CUA) interface specifications from Microsoft Windows and Presentation Manager.

These specifications make heavy use of the TAB key to move between elements in a user interface; related interface controls are grouped into what are called *tab groups*. Some examples of tab groups are a set of ToggleButtons or a collection of PushButtons. Just as only one shell on the screen can have the keyboard focus, only one widget at a time can have the input focus. When keyboard activity occurs in a window, the toolkit knows which tab group is current and directs the input focus to the active item within that group.

The user can move from one item to the next within a tab group using the arrow keys. The user can move from one tab group to the next using the TAB key. To traverse the tab groups in the reverse direction, the user can use SHIFT-TAB. The CTRL key can be used with the

TAB key in a Text widget to differentiate between a traversal operation and the use of the TAB key for input. The SPACEBAR activates the item that has the keyboard focus.

To illustrate the keyboard traversal mechanisms, let's examine *tictactoe.c* from Example 8-4. This program contains one tab group, the Form widget. Because the PushButtons inside of it are elements in the tab group, the user can move between the items in the tic-tac-toe board using the arrow keys on the keyboard, as illustrated in Figure 8-22.

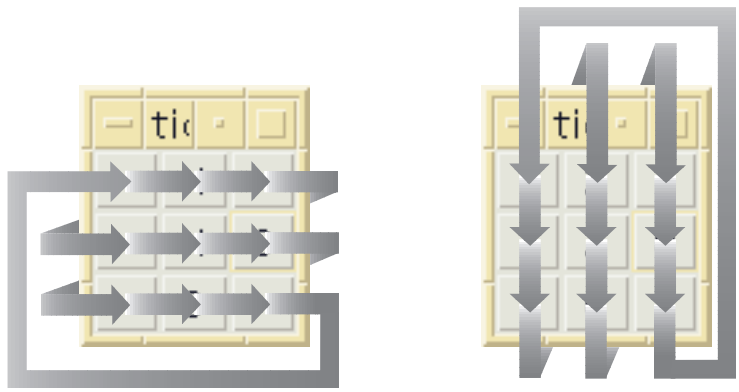


Figure 8-22 Keyboard traversal for the tictactoe program

Pressing the TAB key causes the input focus to be directed to the next tab group and set to the first item in the group, which is known as the *home* element. Since there is only one tab group in this application, the traversal mechanism moves the input focus to the first element in the same group. Thus, pressing the TAB key in this program always causes the home item to become the current input item.

The conceptual model of the tab group mechanism corresponds to the user's view of an application. With tab groups, the widget tree is flattened out into two simple layers: the first layer contains tab groups and the second layer contains the elements of those groups. In this model, there is no concept of managers and children or any sort of widget hierarchy. But as you know, an application is based on a very structured widget hierarchy. The implementation of tab groups is based on lists of widget pointers that refer to existing widgets in the widget tree. These lists, known as navigation groups, are maintained by the VendorShell and MenuShell widgets and are accessed by the input-handling mechanisms of the Motif toolkit.

Each widget class in the Motif toolkit is initialized either as a tab group itself or as a member of a tab group. Manager widgets, Lists, and Text widgets are usually tagged as tab groups, since they typically contain sub-elements that can be traversed. For example, the elements in a List can be traversed using the arrow keys on the keyboard; the up arrow moves the selection to the previous element in the List widget. In a Text widget, the arrow

keys move the insertion cursor. The other primitive widgets, such as PushButtons and ToggleButtons, are usually tagged as tab group members. Output-only widgets are not tagged at all and are excluded from the tab group mechanism, since you cannot traverse to an output-only widget.These default settings are not permanent. For example, a PushButton or a ToggleButton can be a tab group, although this setting is uncommon and should only be done when you have a special reason for forcing the widget to be recognized as a separate tab group.

When the TAB key is pressed, the next tab group in the list of tab groups becomes the current tab group. Since manager widgets are normally tab groups, the order of tab group traversal is typically based on the order in which the manager widgets are created. This entire process is automated by the Motif toolkit, so an application does not have to do anything unless it wants to use a different system of tab groups for some reason.In order to maintain Motif compliance, we recommend that you avoid interfering with the default behavior.

We are discussing keyboard traversal in the chapter on manager widgets because managers play the most visible role in keyboard traversal from the application programmer's perspective. Managers, by their nature, contain other widgets, which are typically primitive widgets that act as tab group members. Furthermore, manager widgets must handle all of the input events for gadgets, so there is a great deal of functionality that supports keyboard traversal written into the Manager widget class.

Before we discuss the details of dealing with tab groups, there are a few things we should mention. The implementation of tab groups has changed from earlier versions of the toolkit; to maintain backwards compatibility, remnants of the older implementation are still resident in the current implementation, which may cause some confusion in the current API. The technology of keyboard traversal is still being improved. Although later implementations may not change the existing API, new versions of the toolkit may optimize the process substantially. Since the current implementation of tab groups is not perfect, some people want to change the default behavior and control it entirely on their own. We do not recommend this approach. You should avoid interfering with the keyboard traversal mechanisms, as it will make it easier to maintain compatibility with other Motif applications and it won't require any changes for new versions of the toolkit. If you are going to modify the operation of keyboard traversal, you should be careful and test your changes thoroughly.

Turning Traversal Off

You can prevent a widget from participating in keyboard traversal by removing the widget from the traversal list. To remove a widget from the traversal list, set its `XmNtraversalOn` resource to `False`. If the widget is a member of a tab group, it is simply removed from the list and the user cannot traverse to it. If the widget is a tab group, it is removed and all of its elements are also all removed.

Let's experiment with tab group members by modifying `tictactoe.c`. We can modify the `pushed()` callback routine to remove the selected PushButton from the traversal list when it is selected. If the keyboard is used to traverse and select the items on the tic-tac-toe board, the toolkit automatically skips over those that have already been selected. The new callback routine is shown in Example 8-14.*

Example 8-14 The pushed() routine

```
void pushed (Widget w, XtPointer   client_data, XtPointer   call_data)
{
    char         buf[2];
    XmString     str;
    int          letter;

    XmPushButtonCallbackStruct *cbs =
                            (XmPushButtonCallbackStruct *) call_data;

    XtVaGetValues (w, XmNuserData, &letter, NULL);
    if (letter) {
        XBell (XtDisplayOfObject (w), 50);
        return;
    }
    /* Shift key gets an O.  (xbutton and xkey happen to be similar) */
    if (cbs->event->xbutton.state & ShiftMask)
        letter = buf[0] = '0';
    else
        letter = buf[0] = 'X';
    buf[1] = 0;
    str = XmStringCreateLocalized (buf);
    XtVaSetValues (w, XmNlabelString, str,
                      XmNuserData, letter,
                      XmNshadowThickness, 0,
                      XmNtraversalOn,False,
                      NULL);
    XmStringFree (str);
}
```

The user can still click on a previously-selected item with the mouse button, but the routine causes an error bell to sound in this situation.

Output-only widgets, like Labels and Separators, always have their `XmNtraversalOn` resource initialized to `False`. In most cases, setting the value to `True` would be annoying to the user, since these objects cannot respond to keyboard input anyway. The user would have to traverse many unimportant widgets to get to a desired item. However, it is commonly overlooked that a Label can have a `XmNhelpCallback` routine associated with it. If the keyboard traversal mechanism allows the user to traverse to Labels, he could get help on them by pressing the HELP or F1 keys. It may be considered a design flaw in Motif that a non-mouse-driven interface is not supported for getting help for these objects.

* `XtVaAppInitialize`() is considered deprecated in X11R6.

However, this situation is not generally a problem, since most people do not try to get help on Labels and most programmers do not install help for them.

A general problem that people tend to have with the PanedWindow widget is that the Sashes are included in the traversal list. Since the PanedWindow is a manager widget, it is a tab group, which means that all of its children are members of the tab group. If you run the program from Example 8-13 and use the TAB key to move from one widget to the next, you'll find that the traversal also includes the Sash widgets. Many users find it annoying to traverse to Sashes, since it is more likely that they want to skip the Sashes when using keyboard traversal, rather than use them to resize any of the panes. While it is common to resize panes, people usually do so using the mouse, not the keyboard.

It is possible to turn off Sash traversal using the following resource specification in a resource file:

```
*XmSash.traversalOn: False
```

There are some applications that might actually have to be used without a mouse, just as there are some users who prefer to use the keyboard, so you should be careful about turning off keyboard traversal for the Sashes in a PanedWindow widget. If you do turn off Sash traversal, we recommend that you document the behavior and provide a way for the user to control this behavior. For example, you could provide an application-specific resource that controls whether or not Sashes can be traversed using the keyboard.

As noted earlier, `XmNtraversalOn` can be set on tab groups (which tend to be manager widgets) as well as tab group members. If traversal is off for a tab group, none of its members can be traversed. If keyboard traversal is something that you need to modify in your application, you should probably hard-code `XmNtraversalOn` values directly into individual widgets as you create them. Turning off traversal is typically not something that is done on a per-widget-class basis. When you turn traversal off in application code, be careful to make sure that there is no reason that a user would want to traverse to the particular widgets because once you hard-code the resource values, they cannot be modified by the user in a resource file.

Modifying Tab Groups

The `XmNnavigationType` resource controls whether a widget is a tab group itself or is a member of a tab group. When this resource is set to `XmNONE`, the widget is not a tab group, so it defaults to being a member of one. As a member, its `XmNtraversalOn` resource indicates whether or not the user can direct the input focus to the widget using the keyboard. This value is the default for most primitive widgets. When the resource is set to `XmTAB_GROUP`, the widget is a tab group itself, so it is included in keyboard navigation. This value is the default for managers, Lists, and Text widgets. By modifying the default value of the `XmNnavigationType` resource for a widget, you can specify that a primitive widget is a tab group. As a result, the user traverses to the widget using the TAB key rather than one

of the arrow keys. For example, you can modify `tictactoe.c` by setting the `XmNnavigationType` to `XmTAB_GROUP` for each PushButton.

There are two other values for `XmNnavigationType` that are used for backwards compatibility with old Motif 1.0 versions of the toolkit. In this version of the toolkit, there is an application called `XmAddTabGroup()` to make a widget a tab group. Calling `XmAddTabGroup()` is equivalent to setting `XmNnavigationType` to `XmEXCLUSIVE_TAB_GROUP`. If this value is set on a widget, new widgets are no longer added as tab groups automatically. An exclusive tab group is much the same as a normal tab group, but Motif recognizes this special value and ignores all widgets that have the newer `XmTAB_GROUP` value set. You can think of this value as setting exclusivity on the tab group behavior.

The value `XmSTICKY_TAB_GROUP` can also be used for `XmNnavigationType`. If this value is used on a widget, the widget is included automatically in keyboard traversal, even if another widget has its navigation type set to `XmEXCLUSIVE_TAB_GROUP` or if `XmAddTabGroup()` has been called.

You can ignore these two values for all intents and purposes. You should use `XmNONE` and `XmTAB_GROUP` to control whether or not a widget is a tab group or a member of one. To control whether the widget is part of the whole keyboard traversal mechanism, use the `XmNtraversalOn` resource. `XmEXCLUSIVE_TAB_GROUP`, `XmSTICKY_TAB_GROUP`, and `XmAddTabGroup`() should be considered deprecated as far as the application programmer is concerned.

Handling Event Translations

In order for manager widgets to implement keyboard traversal, they have their own event translation tables that specify what happens when certain events occur. As discussed in Chapter 2, *The Motif Programming Model*, a translation table specifies a series of one or more events and an action that is invoked if the event occurs. The X Toolkit Intrinsics handles event translations automatically; when the user presses the TAB key, Xt looks up the event `<Key> Tab` in the table and invokes the corresponding action procedure. In this case, the procedure changes the input focus from the current tab group to the next one on the list.

This mechanism is dependent on the window hierarchy of the widget tree. Events are first delivered to the widget associated with the window where the event took place. If that widget (or its window) does not handle the type of event delivered, it passes the event up the window tree to its parent, which then has the option of dealing with the event. Assuming that the parent is a manager widget of some kind, it now has the option to process the event. If the event is a keyboard traversal event, the appropriate action routine moves the input focus. The default event translations that manager widgets use to handle keyboard traversal are currently specified as follows:

```
<Key>osfBeginLine:          ManagerGadgetTraverseHome()
<Key>osfUp:                 ManagerGadgetTraverseUp()
<Key>osfDown:               ManagerGadgetTraverseDown()
<Key>osfLeft:               ManagerGadgetTraverseLeft()
<Key>osfRight:              ManagerGadgetTraverseRight()
Shift ~Meta ~Alt <Key>Tab:  ManagerGadgetPrevTabGroup()
~Meta ~Alt <Key>Tab:        ManagerGadgetNextTabGroup()
<EnterWindow>:              ManagerEnter()
<LeaveWindow>:              ManagerLeave()
<FocusOut>:                 ManagerFocusOut()
<FocusIn>:                  ManagerFocusIn()
```

The OSF-specific keysyms are vendor-defined, which means that the directional arrows must be defined by the user's system at run-time. Values like `<Key>osfUp` and `<Key>osfDown` may not be the same as `<Key>Up` and `<Key>Down`.

The routines that handle keyboard traversal are prefixed by `ManagerGadget`. Despite their names, these functions are not specific to gadgets; they are used to handle keyboard traversal for all of the children in the manager. If a primitive widget inside of a manager widget specifies an event translation that conflicts with one of the manager's translations, the primitive widget can interfere with keyboard traversal. If the primitive widget has the input focus, the user cannot use the specified event to move the input focus with the keyboard. The following code fragment shows how the translation table for a PushButton can interfere with the keyboard traversal mechanism in its parent:

```
Widget                pb;
XtActionRec           action;
extern XtAppContext   app_context;
extern Widget         parent;
extern void           do_tab(Widget, XEvent *, String *, Cardinal *);

actions.string = "do_tab";
actions.proc = do_tab;
XtAppAddActions (app_context, &actions, 1);

pb = XmCreatePushButton (parent, "name", resource-value-array,
                         resource-value-coumt);
XtOverrideTranslations (pb, XtParseTranslationTable ("<Key>Tab: do_tab"));
```

The translation table is merged into the existing translations for the PushButton widget. This translation table does not interfere with the translation table in the manager widget, but it does interfere with event propagation to the manager. When the TAB key is pressed, the action routine `do_tab()` is called and the event is consumed by the PushButton widget. The event is not propagated up to the manager widget so that it can perform the appropriate keyboard traversal action. The work around for this problem is to have `do_tab()` process the keyboard traversal action on its own, in addition to performing its own action. This technique is discussed in the next section.

Since a manager can also contain gadgets, the manager widget must also handle input that is destined for gadgets. Since gadgets do not have windows, they cannot receive events.

Only the manager widget that is the parent of a gadget can receive events for the gadget. The manager widget has the following additional translations to handle input on behalf of gadgets:

```
<Key>osfActivate:                   ManagerParentActivate()
<Key>osfCancel:                     ManagerParentCancel()
<Key>osfSelect:                     ManagerGadgetSelect()
<Key>osfHelp:                       ManagerGadgetHelp()
~Shift ~Meta ~Alt <Key>Return:      ManagerParentActivate()
~Shift ~Meta ~Alt <Key>space:       ManagerGadgetSelect()
<Key>:                              ManagerGadgetKeyInput()
<BtnMotion>:                        ManagerGadgetButtonMotion()
<Btn1Down>:                         ManagerGadgetArm()
<Btn1Down>,<Btn1Up>:                ManagerGadgetActivate()
<Btn1Up>:                           ManagerGadgetActivate()
<Btn1Down>(2+):                     ManagerGadgetMultiArm()
<Btn1Up>(2+):                       ManagerGadgetMultiActivate()
<Btn2Down>:                         ManagerGadgetDrag()
```

Unlike with keyboard traversal translations, widget translations cannot interfere with the manager translations that handle events destined for gadgets. If a widget had the input focus, the user's actions cannot be destined for a gadget, since the user would have to traverse to the gadget first, in which case the manager would really have the input focus.

In Chapter 11, *The DrawingArea Widget*, we discuss the problems involved in handling input events on the DrawingArea widget. The problems arise because the widget can be used for interactive drawing, as well as serve as a manager. There may be events that you want to process in your application, but they could also be processed by the DrawingArea itself. The problem is really a semantic one, as there is no way to determine which action procedure should be invoked for each event if the DrawingArea has a manager-based action and the application defines its own action. For more information on translation tables and action routines, see Chapter 2, *The Motif Programming Model*, and Volume 4, *The X Toolkit Intrinsics Programming Manual*.

Processing Traversal Manually

At times, an application may want to move the input focus as a result of something that the user has done. For example, you might have an action area where each PushButton invokes a callback function and then sets the input focus to the home item in the tab group, presumably to protect the user from inadvertently selecting the same item twice. Example 8-15 demonstrates how this operation can be accomplished.*

Example 8-15The proc_traverse.c program

```
/* proc_traverse.c -- demonstrate how to process keyboard traversal
** from a PushButton's callback routine.  This simple demo contains
```

* XtVaAppInitialize() is considered deprecated in X11R6.

```
** a RowColumn (a tab group) and three PushButtons.  If any of the
** PushButtons are activated (selected), the input focus traverses
** to the "home" item.
*/

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

main (int argc, char *argv[])
{
    Widget         toplevel, rowcol, pb;
    XtAppContext   app;
    Arg            args[2];
    void           do_it(Widget, XtPointer, XtPointer);

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

    XtSetArg (args[0], XmNorientation, XmHORIZONTAL);
    rowcol = XmCreateRowColumn (toplevel, "rowcolumn",  args, 1);

    pb = XmCreatePushButton (rowcol, "OK", NULL, 0);
    XtManageChild (pb);

    pb = XmCreatePushButton (rowcol, "Cancel", NULL, 0);
    XtAddCallback (pb, XmNactivateCallback, do_it, NULL);
    XtManageChild (pb);

    pb = XmCreatePushButton (rowcol, "Help", NULL, 0);
    XtAddCallback (pb, XmNactivateCallback, do_it, NULL);
    XtManageChild (pb);

    XtManageChild (rowcol);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

/* callback for pushbuttons */
void do_it (Widget widget, XtPointer client_data, XtPointer call_data)
{
    /* do stuff here for PushButton widget */
    XmProcessTraversal(widget, XmTRAVERSE_HOME);
}
```

The three frames in Figure 8-23 show the movement of keyboard focus in the program. In the figure, the current input focus is on the *Cancel* button; when it is selected, the input focus is changed to the *OK* button.

Figure 8-23 Output of the proc_traversal program

The callback routine associated with the PushButtons does whatever it needs and then calls `XmProcessTraversal()` to change the input item to the home item, which happens to be the *OK* button. This function can be used when an application needs to set the current item in the tab group to another widget or gadget or it can be used to traverse to a new tab group. The function takes the following form:

```
Boolean XmProcessTraversal (Widget widget, int direction)
```

The function returns `False` if the VendorShell associated with the widget has no tab groups, the input focus policy doesn't make sense, or if there are other extenuating circumstances that would be considered unusual. It is unlikely that you'll ever have this problem.

The *`direction`* parameter specifies where the input focus should be moved. This parameter can take any of the following values:*

```
XmTRAVERSE_CURRENT                  XmTRAVERSE_NEXT
XmTRAVERSE_PREV                     XmTRAVERSE_HOME
XmTRAVERSE_UP                       XmTRAVERSE_DOWN
XmTRAVERSE_LEFT                     XmTRAVERSE_RIGHT

XmTRAVERSE_GLOBALLY_FORWARD         XmTRAVERSE_GLOBALLY_BACKWARD
XmTRAVERSE_NEXT_TAB_GROUP           XmTRAVERSE_PREV_TAB_GROUP
```

All but the last four values are for traversing to items within the current tab group; the last two are for traversing to the next or previous tab group relative to the current one. The values `XmTRAVERSE_GLOBALLY_FORWARD` and `XmTRAVERSE_GLOBALLY_BACKWARD` primarily exist in order to implement the XmDisplay object resource `XmNenableButtonTab`†. In this scheme, navigation using the arrow keys proceeds within a tab group as normal, except that if the end (or beginning, if traversing backwards) of the group is reached, navigation does not cycle round the group members but jumps to the next

* `XmTRAVERSE_GLOBALLY_FORWARD` and `XmTRAVERSE_GLOBALLY_BACKWARD` are available from Motif 2.0 onwards.

† `XmNenableButtonTab` is available from Motif 2.0 onwards.

(or preceding) tab group. The interpretation of "forwards" and "backwards" in a group depends upon the XmNlayoutDirection* resource: if this is XmRIGHT_TO_LEFT, the natural interpretation of "forwards" and "backwards" is reversed.

In Example 8-15, the call to XmProcessTraversal() forces the home element to be the current item in the current tab group. For a more sophisticated example of manipulating the input focus, see Chapter 18, *Text Widgets*. One problem with XmProcessTraversal() is that you can only move in a relative direction from the item that has the input focus. This functionality is sufficient in most cases, since the logic of your application should not rely on the user following any particular input sequence. If you need to traverse to a specific widget regardless of the current item, in most cases you can make the following call:

```
XmProcessTraversal (desired_widget, XmTRAVERSE_CURRENT);
```

This calling sequence specifies that the *desired_widget* takes the input focus, but only if the shell that contains the widget already has the keyboard focus. If the shell does not have the focus, nothing happens until the shell obtains the keyboard focus. When it does, the desired_widget should have the input focus.

Under certain conditions, this function may appear not to work. For example, if you create a dialog and want to set the input focus to one of its subwidgets, you may or may not get this to happen, depending on whether or not the dialog has been realized and mapped to the screen and whether or not keyboard focus has been accepted. Unfortunately, there is no general solution to this problem because the Motif toolkit isn't very robust about the programmer changing input focus out from under it. You cannot call generic X functions like XSetInputFocus() to force a widget to take input focus or you will undermine Motif's attempt at monitoring and controlling the input policy on its own.

There are some functions that make it easier for an application to try to control keyboard traversal. The XmGetFocusWidget() routine returns the widget that has the input focus, while XmGetTabGroup() returns the widget that is the tab group for the specified widget. You can also call XmIsTraversable() to determine whether or not a particular widget is eligible to receive the input focus.

Summary

Manager widgets are the backbone of an application. Without them, primitive widgets have no way of controlling their size, layout, and input focus. While the Motif toolkit provides many different manager widget classes, you may find that there are some things that you cannot do with them. Experienced toolkit programmers have found that it is possible to port Constraint class widgets from other toolkits to the Motif toolkit, by subclassing them from the generic Manager widget class. This topic is beyond the scope of this book.

* XmNlayoutDirection is available from Motif 2.0 onwards.

This chapter introduces the Motif manager widgets, but it does not discuss in detail some of the basic issues of geometry management. If the basic concepts presented in this chapter are still somewhat foreign to you, see Volume 4, *The X Toolkit Intrinsics Programming Manual*, for a more in-depth discussion of composite widgets and geometry management.

In this chapter:

- *Creating a Container*
- *Creating IconGadgets*
- *Container Resources*
- *IconGadget Resources*
- *Container Constraints*
- *Container Callbacks*
- *Container Functions*
- *Summary*
- *Exercises*

9

Containers and IconGadgets

The Container was possibly the most complex widget introduced into Motif 2. The Container is a Constraint Manager widget which is designed to work alongside the IconGadget. The idea is that IconGadgets are meant to pictorially represent application objects in some abstract way, and the Container lays them out in a variety of formats appropriate to the underlying data model. The Container can lay out its children in Tree, Multi-column Table, Grid, or Free-Floating formats; it is possible to dynamically switch layout in order to provide alternative presentations of the objects which it contains, and thus the widget is an approximation to a Model-View-Controller (MVC) component for the Motif widget set.

IconGadgets are derived from the Gadget class, as the name suggests. Although not derived from the LabelGadget class, they are similar to LabelGadgets in that they can display a Label string and an image Pixmap, the difference being that the IconGadget can display these simultaneously. It is also possible to associate with an IconGadget an array of compound strings which represents additional information about the object. This additional information is known as Detail, and the Container widget knows how to lay the Detail of its various IconGadget children out so that it is all arranged in columns. Each column may be assigned a heading label, much like a Table. Figure 9-1 shows a Container widget with IconGadget children laid out in the Detail style. The column headers are simply specified through Container resources.

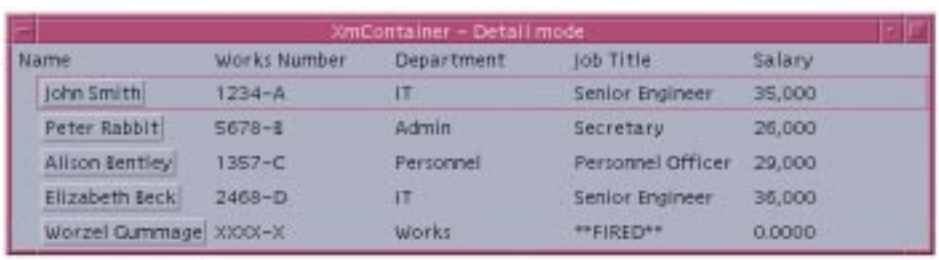


Figure 9-1 A Container in Detail layout, with IconGadget children

Since the Container and IconGadget widget classes fully support Render Table resources, and since the Detail information is held in the form of compound strings, each of the columns and column headers could appear in distinct colors and fonts, although Figure 9-1 does not make use of this feature. Render Tables are discussed in more detail in Chapter 24.

IconGadgets support not one but two image Pixmap resources: the programmer can specify a large and a small image, and then switch between the two views of the IconGadget dynamically. This could be used for displaying both a representative thumbnail and a detailed picture of the application object which the IconGadget represents. A typical usage of the small image display is when the IconGadget is placed in a Container which is configured to lay out its children in a Tree (or more properly, Outline) format. Figure 9-2 has such an example, which is a simplified file system browser.

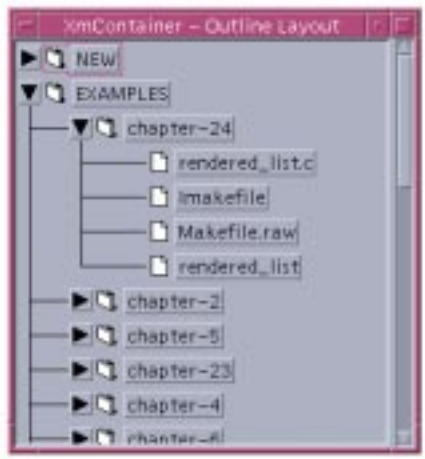


Figure 9-2 A Container in the Outline layout style

In this exampleonly small images are required in order to represent the general type of the objects concerned. The Tree layout is effected through simple constraints placed on the IconGadget children: a given IconGadget A is placed as a child of IconGadget B if the `XmNentryParent` constraint of A has the value B. The order of children can be controlled using the `XmNpositionIndex` constraint, although the Container will lay them out in child order by default.

The Detail column format is not restricted to a particular view of the application object relationships. That is, it is possible to combine a Tree layout with multi-column Detail information. Since the relationships between the application objects is expressed using Container constraints on each of the IconGadget children, this is independent of the current Container layout style. For example, Figure 9-3 shows a Container where the IconGadgets

are laid out in a Tree format because their XmNentryParent constraints are set, with the extra IconGadget information still visible because the Container layout style is set to Detail.

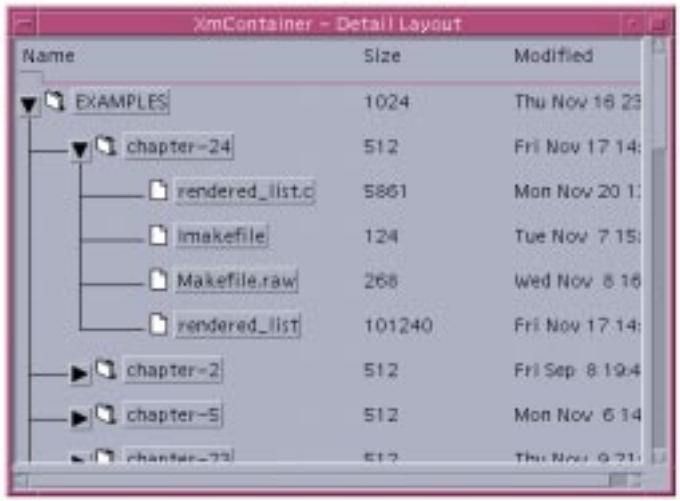


Figure 9-3 A Container in Detail layout, with IconGadgets in a Tree format

The remaining Container layout to consider is known as the Spatial style. In this style, the Container does not attempt to attach Detail to the IconGadget children. The Spatial style is really two styles, which is configured by further resources. Either the IconGadget children are simply placed at the coordinates specified by their XmNx, XmNy resources, very much like a BulletinBoard, or the IconGadget children are laid out in a Grid formation. In the Grid formation, the layout is considered to consist of cells in a rectangular arrangement, and the IconGadget children are aligned within the cells, although it is possible for an IconGadget to span multiple cells. Only one IconGadget may occupy any given cell, however, at any one time. The Grid arrangement behaves very much like a RowColumn in that if the Container is resized, the IconGadgets are automatically laid out to maintain a suitable rectangular arrangement in the space available.

The difference to a BulletinBoard or RowColumn is that the Container in the Spatial layout style supports dynamic selection and drag-and-drop of the IconGadget children: the user can move the IconGadget children around the Container using the mouse. A simple application using the Spatial style would be an Icon Box, or a Tool launch site, where each IconGadget represents pictorially a separate tool.

Figure 9-4 shows such an application in a free-format arrangement - the only positioning is by the x, y coordinate resources of each IconGadget as the Container is initialized, and any movement which the user makes of each IconGadget thereafter. This time, the IconGadgets are configured to display the large images.

Figure 9-4 A Container in Spatial layout using coordinate placement

Figure 9-5 is also in the Spatial style, except that the same data has been configured into a Grid format. The x, y coordinates of the IconGadget children is ignored, and the user may only move IconGadgets into empty cells of the grid.

Figure 9-5 A Container in Spatial layout using Grid placement

The Container in Spatial layout supports single, multiple, browse, and extended selection of the IconGadget children. Selection can be effected either by directly clicking on an IconGadget, or by drawing a rectangle around the items to be selected. The rubberband rectangle which the user describes using the mouse is formally known as a *marquee*. An alternative selection method is to simply swipe the mouse over the IconGadget to be selected. The style of selection preferred is controlled through the `XmNselectionTechnique` resource. Note that the user may only *move* Container items if the layout style is Spatial.

Notification of selection is through callbacks of the Container: the IconGadget does not support any callback resources of its own. The Container supports both the `XmNconvertCallback` and `XmNdestinationCallback` resources, and so fully participates in the Uniform Transfer Model, which is described in Chapter 23.

Creating a Container

Applications that wish to use the Container need to include the file <*Xm/Container.h*>. This file defines the types and functions associated with the widget, as well as defining the widget class name `xmContainerWidgetClass`. A Container can be created in either of the ways shown in the following code fragment:

```
Widget container = XmCreateContainer ( parent, name, resource-value-array,
                                       resource-value-count)
Widget container = XtCreateWidget ( "name", xmContainerWidgetClass, parent,
                                    resource-value-list, NULL);
```

The Container can potentially contain a very large number of IconGadget children, and some of the layout algorithms which it applies are quite complex, and so it is probably best not to create the widget in a managed state (*XtCreateManagedWidget*()) otherwise performance may suffer. See Chapter 8, *Manager Widgets*, for a discussion of when widgets should be created in the managed or unmanaged state.

The *parent* of the Container can be any Shell or Manager widget. Once the Container has been instantiated, the next step is to add IconGadget children.

Creating IconGadgets

The data types and functions associated with the IconGadget object are defined in the header file <*Xm/IconG.h*>, which should be included in any Application which uses the gadget. The header file also defines the gadget class `xmIconGadgetClass`. The following code fragment illustrates how to create an IconGadget:

```
Widget icon = XmCreateIconGadget ( parent, name, resource-value-array,
                                   resource-value-count)
Widget icon = XtCreateWidget ( "name", xmIconGadgetClass, parent,
                               resource-value-list, NULL);
```

The *parent* of an IconGadget can be any Manager widget, although only the Container widget has sufficient knowledge of the IconGadget to display any Detail information associated with the object, or to provide IconGadget selection. The IconGadget can successfully be used outside the Container context if all that you require is the ability to display a label and a Pixmap simultaneously.

Container Resources

The resources of the Container can be logically divided into four groups; those which control the general layout of the IconGadget children, those related to visual aspects of the Container, those associated with any Detail to be displayed, and those which control the selection mechanisms.

Layout Resources

The general layout policy of the Container is controlled through the XmNlayoutType resource, which has the following possible values:

```
XmOUTLINE          XmSPATIAL          XmDETAIL
```

The default value is XmSPATIAL, which results in the BulletinBoard or Grid-style layout. The Tree layout is specified through the value XmOUTLINE, and the IconGadget Detail is made visible using the value XmDETAIL. The XmDETAIL layout may also logically appear as a Tree if the relationship between the IconGadget children has been specified using the XmNentryParent constraint. This is described later in this chapter.

Spatial Layout

In a Spatial layout, the specific Container layout algorithm depends upon a further resource, XmNspatialStyle. If the XmNspatialStyle is XmNONE, layout depends only upon the XmNx, XmNy values of the IconGadget children. If XmNspatialStyle is XmGRID, the Container is laid out in a grid of same-sized cells, and an IconGadget may occupy only once cell. If the spatial style is XmCELLS, the Container is also laid out in same-sized cells, except that this time an IconGadget may span multiple cells.

How the IconGadget is placed within a cell depends upon the XmNspatialSnapModel resource.The value XmCENTER centers the IconGadget within the cell. The resource value XmSNAP_TO_GRID positions the IconGadget at the upper left or right of the cell, depending upon any specified XmNlayoutDirection. The value XmNONE places the IconGadget within the cell using the XmNx, XmNy resources of the object, provided that these coordinates fall within the bounds of the cell - otherwise the IconGadget is laid out in the cell as though the value is XmSNAP_TO_GRID.

The size of a logical cell is specified using the XmNsmallCellHeight, XmNsmallCellWidth, XmNlargeCellHeight, and XmNlargeCellWidth resources. Which of these is operative depends upon whether the Container is configured to display IconGadgets using the large or small Pixmap of the object. This is described in the following *Visual Resources* Section The Container in the spatial style is like a RowColumn in that it creates a logically rectangular arrangement of cells. The difference is that for a RowColumn you have to specify the number of columns and not the size of the cells.

The manner in which the IconGadget children fill the cells depends upon the XmNspatialIncludeModel. If the value is XmFIRST_FIT, the cells are filled in the order that they are free, although this may be in a right to left sense depending upon the XmNlayoutDirection. The value XmAPPEND places an IconGadget into the first free cell after the last filled cell. XmFIRST_FIT and XmAPPEND may well have the same effect when the Container is first displayed, but will have different behavior if a new IconGadget is added after the user has moved the existing objects around. The value XmCLOSEST places an IconGadget into the nearest free cell to the x, y coordinates specified for the IconGadget.

Lastly on the subject of Spatial layout, the XmNspatialResizeModel controls the way in which the Container attempts to grow when there is insufficient space to contain a new IconGadget child. The value XmGROW_BALANCED causes the Container to request both new width and height from its parent as required. The other possible values, XmGROW_MAJOR and XmGROW_MINOR, depend for their interpretation on the value of the XmNlayoutDirection resource. The major dimension is width if the layout policy is horizontally oriented, and height if the policy is vertically oriented. The minor dimension is the reverse: height for a horizontal layout, width for the vertical. Assuming that the layout policy is horizontally aligned, then XmGROW_MAJOR will cause the Container only to ask for more width from its parent. The default is XmGROW_MINOR.

Outline Layout

Whether or not to create a PushButtonGadget used for folding/unfolding portions of the outline Tree is controlled through the XmNoutlineButtonPolicy resource. The default value, XmOUTLINE_BUTTON_PRESENT, creates the button for every container item which has logical children, specified by the constraint resource XmNentryParent. Folding buttons can be disabled by specifying the value XmOUTLINE_BUTTON_ABSENT.

Connecting lines and indentation between portions of the Tree can be configured in a number of ways. The resource XmNoutlineLineStyle can be used to completely disable line drawing - so that indentation is the only clue of the logical parent/child relationships - by specifying the value XmNO_LINE. The default value, XmSINGLE, draws a single line between related Container items. The indentation level, and thus indirectly the length of the lines, depends upon the XmNoutlineIndentation resource. The line length is also partially controlled by the XmNoutlineColumnWidth resource, which specifies preferred width of the first column. The default value, zero, causes the Container to deduce a default value based upon the width of the widest item and the XmNoutlineIndentation specification.

Visual Resources

The Container can specify whether the IconGadget children are forced to display large or small Pixmaps through the XmNentryViewType resource. The default value, XmANY_ICON, leaves the decision to each individual Container item. However, the values XmLARGE_ICON and XmSMALL_ICON can be used to override the settings on each IconGadget in order to give a consistent logical size to all the Container items.

The Container has two Pixmap resources associated with it which control the appearance of the fold/unfold buttons when displaying an Outline layout. The XmNexpandedStatePixmap resource specifies the unfolded state image, and the resource XmNcollapsedStatePixmap specifies the folded image. The default image for the folded state is a sideways pointing arrow, the direction of which depends upon the

XmNlayoutDirection resource. The default for the collapsed (folded) state is a downwards pointing arrow.

In common with all other widgets in the Motif set, the Container supports the XmNrenderTable resource, which controls the rendering of compound strings in the widget. Render Tables are described fully in Chapter 24, Compound Strings in Chapter 25.

Detail Resources

The heading labels which appear at the top of each column of the Container are specified through the XmNdetailColumnHeading and XmNdetailColumnHeadingCount resources. These resources specify an array of compound strings. Each separate compound string in the array forms a distinct column header. The first column header is placed over the IconGadget children themselves.

By default, the order of compound strings in the XmNdetailColumnHeading resource corresponds to the order in which the logical columns appear in the Detail. That is, the first compound string in the XmNdetailColumnHeading array appears above the first column, and so forth. But this need not be the case. The resources XmNdetailOrder and XmNdetailOrderCount specify an array of Cardinal values which represents the order of the columns. If this is not NULL, then column ordering depends on the index values specified. For example, suppose we have three heading labels, A, B, and C. In the absence of any XmNdetailOrder, the headings will appear A, B, C from left to right at the top of the Container. If however we construct a list of Cardinal values, 2, 3, 1, and apply this as the XmNdetailOrder value, then the column headings will appear in the order B, C, A.

By default, the Container works out for itself where each column of the Detail is to begin. It is possible to override the internal algorithm by constructing a Tab List, and then specifying this as the XmNdetailTabList resource. Tab Lists are described in Chapter 24, *Render Tables*, although in this unique instance the required Tab List is used outside the context of a Render Table. Possibly confusing is the fact that the Container and IconGadget also support the XmNrenderTable resource, however both the IconGadget and the Container use the XmNdetailTabList value when calculating the column header layout, although in the case of the IconGadget the value is fetched from the Container parent.

Selection Resources

The XmNselectionPolicy resource specifies the way in which IconGadget children may be selected in the Container. The possible values are:

```
XmSINGLE_SELECT                 XmBROWSE_SELECT
XmMULTIPLE_SELECT               XmEXTENDED_SELECT
```

The default is XmEXTENDED_SELECT, which allows for a discontiguous range of IconGadget children to be selected. Discontiguous in this context means that the children are not necessarily contained within adjacent cells in the Spatial layout. The value XmMULTIPLE_SELECT forces selection from adjacent cells.

The way in which multiple selection is performed using a marquee is configured through the XmNselectionTechnique resource. If an IconGadget must be wholly contained within the marquee rectangle for selection to take place, the value XmMARQUEE should be specified. If the value is XmMARQUEE_EXTEND_START, it is sufficient for the starting coordinates of the marquee to fall within the IconGadget bounds for selection to take effect. The value XmMARQUEE_EXTEND_BOTH includes in the selected set any IconGadget which falls partially within either the start or the end coordinates of the marquee. If selection should take place only if the mouse directly passes over an IconGadget when describing the marquee, the value XmTOUCH_OVER should be specified. Lastly, XmTOUCH_ONLY only selects items which fall between the marquee start and end coordinates. The default is XmTOUCH_OVER.

The set of selected objects for the Container is described by the XmNselectedObjects and XmNselectedObjectCount resources. These specify an array of Widget IDs corresponding to the selected IconGadgets.

The way in which selection callbacks are invoked depends upon the XmNautomaticSelection resource. If the value is XmAUTO_SELECT, callbacks are invoked every time and immediately that an item is selected. The value XmNO_AUTO_SELECT delays callback invocation until the user has finished her current action.

Whether the Container takes control of the Primary Selection when the user selects IconGadget children depends upon the XmNprimaryOwnership resource. The possible values are:

XmOWN_NEVER	XmOWN_ALWAYS
XmOWN_MULTIPLE	XmOWN_POSSIBLE_MULTIPLE

The default value is XmOWN_POSSIBLE_MULTIPLE, which indicates that Primary Selection is owned if multiple selection is possible (the XmNselectionPolicy is XmMULTIPLE_SELECT or XmEXTENDED_SELECT). XmOWN_MULTIPLE is similar, except that ownership takes place only if multiple selection has occurred. The difference is that XmOWN_POSSIBLE_MULTIPLE would allow ownership of the selection even if a single IconGadget is selected (although the selection policy allows for more). The other values have a natural interpretation.

Lastly, the color of a selected IconGadget can be specified using the XmNselectColor resource, which takes a Pixel as its value. Since this is specified on a Container-wide basis, it is not possible to control the selection color of IconGadget children separately.

IconGadget Resources

The IconGadget has two logical sets of resources: those which control the visual aspects of the gadget, and those which specify the Detail information. The Detail resources are only operative in the context of a Container - no other Motif widget knows how to lay out an IconGadget Detail.

Visual Resources

The label of the IconGadget is specified using the XmNlabelString resource: this is a compound string. If the value is NULL, the IconGadget label is derived from the name of the widget as passed to the widgets create routine.*

The image which the IconGadget displays is specified in two halves: a Pixmap representing the foreground image, and a Pixmap controlling the background Mask. Since the IconGadget in fact supports two logical images - small and a large - there are therefore four resources to consider. These are the resources XmNsmallIconPixmap, XmNsmallIconMask, XmNlargeIconPixmap, and XmNlargeIconMask. The Mask resources should only have a depth of 1 - they should be bitmaps. Since the IconGadget only displays one image at a time, the choice of large or small icon is controlled through the XmNviewType resource, which has the possible values XmLARGE_ICON (the default) or XmSMALL_ICON. This may be overridden by a Container parent if the Container resource XmNentryViewType is not set to XmANY_ICON.

The XmNalignment resource specifies the relative alignment of the image and label of the IconGadget. Possible values are:

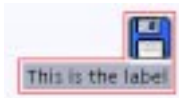
```
XmALIGNMENT_BEGINNING      XmALIGNMENT_CENTER      XmALIGNMENT_END
```

The vertical distance between the image and the label is controlled through the resource XmNspacing. Figure 9-6 shows the effect of setting the alignment and spacing resources.

* If, however, the value is an *empty* compound string (no text components), the widget displays only the image, not both image and label. This can be achieved by creating a simple compound string consisting of a single direction component.

XmALIGNMENT_BEGINNING

XmALIGNMENT_CENTER



XmALIGNMENT_END

XmALIGNMENT_CENTER,
XmNspacing = 20

Figure 9-6 The effect of IconGadget alignment and spacing resources

In common with other widgets in the Motif set, the IconGadget supports the `XmNrenderTable` resource which controls the appearance of compound strings in the object. Render Tables are covered in detail in Chapter 24.

Detail Resources

The resources `XmNdetail` and `XmNdetailCount` specify an array of compound strings which represents the Detail of the IconGadget. Each entry in the compound string array represents a single column entry in the Container parent. Only the Container knows how to lay IconGadget detail out - the resource has no effect in any other parent context.

Container Constraints

There are three constraint resources defined by the Container: `XmNentryParent`, `XmNoutlineState`, and `XmNpositionIndex`. The constraints have no effect in a Spatial layout - they are used to specify the logical Tree arrangement for Outline and Detail layouts.

The `XmNentryParent` resource is used to specify the parent-child relationships between IconGadget children in the Container. An IconGadget A is considered to be a logical child of IconGadget B if the `XmNentryParent` constraint resource of A has the value B. If the value is `NULL`, the IconGadget is considered to be a root object, and is not indented in the layout.

The XmNoutlineState resource controls whether a given IconGadget is visible in an Outline or Detail layout - the value XmCOLLAPSED hides the object, XmEXPANDED displays it.

The order in which IconGadgets are laid out is controlled through the XmNpositionIndex resource. This is not the same thing as specifying the level of indentation, as specified through the XmNentryParent. The XmNpositionIndex resource sorts children at the same level of indentation. For example, suppose objects A and B both have the same XmNentryParent value. A will appear first in the Container if the XmNpositionIndex constraint is less than that of B.

Example 9-1 is a simple application which constructs a partial view of the file system from the current working directory. It does this by creating a Container in the Outline layout style, and adds file system entries in the form of IconGadget children. The file system structure is specified through XmNentryParent resources. The program is not meant to be a full working application, but a demonstration of how the Tree structure is specified for an Outline layout. Output from the program is identical in effect to that of Figure 9-2.

Example 9-1 The outline.c program

```
/* outline.c -- demonstrate the container and icon gadget
** in an outline layout
*/

#include <Xm/Xm.h>
#include <Xm/Form.h>
#include <Xm/Container.h>
#include <Xm/IconG.h>
#include <Xm/ScrolledW.h>
#include <Xm/XmosP.h>
#include <sys/stat.h>
#include <dirent.h>

static Pixmap small_folder_icon = XmUNSPECIFIED_PIXMAP;
static Pixmap small_folder_mask = XmUNSPECIFIED_PIXMAP;
static Pixmap small_file_icon = XmUNSPECIFIED_PIXMAP;
static Pixmap small_file_mask = XmUNSPECIFIED_PIXMAP;

/*
** Adds one level of directory entries to the Container, using
** parent_entry as the XmNentryParent value
*/
static void add_directory( char       *path,
                           Widget     parent_entry,
                           Widget     container)
{
    DIR             *dir;
    struct dirent   *entry;
    Arg             args[8];
    int             n;
```

```
/* Loop through the directory entries */
dir = opendir (path);

while ((entry = readdir (dir)) != NULL) {
    char   name[256];
    struct stat sb;

    /* Get the file details. Note: no real error checking */
    (void) sprintf (name, "%s/%s", path, entry->d_name);

    if (stat (name, &sb) == 0) {
        XmString s;
        Widget item;
        int isDirectory;

        if ((strcmp (entry->d_name, ".") == 0) ||
                (strcmp (entry->d_name, "..") == 0)) {
            continue;
        }

        s = XmStringCreateLocalized (entry->d_name);

        isDirectory = ((sb.st_mode & S_IFDIR) != 0);

        /* Create the IconGadget */
        n = 0;
        XtSetArg (args[n], XmNlabelString, s); n++;

        if (isDirectory) {
            XtSetArg (args[n], XmNsmallIconPixmap, small_folder_icon);
            n++;
            XtSetArg (args[n], XmNsmallIconMask, small_folder_mask);
            n++;
        }
        else {
            XtSetArg (args[n], XmNsmallIconPixmap, small_file_icon);
            n++;
            XtSetArg (args[n], XmNsmallIconMask, small_file_mask);
            n++;
        }

        /* This gives the Tree its structure */
        XtSetArg (args[n], XmNentryParent, parent_entry); n++;

        item = XmCreateIconGadget (container, "icon", args, n);
        XtManageChild (item);

        /* Recurse to subdirectories */
        /* This item becomes the new XmNentryParent */
        if (entry->d_name[0] != '.') {
            if (isDirectory) {
                add_directory (name, item, container);
            }
        }
```

```
            }
        }

        (void) closedir (dir);
    }

    /* Utility: set up a path to find bitmaps and pixmaps */
    static char *GetBitmapPath (Display *display, char *file)
    {
        char               *bmPath = (char *) 0;
        char               *name = (char *) 0;
        Boolean            user_path = False;
        SubstitutionRec    subs;

        subs.substitution = file;

        bmPath = (char *) _XmOSInitPath (file, "XBMLANGPATH", &user_path);

        if (user_path)   subs.match = 'B';
        else             subs.match = 'P';

        name = XtResolvePathname (display, "bitmaps", file, NULL, bmPath,
                                  &subs, 1, NULL);

        if (bmPath) {
            /* Some XtResolvePathname() return non-heap copy of parameter */
            /* This causes serious memory corruption if freed inadvertently */

        if (name != bmPath) {
            XtFree (bmPath);
        }
    }

    return name;
}

/* Utility: load a Pixmap file for the IconGadget image */
static Pixmap GetPixmap (Widget widget, char *file)
{
    Screen *screen = XtScreen (widget);
    char *path = GetBitmapPath (XtDisplay (widget), file);

    return XmGetPixmap  (screen,
                         path,
                         BlackPixelOfScreen (screen),
                         WhitePixelOfScreen (screen));
}

/* Utility: load a Bitmap file for the IconGadget image mask */
static Pixmap GetBitmap (Widget widget, char *file)
{
    Pixmap           pixmap = XmUNSPECIFIED_PIXMAP;
    unsigned int     width;
    unsigned int     height;
```

```
    int             hotx;
    int             hoty;
    char            *path = GetBitmapPath (XtDisplay (widget), file);

    XReadBitmapFile (XtDisplay (widget), XtWindow (widget), path, &width,
                     &height, &pixmap, &hotx, &hoty);

    return pixmap;
}

main (int argc, char *argv[])
{
    Widget          toplevel, form, scrolled_win, container;
    XtAppContext    app;
    int             n;
    Arg             args[8];

    XtSetLanguageProc (NULL, NULL, NULL);

    /* Create the top-level shell and two RowColumns */
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv, NULL,
                                    sessionShellWidgetClass, NULL);

    XtVaSetValues (toplevel, XmNtitle, "XmContainer - Outline Layout", NULL);

    form = XmCreateForm (toplevel, "form", NULL, 0);

    n = 0;
    XtSetArg (args[n], XmNscrollingPolicy, XmAUTOMATIC);    n++;
    XtSetArg (args[n], XmNtopAttachment, XmATTACH_FORM);    n++;
    XtSetArg (args[n], XmNbottomAttachment, XmATTACH_FORM); n++;
    XtSetArg (args[n], XmNleftAttachment, XmATTACH_FORM);   n++;
    XtSetArg (args[n], XmNrightAttachment, XmATTACH_FORM);  n++;
    scrolled_win = XmCreateScrolledWindow (form, "scrolled_win", args, n);

    /* Create the Container */

    n = 0;
    XtSetArg (args[n], XmNwidth, 600);                  n++;
    XtSetArg (args[n], XmNheight, 500);                 n++;
    XtSetArg (args[n], XmNlayoutType, XmOUTLINE);       n++;
    XtSetArg (args[n], XmNentryViewType, XmSMALL_ICON); n++;
    container = XmCreateContainer (scrolled_win, "container", args, n);

    XtManageChild (container);
    XtManageChild (scrolled_win);
    XtManageChild (form);
    XtRealizeWidget (toplevel);

    /* Read in the icons and their masks */
    /* The bitmaps require a window, therefore must be done post-realize */
    small_folder_icon = GetPixmap (container, "folder_small.xbm");
    small_file_icon = GetPixmap (container, "file_small.xbm");
```

```
    small_folder_mask = GetBitmap (container, "folder_small_mask.xbm");
    small_file_mask = GetBitmap (container, "file_small_mask.xbm");

    /* Load the Icon Gadgets */
    add_directory(".", NULL, container);

    XtAppMainLoop (app);
}
```

Container Callbacks

The Container supports a number of callback resources: not all callback types will be invoked in all contexts since some callbacks depend upon the layout policy of the widget.

Firstly, the Container, supporting as it does the dragging and dropping of IconGadget children, defines the XmNconvertCallback and XmNdestinationCallback resources, which are fully described in Chapter 23, *The Uniform Transfer Model*. They will not otherwise be discussed here.

Outline Callbacks

In Outline layout style, the Container calls the XmNoutlineChangedCallback whenever an item in the Tree changes from a collapsed or expanded state. Each callback of this type is passed a pointer to an XmContainerOutlineCallbackStruct data structure, which is defined as follows:

```
typedef struct
{
    int             reason;
    XEvent          *event;
    Widget          item;
    unsigned char   new_outline_state;
} XmContainerOutlineCallbackStruct;
```

The reason element will be either XmCR_COLLAPSED or XmCR_EXPANDED, depending upon the new state of the IconGadget item. The new_outline_state element will be either XmCOLLAPSED or XmEXPANDED, logically matching the reason element. However, the new_outline_state element can be changed by the programmer to force a particular state for the changed item.

Selection Callbacks

There are two selection callbacks supported by the Container. The XmNdefaultActionCallback is invoked whenever an IconGadget item is double clicked on using the mouse, or if the user presses the ACTIVATE key over the item. The XmNselectionCallback is invoked if the user selects an item otherwise. Both types of callback are passed a pointer to an XmContainerSelectCallbackStruct structure, which is specified as follows:

```
typedef struct
{
    int             reason;
    XEvent          *event;
    WidgetList      selected_items;
    int             selected_item_count;
    unsigned char   auto_selection_type;
} XmContainerSelectCallbackStruct;
```

The reason element will reflect the current selection policy: it will be either the value XmCR_SINGLE_SELECT, XmCR_BROWSE_SELECT, XmCR_MULTIPLE_SELECT, or XmCR_EXTENDED_SELECT.

The set of selected Container IconGadget children is specified by the selected_items and selected_item_count elements.

If automatic selection is operative, the auto_selection_type element specifies the current user action state: it will be either XmAUTO_BEGIN (the user has started a new selection), XmAUTO_CANCEL (the user cancels the current selection action), XmAUTO_CHANGE (a new item has been added to the selected set), XmAUTO_MOTION (a new item has been added by a button drag action), or XmAUTO_NO_CHANGE (the user action has not modified the current selected set). If automatic selection is inoperative, the value is XmAUTO_UNSET.

Example 9-2 extends the code of Example 9-1 to include both Detail information and a selection callback which prints out the file system path for the currently selected IconGadget.

Example 9-2 The detail.c program

```
/* detail.c -- demonstrate the container and icon gadget
** in Detail layout, with a selection callback printing
** out the selected IconGadget data
*/

#include <Xm/Xm.h>
#include <Xm/Form.h>
#include <Xm/Container.h>
#include <Xm/IconG.h>
#include <Xm/ScrolledW.h>
#include <Xm/XmosP.h>
#include <sys/stat.h>
#include <dirent.h>
#include <time.h>

static char *column_headings[] = { “Name”, “Size”, “Modified” };

static Pixmap large_folder_icon = XmUNSPECIFIED_PIXMAP;
static Pixmap small_folder_icon = XmUNSPECIFIED_PIXMAP;
static Pixmap large_folder_mask = XmUNSPECIFIED_PIXMAP;
static Pixmap small_folder_mask = XmUNSPECIFIED_PIXMAP;
static Pixmap large_file_icon = XmUNSPECIFIED_PIXMAP;
```

```
static Pixmap small_file_icon = XmUNSPECIFIED_PIXMAP;
static Pixmap large_file_mask = XmUNSPECIFIED_PIXMAP;
static Pixmap small_file_mask = XmUNSPECIFIED_PIXMAP;

/* Create the Icongadget children at a given level in the
** file system. This time, Detail is added about the files.
*/
static void add_directory( char       *path,
                           Widget     parent_entry,
                           Widget     container)
{
    DIR             *dir;
    struct dirent   *entry;
    Arg             args[12];
    int             n;

    /* Loop through the directory entries */
    dir = opendir (path);

    while ((entry = readdir (dir)) != NULL) {
        char name[256];
        struct stat sb;

        /* Get the file details. Note: no real error checking */
        (void) sprintf (name, "%s/%s", path, entry->d_name);

        if (stat (name, &sb) == 0) {
            XmString        s;
            XmStringTable   details;
            char            buf[20];
            Widget          item;
            int             isDirectory;

            if ((strcmp (entry->d_name, ".") == 0) ||
                    (strcmp (entry->d_name, "..") == 0)) {
            continue;
        }

        s = XmStringCreateLocalized (entry->d_name);

        isDirectory = ((sb.st_mode & S_IFDIR) != 0);

        /* Create the details array */
        details = (XmStringTable) XtMalloc(2 * sizeof (XmString *));

        (void) sprintf (buf, "%d", sb.st_size);
        details[0] = XmStringCreateLocalized (buf);
        details[1] = XmStringCreateLocalized (
                                (char *) ctime(&sb.st_mtim.tv_sec));

        /* Create the IconGadget */
        n = 0;
        XtSetArg (args[n], XmNlabelString, s); n++;
```

```
        if (isDirectory) {
            XtSetArg (args[n], XmNlargeIconPixmap, large_folder_icon);
            n++;
        }
        else {
            XtSetArg (args[n], XmNlargeIconPixmap, large_file_icon);
            n++;
        }

        if (isDirectory) {
            XtSetArg (args[n], XmNlargeIconMask, large_folder_mask);
            n++;
        }
        else {
            XtSetArg (args[n], XmNlargeIconMask, large_file_mask);
            n++;
        }

        if (isDirectory) {
            XtSetArg (args[n], XmNsmallIconPixmap, small_folder_icon);
            n++;
        }
        else {
            XtSetArg (args[n], XmNsmallIconPixmap, small_file_icon);
            n++;
        }

        if (isDirectory) {
            XtSetArg (args[n], XmNsmallIconMask, small_folder_mask);
            n++;
        }
        else {
            XtSetArg (args[n], XmNsmallIconMask, small_file_mask);
            n++;
        }

        XtSetArg (args[n], XmNentryParent, parent_entry); n++;
        XtSetArg (args[n], XmNdetail, details);           n++;
        XtSetArg (args[n], XmNdetailCount, 2);            n++;

        item = XmCreateIconGadget (container, "icon", args, n);
        XtManageChild (item);

        XmStringFree(details[0]);
        XmStringFree(details[1]);
        XtFree ((char *) details);

        /* Recurse to subdirectories */
        if (entry->d_name[0] != '.') {
            if (isDirectory) {
                add_directory (name, item, container);
            }
        }
    }
```

```
    }

    (void) closedir (dir);
}

/* For brevity, these are copied from Example 9-1 */
extern char *GetBitmapPath (Display *display, char *file);
extern Pixmap GetPixmap (Widget widget, char *file);
extern Pixmap GetBitmap (Widget widget, char *file);

/* Construct a file system path from the selected IconGadget */
/* This is crude, but it works. Optimization is a reader exercise */
static char *GetPath (Widget w, char *sofar)
{
    static char      buffer[512];
    char             temp[512], *leaf;
    Widget           parent = NULL;
    XmString         xms;

    if (sofar == NULL) {
        (void) strcpy (buffer, "");
    }

    if (w == NULL) {
        return buffer;
    }

    (void) strcpy (temp, buffer);

    XtVaGetValues (w, XmNentryParent, &parent, XmNlabelString, &xms, NULL);

    leaf = (char *) XmStringUnparse (xms, NULL,
                                     XmCHARSET_TEXT,
                                     XmCHARSET_TEXT,
                                     NULL, 0,
                                     XmOUTPUT_ALL);

    (void) sprintf (buffer, "%s%s%s", leaf, (sofar ? "/" : ""), temp);

    XtFree (leaf);

    return GetPath (parent, buffer);
}

/* The selection callback: simply prints out
** the selected IconGadget file system path
*/
static void select_callback (   Widget       w,
                              XtPointer    client_data,
                              XtPointer    call_data)
    {
    XmContainerSelectCallbackStruct *cptr;
    int i;
    char *path;
```

```
    cptr = (XmContainerSelectCallbackStruct *) call_data;

    for (i = 0; i < cptr->selected_item_count; i++) {
        printf ("Selected: %s\n",
        GetPath (cptr->selected_items[i], NULL));
    }
}

main (int argc, char *argv[])
{
    Widget          toplevel, form, scrolled_win, container;
    XtAppContext    app;
    int             i, n, count;
    Arg             args[12];
    XmStringTable   column_headings_table;

    XtSetLanguageProc (NULL, NULL, NULL);
    /* Create the top-level shell and two RowColumns */
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv, NULL,
                                    sessionShellWidgetClass, NULL);

    XtVaSetValues (toplevel,
    XmNtitle, "XmContainer - Detail Layout", NULL);

    form = XmCreateForm (toplevel, "form", NULL, 0);

    n = 0;
    XtSetArg (args[n], XmNscrollingPolicy, XmAUTOMATIC);    n++;
    XtSetArg (args[n], XmNtopAttachment, XmATTACH_FORM);    n++;
    XtSetArg (args[n], XmNbottomAttachment, XmATTACH_FORM); n++;
    XtSetArg (args[n], XmNleftAttachment, XmATTACH_FORM);   n++;
    XtSetArg (args[n], XmNrightAttachment, XmATTACH_FORM);  n++;
    scrolled_win = XmCreateScrolledWindow (form, "scrolled_win", args, n);

    /* Create the Container */
    count = XtNumber (column_headings);
    column_headings_table = (XmStringTable) XtMalloc(
    count * sizeof (XmString *));
    for (i = 0; i < count; i++)
        column_headings_table[i] =
                        XmStringCreateLocalized (column_headings[i]);

    n = 0;
    XtSetArg (args[n], XmNwidth, 600);                                  n++;
    XtSetArg (args[n], XmNheight, 500);                                 n++;
    XtSetArg (args[n], XmNdetailColumnHeading, column_headings_table);  n++;
    XtSetArg (args[n], XmNdetailColumnHeadingCount, count);             n++;
    XtSetArg (args[n], XmNlayoutType, XmDETAIL);                        n++;
    XtSetArg (args[n], XmNentryViewType, XmSMALL_ICON);                 n++;
    XtSetArg (args[n], XmNselectionPolicy, XmSINGLE_SELECT);            n++;
    container = XmCreateContainer (scrolled_win, "container", args, n);

    /* Register a Selection Callback for the Container */
```

```
        XtAddCallback (container,
        XmNselectionCallback, select_callback, NULL);

        /* Reclaim Memory which the widgets have copied */
        for (i = 0; i < count; i++)
            XmStringFree (column_headings_table[i]);
        XtFree ((char *) column_headings_table);

        XtManageChild (container);
        XtManageChild (scrolled_win);
        XtManageChild (form);
        XtRealizeWidget (toplevel);

        /* Read in the icons and their masks */
        /* The bitmaps require a window, therefore must be done post-realize */
        large_folder_icon = GetPixmap (container, "folder_large.xbm");
        small_folder_icon = GetPixmap (container, "folder_small.xbm");
        large_file_icon = GetPixmap (container, "file_large.xbm");
        small_file_icon = GetPixmap (container, "file_small.xbm");

        large_folder_mask = GetBitmap (container, "folder_large_mask.xbm");
        small_folder_mask = GetBitmap (container, "folder_small_mask.xbm");
        large_file_mask = GetBitmap (container, "file_large_mask.xbm");
        small_file_mask = GetBitmap (container, "file_small_mask.xbm");

        add_directory(".", NULL, container);

        XtAppMainLoop (app);
    }
```

The example output is logically the same as Figure 9-3, although the contents of the Tree will naturally depend upon your file system.

Container Functions

The Container support utilities fall into two groups: those concerned with transferring Container selected items to and from the Clipboard, and general purpose utilities. It is probably best to start in the reverse order and describe the utility functions first, as these are generally the more useful for typical application programming.

Forcing Container Layout

The convenience function `XmContainerRelayout`() forces the widget to recalculate the layout of all its IconGadget children. This only takes effect if the layout style is Spatial, and if the Container is managed. The routine could be used if some kind of sorting of the IconGadget children has taken place, either by respecifying the `XmNx`, `XmNy` coordinates of each or through manipulation of the `XmNpositionIndex` resource. The routine has the following functional signature:

```
void XmContainerRelayout (Widget container)
```

XmContainerRelayout() does not cause geometry management side effects: the routine will not result in the Container requesting size changes from its parent in turn.

Sorting Container Items

After a set of Container items has their XmNpositionIndex resource modified, the routine XmContainerReorder() can be used to force the Container to internally update its knowledge of the layout. If the layout style is Detail or Outline, the Container will also refresh the display. Note that XmContainerRelayout() is required for an update to the Spatial style - XmContainerReorder() has no effect on this layout style. XmContainerReorder() has the following specification:

```
void XmContainerReorder ( Widget         container,
                          WidgetList     items,
                          int            item_count)
```

The function simply reorders the *items* in the layout of the given *container* according to the XmNpositionIndex of each, using a quicksort algorithm.

Fetching Container Items

Given an arbitrary Container item in a Detail or Outline layout, presumably as a result of a selection operation, we can deduce the logical parent of that item simply by inspecting the XmNentryParent constraint. If however we wanted to know the logical children of the item, there is no resource which will provide the required information. This is where the routine XmContainerGetItemChildren() is useful. The routine returns the list of IconGadgets whose XmNentryParent resource points at a given item, and it is defined as follows:

```
int XmContainerGetItemChildren ( Widget          container,
                                 Widget         icon_gadget,
                                 WidgetList     *item_children)
```

The set of logical children of the *icon_gadget* is returned at the address specified by *icon_children*. The routine allocates memory for this, and it is the responsibility of the programmer to reclaim the memory using XtFree() at an appropriate point. The number of items in the array is returned by the function.

Container Clipboard Routines

There are five routines defined for copying and pasting container items into the clipboard. They are XmContainerCut(), XmContainerCopy(), XmContainerCopyLink(), XmContainerPaste(), and XmContainerPasteLink(). They are defined as follows:

```
Boolean XmContainerCut (Widget container, Time timestamp);
Boolean XmContainerCopy (Widget container, Time timestamp)
Boolean XmContainerCopyLink (Widget container, Time timestamp)
Boolean XmContainerPaste (Widget container)
Boolean XmContainerPasteLink (Widget container)
```

In each case, the *timestamp* parameter should simply be specified using the routine XtLastTimestampProcessed().

The routines are fully integrated with the Uniform Transfer Model - they do not actually cut or copy data to and from the clipboard, but internally invoke convert and destination callbacks to perform these tasks. The Uniform Transfer Model is covered in detail in Chapter 23.

A typical usage of these routines is where an Edit menu has been defined for the Container: the cut/copy/paste actions of the menu entries will be a simple interface onto the above functions, and is appropriate if the programmer is presenting an interface where the user can edit a Tree or reorder a Detail layout by moving items around through the clipboard.

Summary

The Container is the most flexible layout Manager in Motif. It offers multiple views of the same data by allowing the programmer to switch layout dynamically. The selection mechanisms whereby the Container reports actions in the widget are simple to understand and easy to program. By separating the functionality whereby the Container performs the layout and selection and the IconGadget represents the application object, the widget can be considered as an approximation to a Model-View-Controller architecture. Whether the application data needs to be viewed in Tabular, Tree, Grid, or free-floating format, the Container provides the necessary layout.

Exercises

The following exercises expand upon concepts presented in this chapter.

1. Modify the program of Example 9-2 to display file ownership, permissions, and other status information amongst the Detail. Remember to expand the column headings for the Container. Add options to the program to allow the user to display or hide particular columns of detail, and to change the order of the columns.
2. Add new images to the program to represent different types of file: binary, source files, objects. Add a default action callback, and make the callback spawn an appropriate action for the selected object: binaries could be executed, source files edited.

In this chapter:

- *The ScrolledWindow Design Model*
- *Creating a ScrolledWindow*
- *Working With ScrollBars*
- *Implementing True Application-defined Scrolling*
- *Working With Keyboard Traversal in ScrolledWindows*
- *Summary*
- *Exercises*

10

ScrolledWindows and ScrollBars

This chapter describes the ins and outs of scrolling. It pays particular attention to application-defined scrolling, which is often required when the simple scrolling provided by the ScrolledWindow widget is insufficient.

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

The ScrolledWindow Design Model

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

In automatic scrolling mode, the ScrolledWindow operates entirely on its own, adjusting the viewport as necessary in response to ScrollBar activity. The application simply creates the desired data, such as a Label widget that contains a large pixmap, and makes that widget the work window for the ScrolledWindow. When the user operates the ScrollBars to change the visible area, the ScrolledWindow adjusts the Label so that the appropriate portion is visible. This design is demonstrated in Chapter 4, *The Main Window*, and Chapter 11, *The DrawingArea Widget*.

With application-defined scrolling, the ScrolledWindow operates under the assumption that the work window is not complete. The widget assumes that another entity, such as the

application or the internals of another widget, controls the data within the work window and that the data may change dynamically as the user scrolls. In order to control scrolling, the application must control all aspects of the ScrollBars. This level of control is necessary when it is impossible or impractical for an application to provide the ScrolledWindow with a sufficiently large work window (or the data for it) at any one time.

The Automatic Scrolling Model

Most of the time, the ScrolledWindow widget is used in automatic scrolling mode. When it is used in this mode, the ScrolledWindow contains at most three internal widgets: two ScrollBars and a *clip window*[*]. The ScrolledWindow creates these widgets automatically. The work area is an external widget (specified by the `XmNworkWindow` resource) that is clipped by the clip window. This work window is a child of the ScrolledWindow that is provided by the application; it is not created automatically by the ScrolledWindow. When the user interacts with the ScrollBars, the work window is adjusted so that the appropriate part is visible through the clip window. The general design of the ScrolledWindow in automatic scrolling mode is illustrated in Figure 10-1.

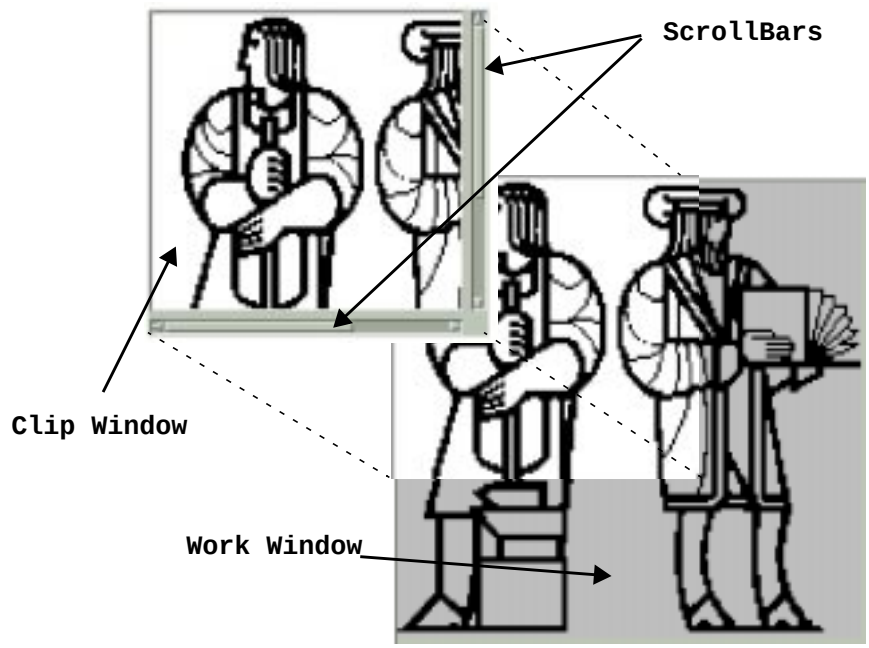


Figure 10-1 Design of an automatic ScrolledWindow

* In Motif 1.2, the clip window is implemented as a DrawingArea. In Motif 2.x, the clip window is a specialized sub-class of the DrawingArea., called `xmClipWindowWidgetClass` There is no public header file for the subclass.

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

Be warned: the ScrolledWindow internally reparents the work window to be a child of the clip window. A call of `XtParent`() on the work window is not going to return the ScrolledWindow even though you specify the ScrolledWindow as parent when you add the work area.

The Application-defined Scrolling Model

In the application-defined scrolling model, which is the default model, the ScrolledWindow always makes itself the same size as the work window. Just as for automatic scrolling, the application must provide the work window as a child of the ScrolledWindow. The main reason to use application-defined scrolling is if the work window contains more data than can possibly be loaded in the automatic scrolling mode. An application may also require different scrolling behavior than the default pixel-by-pixel increments provided by the automatic scrolling mode. Application-defined scrolling is also the best option when the contents of the work window changes dynamically and the application does not want to rely on the ScrolledWindow to scroll new data into view.

The disadvantage of application-defined scrolling is that the application, not the ScrolledWindow, is responsible for the ScrollBars. The application must create and manage the ScrollBars, as well as respond to the scrolling actions initiated by the user. Since what is displayed in the clip window and the work window are identical, the ScrolledWindow widget does not bother to create a clip window*. However, there are still some limitations as to what the ScrolledWindow can support. It is important that you understand the limitations before designing your application, so let's look at two examples.

A Text widget that displays the contents of an arbitrarily large file provides a classic example of application-defined scrolling. Under the automatic scrolling model, the application might have to provide the ScrolledWindow with a work window that is large

* A call of `XtParent`() on the work window in the application-defined scrolling model does return the ScrolledWindow because the work area is *not* reparented.

enough to render thousands of lines of text, so that all of the text is immediately available to the user. An object of such proportions is prohibitive for reasonable performance and resource consumption. Since the work window cannot be as large as it would need to be for automatic scrolling, it might as well be as small as possible, which is the size of the clip window. When the Text widget is a child of a ScrolledWindow, the Text widget creates its own ScrollBars and attaches callback routines to them so that it can be notified of scrolling actions.When the user scrolls, the Text widget changes the text in the work window to the text that corresponds to the new region that just scrolled into view. The user has the illusion that scrolling is taking place, but in reality, the data in the work window has simply changed, thereby saving a great deal of overhead in system and server resources. The List widget uses the same method when it is the child of a ScrolledWindow. The Text and List widgets are the only examples of application-defined scrolling that are supported by the current implementation of the ScrolledWindow.

There is another scenario in which a large amount of data is retrieved dynamically and is not all available at the same time. Even though the ScrolledWindow does not really support this scenario, you should be familiar with the situation, since it may come up in a large application. There are some possible work arounds that we'll discuss later in the chapter. Let's say that the Pacific Gas and Electric Company has an online database that contains all of the pipeline information for California and that an operator wants to view the data for San Francisco county. To display this information, the application must read the data from the database and convert that data into an image that can be presented in a ScrolledWindow.

Although the database cannot get all the information for the whole county all at once, it can get more information than the window can display. Let's say that the window can display 10% of the county and the database can return information on 20% of the county in a reasonable amount of time. The application needs to use the application-defined mechanisms because 100% of the data is not available for automatic scrolling. The fact that more than what can be displayed is available just means that the application could optimize performance by avoiding unnecessary retrieval of data from the database whenever scrolling takes place. The application could reuse the existing work window as a cache, so that if the user scrolls by an amount that is small enough, the work window is redisplayed in a way similar to the automatic scrolling mechanism. The application would still have to control this behavior manually, though.

Unfortunately, the ScrolledWindow does not support this type of behavior. The ScrolledWindow always expands to the size of its work window in application-defined scrolling mode. In other words, you cannot have a work window that is a different size from the clip window. This situation leaves you with several design decisions. You could reduce the amount of data obtained from a database query, throw away excess information not used in your display, or make the viewport of an automatic ScrolledWindow large enough for each query. In any case, the best approach is to use some method that makes the size of the work window the same as the clip window. While this requirement may present some

logistical problems with the design of your application, we'll discuss some work arounds for the situation later in the chapter.

In the two preceding examples, we have defined two fundamentally similar methods of scrolling: semi-automatic scrolling and true application-defined scrolling. In the first case, Text and List widgets handle their own scrolling internally through special-case routines attached to the ScrollBars. We call this method semi-automatic scrolling, since the application programmer is not responsible for the scrolling of these widgets. Nevertheless, the ScrolledWindow is in the application-defined scrolling mode. This situation is in contrast to true application-defined scrolling, where you must handle the ScrollBars and the associated scrolling actions entirely on your own. This method is more intricate and requires a significant amount of code to be implemented properly.

Obviously, the automatic scrolling mechanism provided by the ScrolledWindow is much simpler than the application-defined mechanism and it requires much less application intervention. However, there are some drawbacks in the implementation of automatic scrolling. Automatic ScrolledWindows only scroll in single-pixel increments. If other scrolling behavior is required, you must use application-defined scrolling. And while application-defined scrolling is far more complicated, the advantage is that it provides more flexibility in the ways that the object is scrolled.

Creating a ScrolledWindow

Creating a ScrolledWindow is no different from creating other kinds of Motif widgets. Applications that wish to use ScrolledWindows must include the header file *<Xm/ScrolledW.h>*. The process of creating a ScrolledWindow is shown in the following code fragments:

```
Widget scroll_w = XmCreateScrolledWindow (parent, "name", resource-value-array,
                                          resource-value-count);

Widget scroll_w = XtCreateWidget ( "name", xmScrolledWindowWidgetClass, parent,
                                   resource-value-list, NULL);
```

The `parent` can be a Shell or any manager widget. The ScrolledWindow could be created as a managed widget, since the addition of its child does not cause it to renegotiate its size. (See Chapter 8, *Manager Widgets*, for a discussion of when manager widgets should be created as managed or unmanaged widgets.) The resource-value pairs control the behavior of the ScrolledWindow, as well as its visual effects. The most important resources are `XmNscrollingPolicy`, `XmNvisualPolicy`, and `XmNscrollBarDisplayPolicy`. The value for `XmNscrollingPolicy` can be set to either `XmAUTOMATIC` or `XmAPPLICATION_DEFINED`, depending on which scrolling method you want to use. The use of other ScrolledWindow resources varies depending on the scrolling behavior that is specified.

Automatic Scrolling

In automatic scrolling mode, the ScrolledWindow assumes that all of the data is already available in the work window and that the size of the work window represents the entire size of the viewable data. Even if the data changes and the size of work window is modified, the ScrolledWindow can still manage its display automatically. The ScrolledWindow should never resize itself due to changes in the work windows, so `XmNvisualPolicy` is typically set to `XmCONSTANT`. This value tells the ScrolledWindow not to resize itself when the work window grows or shrinks. If `XmNvisualPolicy` is set to `XmVARIABLE`, the ScrolledWindow always sizes itself to contain the entire work window, which nullifies the need for an automatic ScrolledWindow. Like any other widget, the only time that a ScrolledWindow should change size is when the parent resizes it, presumably for one of the following reasons:

- The shell has been resized.
- The ScrolledWindow is a child of a PanedWindow that the user has resized.
- Adjacent, sibling widgets have been resized, added, removed, etc.
- Application-controlled changes in widget size have been made.

The default size of the ScrolledWindow is never the same size as the work area, unless it's a coincidence*. The default size is not very useful, so you should probably specify the `XmNwidth` and `XmNheight` resources for a ScrolledWindow. A problem arises if you want the ScrolledWindow to initialize itself to the size of the work window and have it be in automatic scrolling mode. To make the ScrolledWindow the same size as the work window, you must use application-defined scrolling.

For automatic scrolling, the only thing left to decide is how you want the ScrollBars to be displayed if the work window dynamically grows or shrinks. There may be situations where the work window is the same size as or smaller than the clip window. In this case, you may not want to display the ScrollBars, since they are not needed. If so, you can set `XmNscrollBarDisplayPolicy` to `XmAS_NEEDED`. If you always want the ScrollBars to be visible, whether or not they are needed, you can set the resource to `XmSTATIC`. Some people prefer static ScrollBars, so that consistency is maintained in the interface; having ScrollBars appear and disappear frequently may be confusing. Perhaps the best thing to do is to allow the user to specify the `XmNscrollBarDisplayPolicy`. You can always set your preference in the application defaults file, as shown below:

```
*XmScrolledWindow.scrollBarDisplayPolicy: STATIC
```

* The internals to the ScrolledWindow widget happen to set the width and height to `100` pixels, although this fact is not officially documented by OSF.

Application-defined Scrolling

In the application-defined scrolling mode, XmNscrollingPolicy is set to XmAPPLICATION_DEFINED. In this case, the work window must be the same size as the clip window, so the size of the work window is set by the toolkit. As a result, the XmNvisualPolicy resource has the value of XmVARIABLE, which indicates that the work window grows and shrinks with the ScrolledWindow. Since the two windows are the same size, the ScrolledWindow doesn't need to have a clip window, so it doesn't create one.

Because application-defined scrolling implies that you are responsible for the creation and management of the ScrollBars, the toolkit forces the XmNscrollBarDisplayPolicy to XmSTATIC, which means that the ScrolledWindow always displays the ScrollBars if they are managed. Since the ScrolledWindow cannot know the size of the entire data, it cannot automate the visibility of the ScrollBars. If you want your application to emulate the XmAS_NEEDED behavior, you must monitor the size of the ScrolledWindow and the work area and manage the ScrollBars manually.

Additional Resources

Another ScrolledWindow resource is the XmNworkWindow, which is used to identify the widget that acts as the ScrolledWindow's work window. A ScrolledWindow can have only one work window and a work window can be associated with only one ScrolledWindow. In other words, you cannot assign the same widget ID to multiple ScrolledWindows to get multiple views into the same object. There are ways of achieving this effect, though, that will become apparent as we go through the chapter.

The XmNclipWindow resource specifies the widget ID for the clip window. This resource is read-only, so it is illegal to set the clip window manually or to reset it to NULL. For practical purposes, this resource should be left alone. The XmNverticalScrollBar and XmNhorizontalScrollBar resources specify the widget IDs of the ScrollBars in the ScrolledWindow. These resources allow you to set and retrieve the ScrollBars, which is useful for monitoring scrolling actions and setting up application-defined scrolling. Like any other manager, the ScrolledWindow also has resources that control the margin height and width and other visual attributes.

An Automatic ScrolledWindow Example

Automatic scrolling is the simpler of the two types of scrolling policies available. Fortunately, it is also the more common of the two. You shouldn't let this simplicity sway you too much, though, as it is a common design error for programmers to use the automatic scrolling mechanisms for designs that are better suited to the application-defined model. On the other hand, if you merely want to monitor scrolling without necessarily controlling it, you can install your own callback routines on the ScrollBars in an automatic ScrolledWindow, as we'll describe in the next section

In automatic mode, a ScrolledWindow automatically creates its own ScrollBars and handles their callback procedures to position the work window in the clip window. All of the examples that use ScrolledWindows in the rest of the chapters in this book (such as those in Chapter 4, *The Main Window*, and Chapter 10, *ScrolledWindows and ScrollBars*) use the automatic scrolling mode. The only exceptions are the ScrolledList and ScrolledText objects, but the List and Text widgets handle application-defined scrolling internally.

Example 10-1 shows a large panel of Labels, ToggleButtons, and Text widgets that are arranged in a collection of Form and RowColumn widgets and managed by a ScrolledWindow widget.*

Example 10-1 The getusers.c program

```
/* getusers.c -- demonstrate a simple ScrolledWindow by showing
** how it can manage a RowColumn that contains a vertical stack of
** Form widgets, each of which contains a Toggle, two Labels and
** a Text widget. The program fills the values of the widgets
** using various pieces of information from the password file.
** Note: there are no callback routines associated with any of the
** widgets created here -- this is for demonstration purposes only.
*/
#include <Xm/PushBG.h>
#include <Xm/LabelG.h>
#include <Xm/ToggleB.h>
#include <Xm/ScrolledW.h>
#include <Xm/RowColumn.h>
#include <Xm/Form.h>
#include <Xm/Text.h>
#include <pwd.h>

typedef struct {
    String     login;
    int        uid;
    String     name;
    String     homedir;
} UserInfo;

/* use getpwent() to read data in the password file to store
** information about all the users on the system. The list is
** a dynamically grown array, the last of which has a NULL login.
*/

UserInfo *getusers(void)
{
    /* extern struct *passwd getpwent(); */
    extern char      *strcpy();
    struct passwd    *pw;
    UserInfo         *users = NULL;
```

* XtVaAppInitialize() is considered deprecated in X11R6.

```
    int             n;

    setpwent();

    /* getpwent() returns NULL when there are no more users */
    for (n = 0; pw = getpwent(); n++) {
        /* reallocate the pointer to contain one more entry. You may choose
        ** to optimize by adding 10 entries at a time, or perhaps more?
        */
        users = (UserInfo *) XtRealloc ((char *) users,
                                        (n+1) * sizeof (UserInfo));
        users[n].login = strcpy (XtMalloc (strlen (pw->pw_name)+1),
                                           pw->pw_name);
        users[n].name = strcpy (XtMalloc (strlen (pw->pw_gecos)+1),
                                           pw->pw_gecos);
        users[n].homedir = strcpy (XtMalloc (strlen (pw->pw_dir)+1),
                                           pw->pw_dir);
        users[n].uid = pw->pw_uid;
    }

    /* allocate one more item and set its login string to NULL */
    users = (UserInfo *) XtRealloc ((char *) users,
    (n+1) * sizeof (UserInfo));
    users[n].login = NULL;
    endpwent();

    return users; /* return new array */
}

main (int argc, char *argv[])
{
    Widget         toplevel, sw, main_rc, form, toggle, child;
    XtAppContext   app;
    UserInfo       *users;
    Arg            args[10];
    int            n;

    XtSetLanguageProc (NULL, NULL, NULL);
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv, NULL,
                                    sessionShellWidgetClass, NULL);
    /* Create a 500x300 scrolled window. This value is arbitrary,
    ** but happens to look good initially. It is resizable by the user.
    */
    n = 0;
    XtSetArg (args[n], XmNwidth, 500);                   n++;
    XtSetArg (args[n], XmNheight, 300);                  n++;
    XtSetArg (args[n], XmNscrollingPolicy, XmAUTOMATIC); n++;
    sw = XmCreateScrolledWindow (toplevel, "scrolled_w",   args, n);

    /* RowColumn is the work window for the widget */
    main_rc = XmCreateRowColumn (sw, "main_rc", NULL, 0);

    /* load the users from the passwd file */
    if (!(users = getusers())) {
```

```
        perror ("Can't read user data info");
        exit (1);
    }

    /* for each login entry found in the password file, create a
    ** form containing a toggle button, two labels and a text widget.
    */
    while (users->login) {
        /* NULL login terminates list */
        char uid[8];

        form = XmCreateForm (main_rc, "", NULL, 0);

        n = 0;
        XtSetArg (args[n], XmNalignment, XmALIGNMENT_BEGINNING);   n++;
        XtSetArg (args[n], XmNtopAttachment, XmATTACH_FORM);       n++;
        XtSetArg (args[n], XmNbottomAttachment, XmATTACH_FORM);    n++;
        XtSetArg (args[n], XmNleftAttachment, XmATTACH_FORM);      n++;
        XtSetArg (args[n], XmNrightAttachment, XmATTACH_POSITION); n++;
        XtSetArg (args[n], XmNrightPosition, 15);                  n++;
        child = XmCreateToggleButton (form, users->login, args, n);
        XtManageChild (child);

        sprintf (uid, "%d", users->uid);
        n = 0;
        XtSetArg (args[n], XmNalignment, XmALIGNMENT_END);         n++;
        XtSetArg (args[n], XmNtopAttachment, XmATTACH_FORM);       n++;
        XtSetArg (args[n], XmNbottomAttachment, XmATTACH_FORM);    n++;
        XtSetArg (args[n], XmNleftAttachment, XmATTACH_POSITION);  n++;
        XtSetArg (args[n], XmNleftPosition, 15);                   n++;
        XtSetArg (args[n], XmNrightAttachment, XmATTACH_POSITION); n++;
        XtSetArg (args[n], XmNrightPosition, 20);                  n++;
        child = XmCreateLabelGadget (form, uid, args, n);
        XtManageChild (child);

        n = 0;
        XtSetArg (args[n], XmNalignment, XmALIGNMENT_BEGINNING);   n++;
        XtSetArg (args[n], XmNtopAttachment, XmATTACH_FORM);       n++;
        XtSetArg (args[n], XmNbottomAttachment, XmATTACH_FORM);    n++;
        XtSetArg (args[n], XmNleftAttachment, XmATTACH_POSITION);  n++;
        XtSetArg (args[n], XmNleftPosition, 20);                   n++;
        XtSetArg (args[n], XmNrightAttachment, XmATTACH_POSITION); n++;
        XtSetArg (args[n], XmNrightPosition, 50);                  n++;
        child = XmCreateLabelGadget (form, users->name, args, n);
        XtManageChild (child);

        /* Although the home directory is readonly, it may be longer
        ** than expected, so don't use a Label widget. Use a Text widget
        ** so that left-right scrolling can take place.
        */
        n = 0;
        XtSetArg (args[n], XmNtopAttachment, XmATTACH_FORM);      n++;
        XtSetArg (args[n], XmNbottomAttachment, XmATTACH_FORM);   n++;
        XtSetArg (args[n], XmNleftAttachment, XmATTACH_POSITION); n++;
```

```
            XtSetArg (args[n], XmNleftPosition, 50);                    n++;
            XtSetArg (args[n], XmNrightAttachment, XmATTACH_FORM);      n++;
            XtSetArg (args[n], XmNvalue, users->homedir);               n++;
            XtSetArg (args[n], XmNeditable, False);                     n++;
            XtSetArg (args[n], XmNcursorPositionVisible, False);        n++;
            child = XmCreateText (form, users->homedir, args, n);
            XtManageChild (child);

            XtManageChild (form);
            users++;
        }

        XtManageChild (main_rc);
        XtManageChild (sw);
        XtRealizeWidget (toplevel);
        XtAppMainLoop (app);
    }
```

Those of you who are familiar with UNIX programming techniques should find the use of `getpwent()` and `endpwent()` quite familiar. If you are not aware of these functions, you should consult the documentation for your UNIX system. In short, they can be used to return information about the contents of the password file (typically */etc/passwd*), which contains information about all of the users on the system. The first call to `getpwent()` opens the password file and returns a data structure describing the first entry. Subsequent calls return consecutive entries. When the entries have been exhausted, `getpwent()` returns `NULL` and `endpwent()` closes the password file. In Example 10-1, the information from the password file is represented using ToggleButtons, Labels, and Text widgets, as shown in Figure 10-2.

Figure 10-2 Output of the getusers program

The components in the program do not have any functionality; the program is used solely to demonstrate how panels of arbitrary widgets can be displayed in a ScrolledWindow. The widget hierarchy is irrelevant to the operation of the ScrolledWindow. In this particular case, the ScrolledWindow is a child of the top-level shell. We could have used a MainWindow widget in place of a ScrolledWindow; these two components are interchangeable because the MainWindow is subclassed from the ScrolledWindow. See Chapter 4, *The Main Window*, for more details on how the MainWindow widget fits into the design of an application.

We used arbitrary values for the width and height of the ScrolledWindow; they were chosen because they seemed to work best. If you are using a ScrolledWindow with a number of other widgets in an interface, you do not need to specify an initial size for the ScrolledWindow. Since the ScrolledWindow is extremely flexible, you can allow its parent or its siblings to control its size. ScrolledWindows work well with PanedWindows because they can be adjusted easily. However, the ScrolledWindow does not have a sensible default size, so you should provide an initial geometry if the ScrolledWindow is going to control its own size. In this case, the size that you choose for the widget should be based on the aesthetics of the data that is being displayed.

In the example, the child of the ScrolledWindow is the `main_rc` widget, which is a RowColumn that contains all of the children that represent the password file information. After `getusers()` is called, the program loops through each item in the array of `UserInfo` structures and creates a Form widget that contains a ToggleButton, two Labels, and a Text widget. All of the Forms are stacked vertically on top of one another in the RowColumn.Once complete, the user can scroll around and access any of the elements without the application having to support any of the scrolling mechanisms because they are completely automated by the toolkit. In most cases, an application does not need to do anything other than what we described in this section to take advantage of automatic scrolling.

Working With ScrollBars

The ScrollBar is the backbone of the ScrolledWindow. Although the ScrollBar is a stand alone widget that can be created and manipulated without being the child of a ScrolledWindow, we are not going to discuss this usage because it is not consistent with the *Motif Style Guide*. The kinds of things that you can do with a ScrollBar individually are no more interesting than the sorts of things that you can do with them as children of ScrolledWindows, anyway. We are going to discuss how to control a ScrollBar directly from an application in the context of a ScrolledWindow widget. This information is useful if you want to monitor scrolling, if you want to fine-tune the way that automatic scrolling is handled, or if you want to implement application-defined scrolling.

Before we begin, it is important to understand that the ScrollBar does *not* handle scrolling itself. The widget merely reports scrolling actions through its callback routines. It is up to the internals of an application or a widget to install callback procedures on the ScrollBar that adjust the work window appropriately. The ScrollBar manages its own display in accordance with scrolling actions, so you do not need to update the ScrollBar's display unless the underlying data of the object being scrolled changes. To change the display, you can set resources that are associated with the different elements of the ScrollBar. Figure 10-3 illustrates the design of a ScrollBar and identifies its elements. This figure represents a vertical ScrollBar; a ScrollBar can also be oriented horizontally.

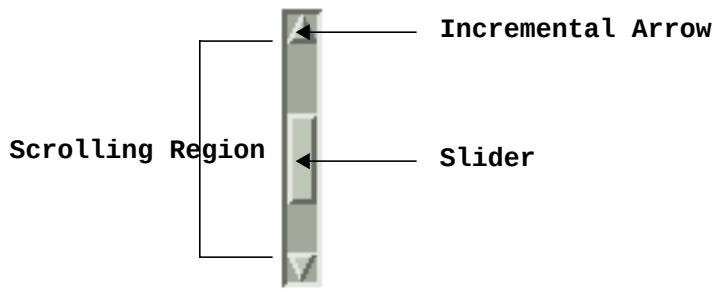


Figure 10-3 Elements of a ScrollBar

The appearance and behavior of a ScrollBar is directly related to the object that it scrolls. The relationship between the ScrollBar and the object it scrolls is proportional, so that the size of the slider in the ScrollBar represents how much of the object that is being scrolled is visible in the clip window. The size of the object being scrolled is broken down into equally sized units; the size of the units is called the *unit length*. When the user clicks on one of the incremental arrows (also called directional arrows), the ScrollBar scrolls in the direction indicated by the arrow in unit increments. It is important to realize that the unit length is stored and interpreted internally by the object being scrolled; it is of no interest to the ScrollBar itself, since it does not affect the display of the ScrollBar. While this value is not set on the ScrollBar itself, it plays a key role in understanding how ScrollBars work. All of the other resource values for the ScrollBar are measured in terms of the unit length. A Text widget might set its unit length for the vertical ScrollBar to the height of the tallest character in the widget's font set, plus some margin for whitespace on the top and bottom of the character. As a result, vertical scrolling adjusts the window so that the text is always displayed without lines being partially obscured. However, it is the Text widget's responsibility to know the unit length value. The unit length for the horizontal ScrollBar unit length might be the average width of the characters in the font that is being used.

The *value* of a ScrollBar is the offset, measured in unit lengths, of the data in the clip window from the object's origin. For example, if the top of the clip window displays the

fourth line of text in a Text widget, the ScrollBar is said to have a value of 3, since it is offset from 0. Clicking and dragging the slider directly changes the ScrollBar's value to an absolute number; clicking on either of the directional arrows changes the ScrollBar's value incrementally; clicking in the scrolling region, but not on the slider itself changes the ScrollBar's value by page lengths. The value is measured in units, not pixels.

The *view length* is the size of the viewable area (clip window), as measured in unit lengths. The vertical ScrollBar for a Text widget that is displaying 15 lines of text would have a view length of 15. The horizontal ScrollBar's view length would be the number of columns that the clip window can display.

The *page length* is measured in unit lengths and is usually one less than the view length. If the user scrolls the window by a page increment, the first line from the old view is retained as the last line in the new view for visual reference because otherwise, the user might lose her orientation.

Resources

Figure 10-4 illustrates the relationship between the elements listed above and introduces the ScrollBar resources that correspond to these values.

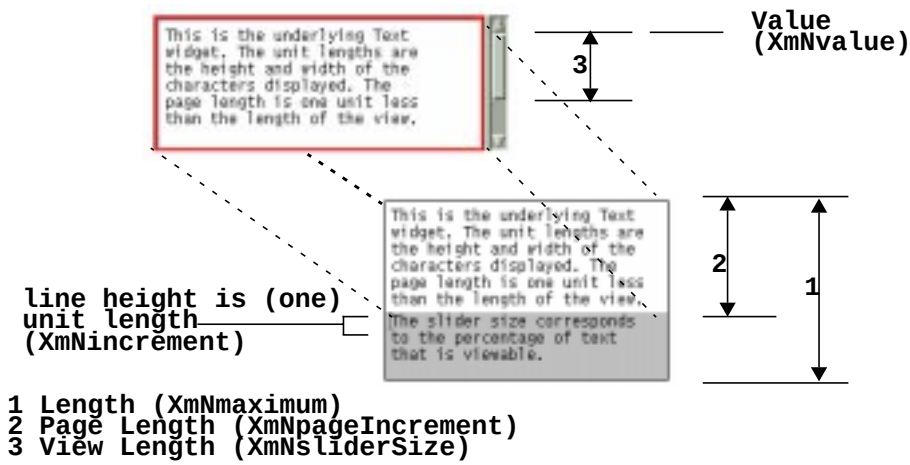


Figure 10-4 Conceptual relationship between ScrollBar and the object that it scrolls

The XmNincrement resource represents the number of units that the ScrollBar reports having scrolled when the user clicks on its incremental arrows. The value for XmNincrement in Figure 10-4 is 1 because each incremental scroll on the vertical ScrollBar should scroll the text one line. Internally, the Text widget knows that the number

of pixels associated with XmNincrement is the height of a line. For an automatic ScrolledWindow, it is rare to set the resource to any value other than 1.

The XmNpageIncrement resource specifies the number of units that the ScrollBar should report having scrolled when the user moves the ScrollBar by a page. Again, the ScrollBar doesn't actually perform the scrolling, it just reports the scrolling action. However, the ScrollBar does use this value to calculate the new visual position for the slider within the scrolling area and to update its display. The application can use this value, multiplied by pixels-per-unit, to determine the new data to display in the work window.

The XmNmaximum resource is the largest size, measured in unit increments, that the object can have. For the Text widget shown above, the value for XmNmaximum is 9.* The XmNminimum resource is the smallest size, measured in unit increments, that the object will ever have. The XmNsliderSize resource corresponds to the view length. The resource specifies the size of the clip window in unit lengths. For example, in Figure 10-4, the clip window can display six lines, so XmNsliderSize is 6.

The XmNvalue is the number of units that the data in the clip window is offset from the beginning of the work window. For example, if the Text widget has been scrolled down by four lines from the top, the value of the vertical ScrollBar's XmNvalue resource would be 4.

Example 10-2 demonstrates how the vertical ScrollBar resources get their values from a typical ScrolledText object.†

Example 10-2 The simple_sb.c program

```
/* simple_sb.c -- demonstrate the Scrollbar resource values from
** a ScrolledText object. This is used as an introductory examination
** of the resources used by Scrollbars.
*/
#include <Xm/ScrolledW.h>
#include <Xm/RowColumn.h>
#include <Xm/PushBG.h>
#include <Xm/Text.h>

/* print the "interesting" resource values of a scrollbar */
void get_sb (Widget widget, XtPointer client_data, XtPointer call_data)
{
    Widget  scrollbar = (Widget) client_data;
    int     increment=0, maximum=0, minimum=0;
    int     page_incr=0, slider_size=0, value=0;

    XtVaGetValues (scrollbar,  XmNincrement,     &increment,
                               XmNmaximum,       &maximum,
```

* The Motif Text widget sets its horizontal ScrollBar's XmNmaximum to the number of characters in its widest visible line, rather than the widest of all of its lines.

† XtVaAppInitialize() is considered deprecated in X11R6.

```
                                XmNminimum,       &minimum,
                                XmNpageIncrement, &page_incr,
                                XmNsliderSize,    &slider_size,
                                XmNvalue,         &value,
                                NULL);
    printf ("increment=%d, max=%d, min=%d, page=%d, slider=%d, value=%d\n",
    increment, maximum, minimum, page_incr, slider_size, value);
}

main (int argc, char *argv[])
{
    Widget         toplevel, rowcol, text_w, pb, sb;
    XtAppContext   app;
    Arg            args[10];
    int            n = 0;

    XtSetLanguageProc (NULL, NULL, NULL);
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv, NULL,
                                    sessionShellWidgetClass, NULL);
    /* RowColumn contains ScrolledText and PushButton */
    rowcol = XmCreateRowColumn (toplevel, "rowcol", NULL, 0);

    XtSetArg (args[n], XmNrows, 10);                    n++;
    XtSetArg (args[n], XmNcolumns, 80);                 n++;
    XtSetArg (args[n], XmNeditMode, XmMULTI_LINE_EDIT); n++;
    XtSetArg (args[n], XmNscrollHorizontal, False);     n++;
    XtSetArg (args[n], XmNwordWrap, True);              n++;
    text_w = XmCreateScrolledText (rowcol, "text_w", args, n);
    XtManageChild (text_w);

    /* get the scrollbar from ScrolledWindow associated with Text widget */
    XtVaGetValues (XtParent (text_w), XmNverticalScrollBar, &sb, NULL);

    /* provide a pushbutton to obtain the scrollbar's resource values */
    pb = XmCreatePushButtonGadget (rowcol, "Print ScrollBar Values", NULL, 0);
    XtAddCallback (pb, XmNactivateCallback, get_sb, sb);
    XtManageChild (pb);

    XtManageChild (rowcol);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}
```

This program simply displays a ScrolledText object and a PushButton. The ScrolledText object does not contain any text by default; you can cut and paste some text into the object. The graphical output of the program is displayed in Figure 10-5.

Figure 10-5 Output of the simple_sb program

When the PushButton is activated, it retrieves some resource values from the vertical ScrollBar of the Text widget's ScrolledWindow. These values are output to *stdout*. The following output shows some possible values for the different resources:

```
increment=1, max=12, min=0, page=9, slider=10, value=0
increment=1, max=12, min=0, page=9, slider=10, value=1
increment=1, max=25, min=0, page=9, slider=10, value=6
increment=1, max=25, min=0, page=9, slider=10, value=12
increment=1, max=25, min=0, page=9, slider=10, value=15
```

The value for `XmNincrement` is always `1`, which indicates that the incremental arrow buttons scroll the text by one unit in either direction. The value for `XmNmaximum` changes according to the number of lines of text that there are in the window. The value of `XmNminimum` is always `0` because this object can have as few as zero lines of text.

The values for `XmNsliderSize` and `XmNpageIncrement` are`10` and `9`, respectively. The values never changed because the ScrolledWindow was not resized. If it had been, the slider size and page increment values would have changed to match the new number of lines displayed in the window. The page increment is one less than the number of lines that can be displayed in the clip window, so that if the user scrolls by a page, the new view contains at least one of the previously-viewed lines for reference.

The value for `XmNvalue` varies depending on the line that is displayed at the top of the clip window. If the beginning of the text is displayed, `XmNvalue` is `0`. As the user scrolls through the text, the value for `XmNvalue` increases or decreases, but it is always a positive value.

Incidentally, you can adjust these resource values to get some different results. For example, you could set the `XmNincrement` resource to `2` in order to modify the number of lines that are scrolled when the user selects the incremental arrows. However, you should not change these resources arbitrarily, as you could really confuse the user.

As mentioned at the beginning of this section, the most important thing to remember about the ScrollBar widget is that it does not cause any actual scrolling of the object in the work

window. The widget merely reports scrolling activity through its callback routines. When scrolling occurs, it is the callback routines that are responsible for modifying the data in the work window, by adjusting elements or redrawing the image. The ScrollBar updates its own display according to the scrolling action. If the widget or the application that owns the callback routines fails to modify the display, the user will see an inconsistency between the ScrollBar display and the data in the clip window.

Orientation

Two ScrollBar resources that are closely related are `XmNorientation` and `XmNprocessingDirection`. These resources specify the horizontal or vertical orientation of the ScrollBar and its normal processing direction. The value for `XmNorientation` can be either `XmHORIZONTAL` or `XmVERTICAL`. When a ScrollBar is oriented horizontally, the normal processing direction for it is such that the minimum value is on the left and the maximum is on the right. When the orientation is vertical, the minimum is on the bottom and the maximum is on the top. You can change the processing direction using the `XmNprocessingDirection` resource. This resource can have the following values:

`XmMAX_ON_LEFT`	`XmMAX_ON_RIGHT`
`XmMAX_ON_TOP`	`XmMAX_ON_BOTTOM`

These values only need to be changed when the user's environment is such that the natural language for the locale is read from right-to-left. In this case, the `XmNscrollBarPlacement` resource for the ScrolledWindow needs to be changed to match the processing direction. This resource can have the following values:

`XmTOP_LEFT`	`XmTOP_RIGHT`
`XmBOTTOM_LEFT`	`XmBOTTOM_RIGHT`

Visual and Input Resources

The ScrollBar is generally used as an input-output component: the slider displays the relationship between the viewport and the underlying work area, the user moves the slider to move the viewport. The ScrollBar however can be used as a read-only component, and has resources to modify its visuals depending upon the required presentation. A read-only ScrollBar can be effected by setting the `XmNeditable` resource to `False`.

Usually, a ScrollBar has a slider which moves unattached to the ends of the trough. It is possible to make one end of the slider attached so that the ScrollBar behaves like an old-fashioned spirit thermometer. The resource to modify for this behavior is `XmNslidingMode`*: the default is `XmSLIDER`, which gives the familiar unfixed slider. A value of `XmTHERMOMETER` fixes the slider at one end.

* `XmNslidingMode` is available from Motif 2.0 onwards.

The general appearance of the slider itself can be modified through the XmNsliderMark* resource. The possible values are: XmETCHED_LINE, XmNONE, XmROUND_MARK, XmTHUMB_MARK. XmNONE simply draws a rectangle in the foreground color of the widget.

Figure 10-6 shows the effect of setting this resource to the various values.

Figure 10-6 The various settings of the XmNsliderMark resource

The coloration algorithm of the slider is configurable through the XmNsliderVisual† resource. The choices are between painting in the widget's trough color (XmTROUGH), in the background color (XmBACKGROUND_COLOUR), in the foreground (XmFOREGROUND), or in a shadowed background (XmSHADOWED_BACKGROUND).

The arrows at the ends of the ScrollBar can also be configured. The resource XmNshowArrows‡ can be set to display arrows on either end (XmEACH_SIDE), the arrows at the maximum end (XmMAX_SIDE), the arrows at the minimum end (XmMIN_SIDE), or no arrows at all (XmNONE). Figure 10-7 shows the various arrangements.

* XmNsliderMark is available from Motif 2.0 onwards.

† XmNsliderVisual is available fro Motif 2.0 onwards.

‡ In Motif 1.2, XmNshowArrows is a simple Boolean: show both arrows, or neither.

Figure 10-7 The various settings of the XmNshowArrows resource

Most of the visual resources mentioned in this section are utilized by the Scale widget, which internally uses a ScrollBar to display the widget value.

Callback Routines

The callback routines associated with the ScrollBar are its only links into the internal mechanisms that actually scroll the data. You can use these callback routines in various contexts, depending on what you want to accomplish. For example, you can monitor scrolling in an automatic or semi-automatic ScrolledWindow, such as a ScrolledText or ScrolledList object. These two activities are identical when it comes to the implementation of what we are about to describe. You can also implement application-defined scrolling, which requires intimate knowledge of the internals of the object being scrolled.

There are different parts of a ScrollBar that the user can manipulate to cause a scrolling action. In fact, each part of the ScrollBar has a separate callback routine associated with it. These callback routines are used both to monitor automatic (or semi-automatic) scrolling and to implement application-defined scrolling. As with all Motif callbacks, the callback routines take the form of an `XtCallbackProc`. All of the ScrollBar callbacks pass a structure of type `XmScrollBarCallbackStruct` for the third parameter. This structure takes the following form:

```
typedef struct {
    int         reason;
    XEvent      *event;
    int         value;
    int         pixel;
} XmScrollBarCallbackStruct;
```

The `reason` field specifies the scrolling action performed by the user. Each callback has a corresponding reason that indicates the action. Table 10-1 lists the callback name, reason, and scrolling action for each ScrollBar callback resource.

Table 10-1Callback Resources for the ScrollBar Widget

Resource Name	Reason	Action
XmNincrementCallback	XmCR_INCREMENT	Top or right directional arrow clicked
XmNdecrementCallback	XmCR_DECREMENT	Bottom or left directional arrow clicked
XmNpageIncrementCallback	XmCR_PAGE_INCREMENT	Area above or right of slider clicked
XmNpageDecrementCallback	XmCR_PAGE_DECREMENT	Area below or left of slider clicked
XmNtoTopCallback	XmCR_TO_TOP	Top or right directional arrow CTRL-clicked
XmNtoBottomCallback	XmCR_TO_BOTTOM	Bottom or left directional arrow CTRL-clicked
XmNdragCallback	XmCR_DRAG	Slider dragged
XmNvalueChangedCallback	XmCR_VALUE_CHANGED	Value changed (see explanation)

The scrolling action that invokes the various increment and decrement callbacks depends on the value of the `XmNprocessingDirection` resource; the table shows the actions for a left-to-right environment. The `XmNvalueChangedCallback` is invoked when the user releases the mouse button after dragging the slider. The callback is also invoked for any of the other scrolling actions if the corresponding callback resource is not set, with the exception of the `XmNdragCallback`. This feature is convenient for cases where you are handling your own scrolling and you are not concerned with the type of scrolling the user invoked.

The `value` field of the callback structure indicates the new position of the ScrollBar. This value can range from `XmNminimum` to `XmNmaximum`. The `pixel` field indicates the *x* or *y* coordinate of the mouse location relative to the origin of the ScrollBar for the `XmNtoTopCallback`, `XmNtoBottomCallback`, and `XmNdragCallback` routines. The origin is the top of a vertical ScrollBar or the left side of a horizontal ScrollBar, regardless of the value of `XmNprocessingDirection`.

Example 10-3 demonstrates how a callback routine can be hooked up to each of the callback resources to allow you to monitor the scrolling in a List widget more precisely. For Text and List widgets, you really should not be using the callback routines to change the default scrolling behavior.*

* `XtVaAppInitialize`() is considered deprecated in X11R6.

Example 10-3 The monitor_sb.c program

```
/* monitor_sb.c -- demonstrate the ScrollBar callback routines by
** monitoring the ScrollBar for a ScrolledList. Functionally, this
** program does nothing. However, by tinkering with the Scrolled
** List and watching the output from the ScrollBar's callback routine,
** you'll see some interesting behavioral patterns. By interacting
** with the *List* widget to cause scrolling, the ScrollBar's callback
** routine is never called. Thus, monitoring the scrolling actions
** of a ScrollBar should not be used to keep tabs on exactly when
** the ScrollBar's value changes!
*/
#include <Xm/List.h>

/* print the interesting resource values of a scrollbar */
void scroll_action (Widget scrollbar, XtPointer client_data,
                    XtPointer call_data)
{
    XmScrollBarCallbackStruct *cbs =
                            (XmScrollBarCallbackStruct *) call_data;

    printf ("cbs->reason: %s, cbs->value = %d, cbs->pixel = %d\n",
    cbs->reason == XmCR_DRAG? "drag" :
    cbs->reason == XmCR_VALUE_CHANGED? "value changed" :
    cbs->reason == XmCR_INCREMENT? "increment" :
    cbs->reason == XmCR_DECREMENT? "decrement" :
    cbs->reason == XmCR_PAGE_INCREMENT? "page increment" :
    cbs->reason == XmCR_PAGE_DECREMENT? "page decrement" :
    cbs->reason == XmCR_TO_TOP? "top" :
    cbs->reason == XmCR_TO_BOTTOM? "bottom" : "unknown",
    cbs->value, cbs->pixel);
}

main (int argc, char *argv[])
{
    Widget         toplevel, list_w, sb;
    XtAppContext   app;
    char           *items = "choice0, choice1, choice2, choice3, choice4,\
                             choice5, choice6, choice7, choice8, choice9, \
                             choice10, choice11, choice12, choice13, \
                             choice14";

    XtSetLanguageProc (NULL, NULL, NULL);
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv, NULL,
                                    sessionShellWidgetClass, NULL, 0);
    list_w = XmCreateScrolledList (toplevel, "list_w", NULL, 0);
    XtVaSetValues (list_w,
            /* Rather than convert the entire list of items into an array
            ** of compound strings, let's just let Motif's type converter
            ** do it and save lots of effort (although not much time).
            */
            XtVaTypedArg, XmNitems, XmRString, items, strlen (items)+1,
            XmNitemCount, 15,
            XmNvisibleItemCount, 5,
            NULL);
```

```
        XtManageChild (list_w);

        /* get the scrollbar from ScrolledWindow associated with Text widget */
        XtVaGetValues (XtParent (list_w), XmNverticalScrollBar, &sb, NULL);

        XtAddCallback (sb, XmNvalueChangedCallback, scroll_action, NULL);
        XtAddCallback (sb, XmNdragCallback, scroll_action, NULL);
        XtAddCallback (sb, XmNincrementCallback, scroll_action, NULL);
        XtAddCallback (sb, XmNdecrementCallback, scroll_action, NULL);
        XtAddCallback (sb, XmNpageIncrementCallback, scroll_action, NULL);
        XtAddCallback (sb, XmNpageDecrementCallback, scroll_action, NULL);
        XtAddCallback (sb, XmNtoTopCallback, scroll_action, NULL);
        XtAddCallback (sb, XmNtoBottomCallback, scroll_action, NULL);

        XtRealizeWidget (toplevel);
        XtAppMainLoop (app);
    }
```

The program displays a simple ScrolledList that contains 15 entries, as shown in Figure 10-8.

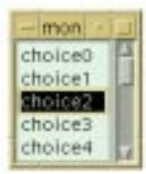


Figure 10-8 Output of the monitor_sb program

The entries in the List are not important; the way that the ScrollBar reacts to the user's interaction is what is interesting. The following output shows what happens when the user scrolls the List:

```
cbs->reason: increment, cbs->value = 1, cbs->pixel = 0
cbs->reason: page increment, cbs->value = 5, cbs->pixel = 0
cbs->reason: drag, cbs->value = 6, cbs->pixel = 46
cbs->reason: drag, cbs->value = 7, cbs->pixel = 50
cbs->reason: value changed, cbs->value = 7, cbs->pixel = 50
cbs->reason: decrement, cbs->value = 6, cbs->pixel = 0
cbs->reason: top, cbs->value = 0, cbs->pixel = 11
```

If you use the keyboard to select elements or scroll around in the list, you'll notice that the callbacks for the ScrollBar are not invoked because the List widget is taking all of the keyboard events from the ScrollBar. Like any other widget, the ScrollBar can receive keyboard events, and it even has translations to map certain key sequences to scrolling actions. However, the List widget sets `XmNtraversalOn` to `False` for the ScrollBar, so that the List can process its own keyboard actions, some of which scroll the window. The Text widget does the same thing with its ScrollBars. As a result, there is a limit to what you can accomplish by monitoring ScrollBar actions on semi-automatic scrolling objects like List and Text widgets.

Implementing True Application-defined Scrolling

In this section, we pull together what we've learned in this chapter and put it to work to implement application-defined scrolling. We are going to use an example that displays a large number of individual bitmaps in a ScrolledWindow, so that the user can view all of the bitmaps by scrolling the window. The intent is to make the appearance and functionality of the ScrolledWindow mimic the automatic scrolling mode as much as possible.

There are actually several ways to go about writing this program, depending on the constraints that we impose. The simplest method is to render each bitmap into one large pixmap and use that pixmap as the `XmNlabelPixmap` for a Label widget. The Label widget can then be used as the work window for an automatic ScrolledWindow. This design is similar to most of the other examples of ScrolledWindows used throughout the book. However, we want to add a constraint such that each incremental scrolling action causes the display to shift by one bitmap cell, so that the top and left sides of the viewport always show a full bitmap. In other words, no partially-displayed bitmaps are allowed. Furthermore, when the user drags the slider, we want the display to scroll in cell-increments, not pixel-by-pixel.

The constraints that we just described define the behavior that the List and Text widgets use for their own displays. Like those widgets, our example program has a conceptual unit size that is represented by the object being scrolled. For the Text and List widgets, the unit size is the height and width of the font used by the entries. For our bitmap viewer, the heights and widths of the bitmaps vary more dramatically than the characters in a font, so for consistency, the unit size is set to the largest of all of the bitmaps. The design of our program is based on the same principles used by the ScrolledWindow's automatic scrolling method. Only in this case, we are going to do the work ourselves. The reason that we need to use application-defined scrolling is that the automatic scrolling method cannot support the scrolling constraints described above; there is no way to change the number of pixels per scrolling unit with an automatic ScrolledWindow.

In our implementation, the work window is a DrawingArea widget whose size is constrained by the size of the viewport in the ScrolledWindow. Initially, the ScrolledWindow sizes itself to the size of the DrawingArea widget, but once the program is running, the size of the DrawingArea is changed by the ScrolledWindow as it is resized. The bitmaps are rendered into a large pixmap, which is rendered into the DrawingArea in connection with scrolling actions. The offset of the pixmap and how much of it is copied into the DrawingArea is controlled by the application, following the same algorithm that the ScrolledWindow uses in automatic scrolling mode. The only difference is that we can adjust for the pixels-per-unit value, whereas the automatic ScrolledWindow is only aware of single-pixel units.

Proper scrolling is not a particularly difficult problem to solve, as it only involves simple arithmetic. The real problem is handling the case where the user or the application causes the ScrolledWindow to resize, since this action changes all of the variables in the calculation. When resizing happens, the ScrolledWindow passes that resizing onto the DrawingArea widget, which must recalculate its size and update the ScrollBar resources so that the display and the graphic representation match. Basically, the program has to solve four independent problems:

1. Read the bitmaps and load them into a sufficiently large pixmap.
2. Create the ScrolledWindow, a DrawingArea widget, and two ScrollBars; the program must initialize each of these widgets' resources so that the ratio between their sizes and the size of the pixmap is consistent.
3. Set up a callback routine for the ScrollBars to respond to scrolling actions.
4. Provide a callback routine for the DrawingArea widget's `XmNresizeCallback` that updates all of the widgets' resources according to the new ratio between the widgets and the pixmap.

Although each of these problems has a simple solution, when combined the general solution becomes quite complex. Rather than trying to solve each problem individually, a well-designed application integrates the solutions to the problems into a single, elegant design. Example 10-4 demonstrates our implementation of the bitmap viewer. Although the program is quite long, you can follow along with the comments embedded in the code to understand what is going on.*

Example 10-4 The app_scroll.c program

```
/* app_scroll.c - Displays bitmaps specified on the command line. All
** bitmaps are drawn into a pixmap, which is rendered into a DrawingArea
** widget, which is used as the work window for a ScrolledWindow. This
** method is only used to demonstrate application-defined scrolling for
** the motif ScrolledWindow. Automatic scrolling is much simpler, but
** does not allow the programmer to impose incremental scrolling units.
**
** Bitmaps are displayed in an equal number of rows and columns if possible.
**
** Example:
** app_scroll /usr/X11R6/include/X11/bitmaps/*
*/

#include <stdio.h>
#include <strings.h>
#include <Xm/ScrolledW.h>
#include <Xm/DrawingA.h>
#include <Xm/ScrollBar.h>
```

* `XtVaAppInitialize`() is considered deprecated in X11R6. `XmScrolledWindowSetAreas`() is deprecated in Motif 2.0.

```
#ifdef max /* just in case--we don't know, but these are commonly set */
#undef max /* by arbitrary unix systems. Also, we cast to int! */
#endif

/* redefine "max" and "min" macros to take into account "unsigned" values */
#define max(a,b) ((int)(a)>(int)(b)?(int)(a):(int)(b))
#define min(a,b) ((int)(a)<(int)(b)?(int)(a):(int)(b))

/* don't accept bitmaps larger than 100x100.. This value is arbitrarily
** chosen, but is sufficiently large for most images. Handling extremely
** large bitmaps would eat too much memory and make the interface awkward.
**/

#define MAX_WIDTH 100
#define MAX_HEIGHT 100
typedef struct {
    char           *name;
    int            len;         /* strlen(name) */
    unsigned int   width, height;
    Pixmap         bitmap;
} Bitmap;

/* get the integer square root of n -- used to calculate an equal
** number of rows and columns for a given number of elements.
*/
int int_sqrt (register int n)
{
    register int i, s = 0, t;
    for (i = 15; i >= 0; i--) {
        t = (s | (1L << i));
        if (t * t <= n)
            s = t;
    }
    return s;
}

/* Global variables */
Widget             drawing_a, vsb, hsb;
Pixmap             pixmap; /* used as the image for DrawingArea widget */
Display            *dpy;
Dimension          view_width = 300, view_height = 300;
int                rows, cols;
unsigned int       cell_width, cell_height;
unsigned int       pix_hoffset, pix_voffset, sw_hoffset, sw_voffset;
void               redraw(Window);

main (int argc, char *argv[])
{
    extern char    *strcpy();
    XtAppContext   app;
    Widget         toplevel, scrolled_w;
    Bitmap         *list = (Bitmap *) NULL;
    GC             gc;
```

```
char           *p;
XFontStruct    *font;
int            i = 0, j = 0, k = 0, total = 0;
unsigned int   bitmap_error;
Arg            args[6];
void           scrolled(Widget, XtPointer, XtPointer);
void           expose_resize(Widget, XtPointer, XtPointer);

XtSetLanguageProc (NULL, NULL, NULL);
toplevel = XtVaOpenApplication (&app, argv[0], NULL, 0, &argc, argv, NULL,
                                sessionShellWidgetClass, NULL);
dpy = XtDisplay (toplevel);
font = XLoadQueryFont (dpy, "fixed");
/* load bitmaps from filenames specified on command line */
while (*++argv) {
    printf ("Loading \"%s\"...", *argv), fflush (stdout);
    if (i == total) {
        total += 10; /* allocate bitmap structures in groups of 10 */
        if (!(list = (Bitmap *) XtRealloc ((char *) list,
                            total * sizeof (Bitmap))))
            XtError ("Not enough memory for bitmap data");
    }
    /* read bitmap file using standard X routine. Save the resulting
    ** image if the file isn't too big.
    */
    if ((bitmap_error = XReadBitmapFile (dpy, DefaultRootWindow (dpy),
                                *argv, &list[i].width, &list[i].height,
                                &list[i].bitmap, &j, &k)) ==
                                BitmapSuccess) {
        /* Get just the base filename (minus leading pathname)
        ** We save this value for later use when we caption the bitmap.
        */
        if (p = rindex (*argv, '/'))
            p++;
        else
            p = *argv;

        if (list[i].width > MAX_WIDTH || list[i].height > MAX_HEIGHT) {
            printf ("%s: bitmap too big\n", p);
            XFreePixmap (dpy, list[i].bitmap);
            continue;
        }

        list[i].len = strlen (p);
        list[i].name = p; /* we'll be getting it later */
        printf ("Size: %dx%d\n", list[i].width, list[i].height);
        i++;
    } else {
        printf ("Couldn't load bitmap: \"%s\": ", *argv);

        switch (bitmap_error) {
            case BitmapOpenFailed  : puts ("Open failed."); break;
            case BitmapFileInvalid : puts ("Bad file format."); break;
            case BitmapNoMemory    : puts ("Not enough memory."); break;
```

```
        }
    }
}

if ((total = i) == 0) {
    puts ("Couldn't load any bitmaps.");
    exit (1);
}
printf ("Total bitmaps loaded: %d\n", total);
/* calculate size for pixmap by getting the dimensions of each. */
printf ("Calculating sizes for pixmap..."),fflush (stdout);
for (i = 0; i < total; i++) {
    if (list[i].width > cell_width)   cell_width = list[i].width;
    if (list[i].height > cell_height) cell_height = list[i].height;
    /* the bitmap's size is one thing, but its caption may exceed it */
    if ((j = XTextWidth (font, list[i].name, list[i].len)) > cell_width)
        cell_width = j;
}
/* compensate for font in vertical dimension; add a 6 pixel padding */
cell_height += 6 + font->ascent + font->descent;
cell_width += 6;
cols = int_sqrt (total);
rows = (total + cols-1)/cols;
printf ("Creating pixmap area of size %dx%d (%d rows, %d cols)\n",
        cols * cell_width, rows * cell_height, rows, cols);

/* Create a single, 1-bit deep pixmap */
if (!(pixmap = XCreatePixmap (dpy, DefaultRootWindow (dpy),
                              cols * cell_width + 1,
                              rows * cell_height + 1, 1)))
    XtError ("Can't Create pixmap");
if (!(gc = XCreateGC (dpy, pixmap, NULL, 0)))
    XtError ("Can't create gc");
XSetForeground (dpy, gc, 0); /* 1-bit deep pixmaps use 0 as background */
/* Clear the pixmap by setting the entire image to 0's */
XFillRectangle (dpy, pixmap, gc, 0, 0,
                cols * cell_width, rows * cell_height);
XSetForeground (dpy, gc, 1); /* Set the foreground to 1 (1-bit deep) */
XSetFont (dpy, gc, font->fid); /* print bitmap filenames (captions) */

/* Draw the grid lines between bitmaps */
for (j = 0; j <= rows * cell_height; j += cell_height)
    XDrawLine (dpy, pixmap, gc, 0, j, cols * cell_width, j);
for (j = 0; j <= cols * cell_width; j += cell_width)
    XDrawLine (dpy, pixmap, gc, j, 0, j, rows*cell_height);
/* Draw each of the bitmaps into the big picture */
for (i = 0; i < total; i++) {
    int x = cell_width * (i % cols);
    int y = cell_height * (i / cols);
    XDrawString (dpy, pixmap, gc, x + 5, y + font->ascent,
    list[i].name, list[i].len);
    XCopyArea (dpy, list[i].bitmap, pixmap, gc, 0, 0, list[i].width,
                list[i].height, x + 5,
                y + font->ascent + font->descent);
```

```
    /* Once we copy it into the big picture, we don't need the bitmap */
    XFreePixmap (dpy, list[i].bitmap);
}
XtFree ((char *) list); /* don't need the array of structs anymore */
XFreeGC (dpy, gc); /* nor do we need this GC */

/* Create automatic Scrolled Window */
i = 0;
XtSetArg (args[i], XmNscrollingPolicy, XmAPPLICATION_DEFINED); i++;
XtSetArg (args[i], XmNvisualPolicy, XmVARIABLE); i++;
scrolled_w = XmCreateScrolledWindow (toplevel, "scrolled_w", args, i);

/* Create a drawing area as a child of the ScrolledWindow.
** The DA's size is initialized (arbitrarily) to view_width and
** view_height. The ScrolledWindow will expand to this size.
*/
i = 0;
XtSetArg (args[i], XmNwidth, view_width); i++;
XtSetArg (args[i], XmNheight, view_height); i++;
drawing_a = XmCreateDrawingArea (scrolled_w, "drawing_a", args, i);
XtAddCallback (drawing_a, XmNexposeCallback, expose_resize, NULL);
XtAddCallback (drawing_a, XmNresizeCallback, expose_resize, NULL);
XtManageChild (drawing_a);

/* Application-defined ScrolledWindows won't create their own
** ScrollBars. So, we create them ourselves as children of the
** ScrolledWindow widget. The vertical ScrollBar's maximum size is
** the number of rows that exist (in unit values). The horizontal
** ScrollBar's maximum width is represented by the number of columns.
*/
i = 0;
XtSetArg (args[i], XmNorientation, XmVERTICAL); i++;
XtSetArg (args[i], XmNmaximum, rows); i++;
XtSetArg (args[i], XmNsliderSize,
min (view_height / cell_height, rows));
i++;
XtSetArg (args[i], XmNpageIncrement,
max ((view_height / cell_height) - 1, 1));
i++;
vsb = XmCreateScrollBar (scrolled_w, "vsb", args, i);
XtManageChild (vsb);

if (view_height / cell_height > rows)
    sw_voffset = (view_height - rows * cell_height) / 2;
i = 0;
XtSetArg (args[i], XmNorientation, XmHORIZONTAL); i++;
XtSetArg (args[i], XmNmaximum, cols); i++;
XtSetArg (args[i], XmNsliderSize,
min (view_width / cell_width, cols));
i++;
XtSetArg (args[i], XmNpageIncrement,
max ((view_width / cell_width) - 1, 1));
i++;
hsb = XmCreateScrollBar (scrolled_w, "hsb", args, i);
```

```
    XtManageChild (hsb);

    if (view_width / cell_width > cols)
        sw_hoffset = (view_width - cols * cell_width) / 2;

    /* Allow the ScrolledWindow to initialize itself accordingly...*/
    XtVaSetValues (scrolled_w, XmNhorizontalScrollBar, hsb,
                               XmNverticalScrollBar, vsb,
                               XmNworkWindow, drawing_a,
                               NULL);

    /* use same callback for both ScrollBars and all callback reasons */
    XtAddCallback (vsb, XmNvalueChangedCallback,
                    scrolled, (XtPointer) XmVERTICAL);
    XtAddCallback (hsb, XmNvalueChangedCallback,
                    scrolled, (XtPointer) XmHORIZONTAL);
    XtAddCallback (vsb, XmNdragCallback,
                    scrolled, (XtPointer) XmVERTICAL);
    XtAddCallback (hsb, XmNdragCallback,
                    scrolled, (XtPointer) XmHORIZONTAL);

    XtManageChild (scrolled_w);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

/* React to scrolling actions. Reset position of ScrollBars; call redraw()
** to do actual scrolling. cbs->value is ScrollBar's new position.
*/
void scrolled (Widget scrollbar, XtPointer client_data,
XtPointer call_data)
{
    int orientation = (int) client_data; /* XmVERTICAL or XmHORIZONTAL */
    XmScrollBarCallbackStruct *cbs = (XmScrollBarCallbackStruct *) call_data;

    if (orientation == XmVERTICAL) {
        pix_voffset = cbs->value * cell_height;
        if (((rows * cell_height) - pix_voffset) > view_height)
            XClearWindow (dpy, XtWindow (drawing_a));
    } else {
        pix_hoffset = cbs->value * cell_width;
        if (((cols * cell_width) - pix_hoffset) > view_width)
            XClearWindow (dpy, XtWindow (drawing_a));
    }
    redraw (XtWindow (drawing_a));
}


/* This function handles both expose and resize (configure) events.
** For XmCR_EXPOSE, just call redraw() and return. For resizing,
** we must calculate the new size of the viewable area and possibly
** reposition the pixmap's display and position offsets. Since we
** are also responsible for the ScrollBars, adjust them accordingly.
*/
```

```
void expose_resize (Widget drawing_a, XtPointer client_data,
                    XtPointer call_data)
{
    Dimension                   new_width, new_height, oldw, oldh;
    Boolean                     do_clear = False;
    XmDrawingAreaCallbackStruct  *cbs =
                        (XmDrawingAreaCallbackStruct *) call_data;

    if (cbs->reason == XmCR_EXPOSE) {
        redraw (cbs->window);
        return;
    }
    oldw = view_width;
    oldh = view_height;
    /* Unfortunately, the cbs->event field is NULL, so we have to have
    ** get the size of the drawing area manually. A mis-design of
    ** the DrawingArea widget--not a bug (technically).
    */
    XtVaGetValues (drawing_a, XmNwidth, &view_width,
    XmNheight, &view_height, NULL);
    /* Get the size of the viewable area in "units lengths" where
    ** each unit is the cell size for each dimension. This prevents
    ** rounding error for the pix_voffset and pix_hoffset values later.
    */
    new_width = view_width / cell_width;
    new_height = view_height / cell_height;
    /* When the user resizes the frame bigger, expose events are generated,
    ** so that's not a problem, since the expose handler will repaint the
    ** whole viewport. However, when the window resizes smaller, no
    ** expose event is generated. The window does not need to be
    ** redisplayed if the old viewport was smaller than the pixmap.
    ** (The existing image is still valid--no redisplay is necessary.)
    ** The window WILL need to be redisplayed if:
    **   1) new view size is larger than pixmap (pixmap needs to be centered).
    **   2) new view size is smaller than pixmap, but the OLD view size was
    **      larger than pixmap.
    */
    if ((int) new_height >= rows) {
        /* The height of the viewport is taller than the pixmap, so set
        ** pix_voffset = 0, so the top origin of the pixmap is shown,
        ** and the pixmap is centered vertically in viewport.
        */
        pix_voffset = 0;
        sw_voffset = (view_height - rows * cell_height)/2;
        /* Case 1 above */
        do_clear = True;
        /* scrollbar is maximum size */
        new_height = rows;
    } else {
        /* Pixmap is larger than viewport, so viewport will be completely
        ** redrawn on the redisplay. (So, we don't need to clear window.)
        ** Make sure upper side has origin of a cell (bitmap).
        */
        pix_voffset = min (pix_voffset, (rows-new_height) * cell_height);
```

```
            sw_voffset = 0; /* no centering is done */
            /* Case 2 above */
            if (oldh > rows * cell_height)
                do_clear = True;
        }
        XtVaSetValues (vsb,  XmNsliderSize, max (new_height, 1),
                             XmNvalue, pix_voffset / cell_height,
                             XmNpageIncrement, max (new_height-1, 1),
                             NULL);
        /* identical to vertical case above */
        if ((int) new_width >= cols) {
            /* The width of the viewport is wider than the pixmap, so set
            ** pix_hoffset = 0, so the left origin of the pixmap is shown,
            ** and the pixmap is centered horizontally in viewport.
            */
            pix_hoffset = 0;
            sw_hoffset = (view_width - cols * cell_width)/2;
            /* Case 1 above */
            do_clear = True;
            /* scrollbar is maximum size */
            new_width = cols;
        } else {
            /* Pixmap is larger than viewport, so viewport will be completely
            ** redrawn on the redisplay. (So, we don't need to clear window.)
            ** Make sure left side has origin of a cell (bitmap).
            */
            pix_hoffset = min (pix_hoffset, (cols-new_width)*cell_width);
            sw_hoffset = 0;
            /* Case 2 above */
            if (oldw > cols * cell_width)
                do_clear = True;
        }
        XtVaSetValues (hsb,  XmNsliderSize, max (new_width, 1),
                             XmNvalue, pix_hoffset / cell_width,
                             XmNpageIncrement, max (new_width-1, 1),
                             NULL);
        if (do_clear) {
            /* XClearWindow() doesn't generate an ExposeEvent */
            /* all 0's means the whole window */
            XClearArea (dpy, cbs->window, 0, 0, 0, 0, True);
        }
    }

    void redraw (Window window)
    {
        static GC gc; /* static variables are *ALWAYS* initialized to NULL */
        if (!gc) {
            /* !gc means that this GC hasn't yet been created. */
            /* We create our own gc because the other one is based on a 1-bit
            ** bitmap and the drawing area window might be color (multiplane).
            ** Remember, we're rendering a multiplane pixmap, not the original
            ** single-plane bitmaps!
            */
            gc = XCreateGC (dpy, window, NULL, 0);
```

```
            XSetForeground (dpy, gc,
            BlackPixelOfScreen (XtScreen (drawing_a)));
            XSetBackground (dpy, gc,
            WhitePixelOfScreen (XtScreen (drawing_a)));
        }

        if (DefaultDepthOfScreen (XtScreen (drawing_a)) > 1)
            XCopyPlane (dpy, pixmap, window, gc, pix_hoffset, pix_voffset,
                        view_width, view_height, sw_hoffset, sw_voffset, 1L);
        else
            XCopyArea (dpy, pixmap, window, gc, pix_hoffset, pix_voffset,
                        view_width, view_height, sw_hoffset, sw_voffset);
    }
```

The bitmaps to be displayed are specified on the command line, as shown in the following command:

```
% app_scroll /usr/X11R6/include/X11/bitmaps/*
```

The output of this command is shown in Figure 10-9.

Figure 10-9 Output of the app_scroll program

The program begins by loading the bitmaps into an array of `Bitmap` structures that are specially designed for this application. Since each bitmap can have a different size, we save all of the information about them for comparison after they are all loaded. At that time, the largest bitmap is found and its size is used as the cell size for the viewer. The pixmap is created with a single-plane (a bitmap), since color is not used to render the standard X11 bitmaps when they are created. This pixmap is used as a virtual work window; its contents are rendered into the real DrawingArea work window.

After the bitmaps are loaded, the ScrolledWindow and DrawingArea are created. The DrawingArea has `XmNexposeCallback` and `XmNresizeCallback` callbacks installed so that the pixmap can be rendered or repositioned within the DrawingArea at any time.

Resizing does not change the pixmap, but it may cause its origin to be repositioned relative to the DrawingArea widget. We create the ScrollBars explicitly, since they are not created automatically when XmNscrollingPolicy is set to XmAPPLICATION_DEFINED. The ScrollBars are created as children of the ScrolledWindow, as shown in the following fragment:

```
XtSetArg (args[i], XmNorientation, XmVERTICAL); i++;
XtSetArg (args[i], XmNmaximum, rows); i++;
XtSetArg (args[i], XmNsliderSize,
min (view_height / cell_height, rows));
i++;
XtSetArg (args[i], XmNpageIncrement,
max ((view_height / cell_height) - 1, 1));
i++;
vsb = XmCreateScrollBar (scrolled_w, "vsb", args, i);
XtManageChild (vsb);

if (view_height / cell_height > rows)
    sw_voffset = (view_height - rows * cell_height) / 2;
```

The ScrollBars are initialized so that the XmNmaximum values are set to the number of rows and columns in the pixmap. Similarly, XmNsliderSize is set to the number of bitmap cells that can fit in the viewport in the horizontal and vertical dimensions. Internally, the application knows how many pixels each scrolling unit represents, since there is no ScrollBar resource for this value. The variables sw_hoffset and sw_voffset are used when the pixmap is smaller than the actual ScrolledWindow. In this case, the variables indicate the origin of the pixmap in the DrawingArea, so that the pixmap appears centered, as shown in Figure 10-10.

Figure 10-10 Output of the app_scroll program when the viewport is larger than the pixmap

The call to `XtVaSetValues()` specifying the `XmNhorizontalScrollBar`, `XmNverticalScrollBar`, and `XmNworkWindow` resources initializes the ScrolledWindow appropriately. This function assigns the ScrollBars and the DrawingArea widget to internal variables within the ScrolledWindow, so that the widget functions properly. While this call is opaque for automatic scrolling, it must be done for application-defined scrolling.

The ScrollBars are assigned a callback routine for the `XmNvalueChangedCallback` and `XmNdragCallback` callbacks. The `scrolled()` routine handles all of the scrolling actions, including incremental and page scrolling, that cause the value of the ScrollBar to change. We pass the values `XmHORIZONTAL` and `XmVERTICAL` as `client_data`, so that the routine knows which of the two ScrollBars invoked it. The routine determines the portion of the pixmap that should be rendered in the DrawingArea by calculating offsets into the pixmap. These offsets are calculated by multiplying the value of the ScrollBar by the pixels-per-unit value for the pixmap.

Finally, the top-level widget is realized and the main loop is started. At this point, the DrawingArea is realized, so the `XmNexposeCallback is` activated, which causes the DrawingArea to draw itself and display the first image of the pixmap. The function `expose_resize()` handles both the `Expose` and `ConfigureNotify` (resize) events. The function determines which event was delivered by checking the `reason` field of the callback structure passed to the function. When the DrawingArea is resized, we need to adjust a number of resources so that the pixmap is scrolled properly. For `Expose` events, no recalculation of variables is necessary, so all we need to do is redraw the display using `redraw()`.

The position at which the pixmap is rendered into the DrawingArea's window is somewhat complicated to calculate. If the pixmap is larger than the clip window, the clip window acts as a view into the pixmap, so only a portion of the pixmap can be seen. If the pixmap is smaller than the clip window, the entire pixmap can be seen, so the pixmap should be centered in the middle of the viewable area. The application controls this behavior using a number of global variables.

The `view_width` and `view_height` variables represent the dimensions of the ScrolledWindow, which are also the dimensions of the DrawingArea window. The area specified by these values is the area of the pixmap that is going to be copied into the window. The `pix_hoffset` and `pix_voffset` variables represent the horizontal and vertical offsets into the pixmap when it is rendered into the DrawingArea. If the pixmap is larger than the clip window, these values are calculated in the `scrolled()` callback routine when the user performs a scrolling action. If the pixmap is smaller than the DrawingArea, these values are set to `0` because the origin of the pixmap is always visible. The `sw_hoffset` and `sw_voffset` variables are used when the pixmap is smaller than the DrawingArea. The values indicate the offsets into the DrawingArea where the entire pixmap is rendered so that it appears centered in the viewport.

The redraw() routine depends on these variables being set. In order to maintain the values, the application monitors the sizeof the DrawingArea. When a ConfigureNotify event occurs on the DrawingArea, the expose_resize() callback routine is invoked. The routine gets the new dimensions of the DrawingArea so that it can update the six variables mentioned above. Normally, we can get the new dimensions directly from the event field of the callback structure. However, the DrawingArea widget invokes the XmNresizeCallback from within the Resize() method, instead of from an action routine, so the callback does not have an XEvent structure associated with it.* Since the event field of the callback structure is set to NULL, we have to get the window's size in another way. We use XtVaGetValues(), as shown in the following code fragment:

```
XtVaGetValues (drawing_a, XmNwidth, &view_width, XmNheight, &view_height,
               NULL);
```

Once we have the dimensions, we need to recalculate the value of the other four variables. Since our variables represent pixel values, while the ScrollBar resources that we need to set use an abstract unit size, we must convert between the two types of values using the cell_width and cell_height values. The variables new_width and new_height represent the new viewport width and height in ScrollBar units.

If the new viewport height exceeds the number of rows in the pixmap, we know that the height of the viewport exceeds the height of the pixmap. In this case, the value for sw_voffset is calculated to determine the offset that causes the pixmap to be centered vertically in the viewport. Since the viewport needs to be completely redisplayed, we set the local variable do_clear to True. We use this variable instead of calling XClearWindow() directly because we may have to do it again later when we calculate the values for the horizontal ScrollBar. The value for new_height is going to be used to set the XmNsliderSize for the vertical ScrollBar, so we make sure that it does not exceed its XmNmaximum value.

On the other hand, if the new viewport height does not exceed the total number of rows, we know that the pixmap is larger than the viewport vertically. The pixmap is not going to be centered in the DrawingArea, so sw_voffset is set to 0. pix_voffset is set to the minimum of its existing value and the difference between the total number of rows and the new height of the viewport. If the viewport used to be bigger than the pixmap, but is now smaller, we need to clear the window and do a complete redisplay. If the pixmap was bigger than the viewport and it still is, then we do not need to clear the window because the current view is still accurate. The different between these two cases is subtle and it is the sort of thing that you catch only when you test your program thoroughly.

* All widget internals have methods that are invoked automatically by the X Toolkit Intrinsics and are not associated with the translation tables normally used to handle events. Resize() is one such method. See Volume 4, *X Toolkit Intrinsics Programming Manual*, for more information.

After the calculations are performed, the application sets the XmNsliderSize, XmNvalue, and XmNpageIncrement resources for the vertical ScrollBar. The exact same calculations are done for the horizontal dimension and the same resources are set on the horizontal ScrollBar. With these resources set, scrolling continues to function properly when the DrawingArea is resized. When redraw() is called, it uses the global variables to copy the relevant portion of the full pixmap directly into the DrawingArea. If the program is running on a color screen, the routine uses XCopyPlane() because the DrawingArea cannot create a 1-bit deep window on a color screen. (Motif widgets always create windows of the same depth as the screen on which they reside.) If the application is run on a monochrome screen, the routine uses XCopyArea(). We determine the depth of the screen using DefaultDepthOfScreen().

Incidentally, while we did not use it, XmScrollBarSetValues() could have been used to set the resources on the ScrollBars. This function takes the following form:

```
void XmScrollBarSetValues (Widget widget, int value, int slider_size,
                           int increment, int page_increment, Boolean notify)
```

The *notify* parameter specifies whether you want the XmNvalueChangedCallback for the ScrollBar to be invoked. Using this interface is probably slightly faster than using the XtVaSetValues() method, but only by a small margin, so we chose to maintain consistency with our own style. The companion function for XmScrollBarSetValues() is XmScrollBarGetValues(). This function retrieves the values from the ScrollBar widget and takes the following form:

```
void XmScrollBarGetValues (Widget widget;, int *value, int *slider_size,
                           int *increment, int *page_increment)
```

Before closing this section, let's examine what the Text and List widgets do and compare it with what we have done in Example 10-4. We stated earlier that while we mimic much of what these widgets do internally, the implementation is quite different. The major difference is that we are fortunate enough to have all of the bitmaps loaded into a large, statically-sized pixmap that we can render at will using the redraw() function. This function is clearly a convenience, since it simply calls XCopyArea() or XCopyPlane() to copy the pixmap into the DrawingArea using pre-calculated internal variables. The Text and List widgets do not have this luxury; they must redraw their respective data directly into the work windows each time they need to redisplay.

If we were to implement the bitmap viewer using this technique, we would have to move the functionality of the main for loop in main() into redraw() and calculate the location of each individual bitmap in the DrawingArea. This process is quite painstaking and very error-prone. If you do not take into account multiple exposures, exposure regions, and other low-level Xlib functionality, you might run into X performance issues. We didn't even take these issues into account in our program. For example, our redraw() routine completely repaints the entire window for every Expose event. Strictly speaking, repainting is inefficient and may not perform adequately for all applications, especially graphic-

intensive ones. To avoid this problem, you could come up with a generic set of routines to handle exposures, so that of all your applications could use the same methodology, but that's the point of a toolkit.

Let's take another look at the PG&E scenario that we discussed at the beginning of the chapter. As you recall, the problem with that particular situation was that the database could retrieve 20% of the county (the work window), but the graphic resolution was such that only 10% of it could be displayed at one time (the viewport). The fundamental problem with the application-defined scrolling mode is that the work window cannot be a different size from the viewport. However, we can work around this problem by complying with the restriction that the work window and viewport are the same size, but we can use the enlarged pixmap idea from Example 10-4 to accomplish the task. Each database query can be converted and rendered into a sufficiently large pixmap, which can then be rendered into the work window as necessary. If the scrolling is small enough, another part of the pixmap can be rendered into the work window, instead of performing a completely new database lookup.

Working With Keyboard Traversal in ScrolledWindows

As we described in Chapter 8, *Manager Widgets*, manager widgets play a significant role in handling keyboard traversal mechanisms. As a manager, the ScrolledWindow supports keyboard traversal. However, one significant difference is that the widgets in a ScrolledWindow may not be visible at all times. There is a callback for the ScrolledWindow that supports keyboard traversal in a ScrolledWindow. The `XmNtraverseObscuredCallback` is invoked when the user attempts to traverse to a widget that is not visible in a ScrolledWindow. If there is no routine specified for this callback, an obscured widget cannot be traversed to. An application can use the callback to cause the ScrolledWindow to make a widget visible, so that it can receive the input focus. Example 10-5 shows the use of the `XmNtraverseObscuredCallback`.*

Example 10-5 The traversal.c program

```
/* traversal.c -- demonstrate keyboard traversal in a ScrolledWindow
** using the XmNtraverseObscuredCallback.
*/
#include <Xm/PushB.h>
#include <Xm/ToggleB.h>
#include <Xm/ScrolledW.h>
#include <Xm/RowColumn.h>

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

* XtVaAppInitialize() is considered deprecated in X11R6.

```
    Widget         toplevel, sw, rc, child;
    XtAppContext   app;
    void           traverse(Widget, XtPointer, XtPointer);
    int            i;
    char           name[10];
    Arg            args[4];

    XtSetLanguageProc (NULL, NULL, NULL);
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv, NULL,
                                    sessionShellWidgetClass, NULL);
    i = 0;
    XtSetArg (args[i], XmNscrollingPolicy, XmAUTOMATIC); i++;
    sw = XmCreateScrolledWindow (toplevel, "scrolled_w", args, i);
    XtAddCallback (sw, XmNtraverseObscuredCallback, traverse, NULL);

    /* RowColumn is the work window for the widget */
    i = 0;
    XtSetArg (args[i], XmNorientation, XmHORIZONTAL); i++;
    XtSetArg (args[i], XmNpacking, XmPACK_COLUMN); i++;
    XtSetArg (args[i], XmNnumColumns, 10); i++;
    rc = XmCreateRowColumn (sw, "rc", args, i);

    for (i = 0; i < 10; i++) {
        sprintf (name, "Toggle %d", i);
        child = XmCreateToggleButton (rc, name, NULL, 0);
        XtManageChild (child);
        sprintf (name, "Button %d", i);
        child = XmCreatePushButton (rc, name, NULL, 0);
        XtManageChild (child);
    }

    XtManageChild (rc);
    XtManageChild (sw);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

void traverse (Widget widget, XtPointer client_data, XtPointer call_data)
{
    XmTraverseObscuredCallbackStruct *cbs =
                        (XmTraverseObscuredCallbackStruct *) call_data;
    XmScrollVisible (widget, cbs->traversal_destination, 10, 10);
}
```

This program creates a bunch of ToggleButtons and PushButtons in a RowColumn widget that is the work area for a ScrolledWindow. The `traverse()` routine is installed as the `XmNtraverseObscuredCallback`. The `call_data` parameter to the callback is an `XmTraverseObscuredCallbackStruct`, which is defined as follows:

```
typedef struct {
int                     reason;
XEvent                  *event;
Widget                  traversal_destination;
XmTraversalDirection    direction;
```

```
} XmTraverseObscuredCallbackStruct;
```

The `reason` field contains the value `XmCR_OBSCURED_TRAVERSAL`. The `traversal_destination` field specifies the widget that is to receive the input focus and `direction` specifies the direction of traversal. The `traverse()` routine calls `XmScrollVisible()` to make the `traversal_destination` widget visible. This routine takes the following form:

```
void XmScrollVisible (Widget scrollw, Widget widget, Dimension hor_margin,
                      Dimension ver_margin)
```

The *`scrollw`* parameter specifies the ScrolledWindow widget, while the *`widget`* parameter specifies the widget that is to be made visible. The *`hor_margin`* and *`ver_margin`* arguments indicate the margins that are used if the viewport of the ScrolledWindow needs to be adjusted to make the widget visible. If you run the program in Example 10-5, you can use the arrow keys to traverse all of the widgets in the ScrolledWindow.

Summary

The ScrolledWindow provides a convenient interface for displaying large amounts of data when you have limited screen real estate. For most situations, the automatic scrolling mode is all that you really need. In this mode, a ScrolledWindow requires very little care and feeding. By installing callback routines on the ScrollBars, you can even monitor the scrolling actions. However, there are some drawbacks to the automatic scrolling mode: all of the data must be rendered into the work window widget and scrolling occurs in single-pixel increments. If the size of the work window that you need is prohibitively large or if you need to support scrolling in other than single-pixel increments, you must use application-defined scrolling.

As demonstrated in Example 10-4, there is quite a bit of work involved in supporting real application-defined scrolling because of the different states in the relationship between the size of the work window and the underlying data. You must be able to support not only the underlying data, but also the way it is rendered into the work window, the ScrollBars, and all of the auxiliary variables required for the scrolling calculations. And that work is just to support the scrolling functionality. When you introduce the complexity of a real application, there is a greater chance of a poor design model. The *xshowbitmap.c* program in the Appendix A, *Additional Example Programs*, is fundamentally the same program as *app_scroll.c*, but it has been enhanced into more of a real-world program.

Exercises

The following exercises focus on the concepts and methods described in this chapter.

1. In Chapter 11, the program *color_draw.c* used a ScrolledWindow to support a DrawingArea widget that allows the user to draw different colored lines. Although this program uses an automatic ScrolledWindow, the work window is constantly updated as new lines are drawn. However, the lines are actually drawn into a background pixmap, rather than into the drawing area. The pixmap is copied into the DrawingArea dynamically, which gives the illusion that the user is drawing directly into it. This method of indirection can be used to provide a way for the user to have two different views into the same pixmap. Write a program that uses two automatic ScrolledWindows and two DrawingArea widgets to draw into a single pixmap.
2. The *getusers.c* example uses an automatic ScrolledWindow to display a manager widget that contains many widgets and gadgets. Modify the program to use application-defined scrolling, so that the scrolling increment for the vertical ScrollBar is the sizeof the height of one of the Forms. The Forms all have the same height.

In this chapter:

- *Creating a DrawingArea Widget*
- *Using DrawingArea Callback Functions*
- *Using Translations on a DrawingArea*
- *Using Color in a DrawingArea*
- *Summary*
- *Exercises*

11

The DrawingArea Widget

This chapter describes the Motif DrawingArea widget, which provides a canvas for interactive drawing. The chapter does not try to teach Xlib drawing, but rather it highlights, with numerous code examples, the difficulties that may be encountered when working with this widget. The chapter assumes some knowledge of Xlib. See *Volume 1, Xlib Programming Manual*, for additional information.

The DrawingArea widget provides a blank canvas for interactive drawing using basic Xlib drawing primitives. The widget does no drawing of its own, nor does it define or support any Motif user-interface design style. Since it is subclassed from the Manager widget class, the DrawingArea widget may also contain other widgets as children, although there is no regimented layout policy. In short, the DrawingArea is a free-form widget that you can use for interactive drawing or object placement when conventional user-interface rules do not apply.

The most intuitive use of the DrawingArea is for a drawing or painting program. Here, the user can interactively draw geometric objects and paint arbitrary colors. Another application might used a DrawingArea widget to display a map of a country with dynamically-drawn line segments representing the flight paths taken by aeroplanes. The actual aeroplanes could be represented by PushButton widgets displaying pixmaps. Each aeroplane icon moves dynamically along its flight path unless the user grabs and moves it interactively in order to change the flight path. Both of these examples demonstrate how certain applications require visual or interactive interfaces that go beyond the scope of the structured interface provided by Motif.

In order to support the widest range of uses for the DrawingArea widget, the toolkit provides callback resources for exposure, configure (resize), and input (button and key presses) events. Each of these callbacks allows you to install very simple drawing routines without doing substantial event-handling of your own. Unfortunately, this level of event-handling support is usually insufficient for most robust applications. As a result, most applications install direct event handlers or action routines to manage user input. The free-form nature of the DrawingArea makes it one of the few Motif widgets where you can do

handle events at this level without risking non-compliance with the *Motif Style Guide*. (Most Motif widgets either do not allow programmer-installed translations or (silently) accept only a few override translations for fear that you might inadvertently interfere with Motif GUI specifications.)

If you are using a DrawingArea as a manager widget, there are two important things to keep in mind: translation tables and widget layout management. As a Manager widget subclass, the DrawingArea inherits certain translation and action tables that pass events to gadget children and handle tab group traversal. Because of the inherited translations, you must be careful about application-specific translations that you may introduce into particular instances of the DrawingArea. If you are planning to use the DrawingArea to contain children and to have those children follow the standard Motif keyboard traversal motions, you must be careful not to override the existing translations.

However, if you need a manager widget in the conventional sense, you should probably choose something other than a DrawingArea widget, since the widget has no geometry management policy of its own. The DrawingArea should probably only be used to manage children when no structured widget layout policy is needed, as in the airline application. In this situation, the widget assumes the dual responsibility of managing children and allowing for application-defined interaction. As a result, there are going to be some complexities and inconveniences with event handling, since the application is trying to take advantage of both aspects of the widget simultaneously.

Creating a DrawingArea Widget

Applications that wish to create DrawingArea widgets must include the standard header file *<Xm/DrawingA.h>*. To create a DrawingArea widget, you can use the following calls:

```
Widget drawing_a = XmCreateDrawingArea ( parent, "name", resource-value-array,
                                         resource-value-count);
Widget drawing_a = XtCreateWidget ( “name”, xmDrawingAreaWidgetClass, parent,
                                    resource-value-list, NULL);
```

The parent of a DrawingArea must be either some type of Shell or a manager widget. It is quite common to find a DrawingArea widget as a child of a ScrolledWindow or a MainWindow, since drawing surfaces tend to be quite large, or at least dynamic in their growth potential.

If the DrawingArea widget is to have children, you might want to follow the guidelines set forth in Chapter 8, *Manager Widgets*, about creating the widget in an unmanaged state. The widget can be managed with a call to `XtManageChild()` after its children have been created. Here we are not going to use the widget as a traditional manager and there is not going to be a great deal of parent-child interaction involving geometry management.

Using DrawingArea Callback Functions

The DrawingArea widget provides virtually no visual resources and very few functional ones. The most important resources are those that allow you to provide callback functions for handling expose, resize, and input events. The DrawingArea is typically input-intensive and, unlike most of the other Motif widgets, requires the application to provide all of the necessary redrawing.

The callback routine for the `XmNexposeCallback` is invoked whenever an `Expose` event is generated for the widget. In this callback function, an application must repaint all or part of the contents of the DrawingArea widget. If an application does not redraw the contents of the widget, it appears empty, as the widget is cleared automatically. Similarly, the `XmNresizeCallback` is called whenever a `ConfigureNotify` event occurs as a result of the DrawingArea being resized. The generalized `XmNinputCallback` is invoked as a result of every keyboard and button event except button motion events.

As discussed in Chapter 2, *The Motif Programming Model*, callback routines are invoked by internal action routines that are an integral part of all Motif widgets. Translation tables are used to specify X event sequences that invoke the action routines. Action functions typically invoke the appropriate application callback functions associated with the widget's resources.

Most Motif widgets do not allow the application to override or replace their default translations; the input model that allows the application to conform to the Motif specifications is not to be overridden by the application. However, because of the free-form nature of the DrawingArea widget, you are free to override or replace the default translation tables used for event-handling and notification without non-compliant behavior. If you install your own translation tables, you can have your action routines invoke callback routines as is done by the existing DrawingArea actions, or you can have your action functions do the drawing directly. For even tighter control over event-handling, you can install event handlers at the X Toolkit Intrinsics level.

There are a number of techniques available for doing event management and we only demonstrate a few of them in this chapter. The technique you choose is a matter of personal preference and the intended extensibility of your application. Event handlers involve less overhead, but translations are user-configurable. Either approach provides more flexibility than using the default translation table and callback resources of the DrawingArea. See Volume 4, *X Toolkit Intrinsics Programming Manual*, for a detailed discussion of translation tables and action routines and how they are associated with callback functions.

Handling Input Events

Since the callback approach to event handling is the simplest, we'll begin by discussing that approach. Example 11-1 shows an extremely simple drawing program that associates a line

drawing function with the XmNinputCallback resource. Pressing any of the pointer buttons marks the starting point of a line; releasing the button marks the end point. You can only draw straight lines. Even though the default translation table for the DrawingArea widget selects key events and these events are passed to the callback function, the callback function itself ignores them and thus key events have no effect.

To demonstrate the complications inherent in using the DrawingArea widget as a manager, the program also displays a PushButton gadget that clears the window. A single callback function, drawing_area_callback(), uses both the reason and the event fields of the XmDrawingAreaCallbackStruct to determine whether to draw a line or to clear the window.

This simple application draws directly into the DrawingArea widget; the contents of its window is not saved anywhere. The program does not support redrawing, since its purpose is strictly to demonstrate the way input handling can be managed using the XmNinputCallback. If the window is exposed due to the movement of other windows, the contents of the window is not redrawn. A more realistic drawing application would need code to handle both expose and resize actions. The current application simply clears the window on resize to further illustrate that the DrawingArea does not retain what is in its window.*

Example 11-1 The drawing.c program

```
/* drawing.c -- extremely simple drawing program that introduces
** the DrawingArea widget. This widget provides a window for
** drawing and some callbacks for getting input and other miscellaneous
** events. It's also a manager, so it can have children.
** There is no geometry management, though.
*/
#include <Xm/DrawingA.h>
#include <Xm/PushBG.h>
#include <Xm/RowColumn.h>

main (int argc, char *argv[])
{
    Widget         toplevel, drawing_a, pb;
    XtAppContext   app;
    XGCValues      gcv;
    GC             gc;
    void           drawing_area_callback(Widget, XtPointer, XtPointer);

    XtSetLanguageProc (NULL, NULL, NULL);
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv, NULL,
                sessionShellWidgetClass, XmNwidth, 400, XmNheight, 300, NULL);
    /* Create a DrawingArea widget. */
    drawing_a = XmCreateDrawingArea (toplevel, "drawing_a", NULL, 0);
```

* XtVaAppInitialize() is considered deprecated in X11R6.

```
    /* add callback for all mouse and keyboard input events */
    XtAddCallback (drawing_a, XmNinputCallback, drawing_area_callback, NULL);

    /* Since we're going to be drawing, we will be using Xlib routines
    ** and therefore need a graphics context. Create a GC and attach
    ** to the DrawingArea's XmNuserData to avoid having to make global
    ** variable. (Avoiding globals is a good design principle to follow.)
    */
    gcv.foreground = BlackPixelOfScreen (XtScreen (drawing_a));
    gc = XCreateGC (XtDisplay (drawing_a),
    RootWindowOfScreen (XtScreen (drawing_a)), GCForeground, &gcv);
    XtVaSetValues (drawing_a, XmNuserData, gc, NULL);

    /* add a pushbutton the user can use to clear the canvas */
    pb = XmCreatePushButtonGadget (drawing_a, "Clear",NULL, 0);

    /* if activated, call same callback as XmNinputCallback. */
    XtAddCallback (pb, XmNactivateCallback, drawing_area_callback, NULL);

    XtManageChild (pb);
    XtManageChild (drawing_a);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

/* Callback routine for DrawingArea's input callbacks and the
** PushButton's activate callback. Determine which it is by
** testing the cbs->reason field.
*/
void drawing_area_callback (Widget widget, XtPointer client_data,
                            XtPointer call_data)
{
    static Position x, y;
    XmDrawingAreaCallbackStruct *cbs =
    (XmDrawingAreaCallbackStruct *) call_data;
    XEvent *event = cbs->event;

    if (cbs->reason == XmCR_INPUT) {
        /* activated by DrawingArea input event -- draw lines.
        ** Button Down events anchor the initial point and Button
        ** Up draws from the anchor point to the button-up point.
        */
        if (event->xany.type == ButtonPress) {
            /* anchor initial point (i.e., save its value) */
            x = event->xbutton.x;
            y = event->xbutton.y;
        }
        else if (event->xany.type == ButtonRelease) {
            /* draw full line; get GC and use in XDrawLine() */
            GC gc;

            XtVaGetValues (widget, XmNuserData, &gc, NULL);
            XDrawLine (event->xany.display, cbs->window, gc, x, y,
                            event->xbutton.x, event->xbutton.y);
```

```
                x = event->xbutton.x;
                y = event->xbutton.y;
            }
        }

        if (cbs->reason == XmCR_ACTIVATE)
            /* activated by pushbutton -- clear parent's window */
            XClearWindow (event->xany.display, XtWindow (XtParent (widget)));
    }
```

The output of the program is shown in Figure 11-1.

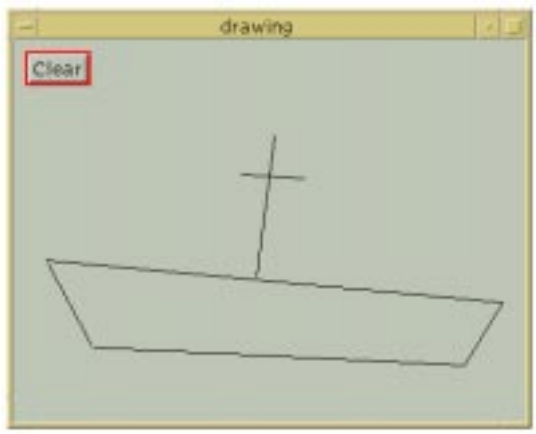


Figure 11-1 Output of the drawing program

The callback routine that is used for the `XmNinputCallback` takes the form of a standard callback routine. The DrawingArea provides a `XmDrawingAreaCallbackStruct` for all of its callbacks. This structure is defined as follows:

```
typedef struct {
    int        reason;
    XEvent     *event;
    Window     window;
} XmDrawingAreaCallbackStruct;
```

The `reason` field identifies the type of occurrence that caused the callback to be invoked. For the `XmNinputCallback`, the value is `XmCR_INPUT`. The `event` field of the callback structure describes the event that caused the callback to be invoked. The `window` field is the window associated with the DrawingArea widget - this is the same value returned by calling `XtWindow()` on the widget.

Since the event itself is passed in as part of the callback structure, we can look at the `type` field of the event for more information than is provided by the callback reason alone. (See Volume 1, *Xlib Programming Manual*, for a detailed description of `XEvent` structures and how to use them.) In fact, since there are many possible events that can be associated with the reason `XmCR_INPUT`, you have to look at the event structure if you need any detail

about what actuallyhappened. Table 11-1 shows the possible event types for each of the DrawingArea callbacks.

Table 11-1Callback Reasons and Event Types

Callback	Reason	Event Type(s)
XmNexposeCallback	XmCR_EXPOSE	Expose
XmNresizeCallback	XmCR_RESIZE	ConfigureNotify
XmNinputCallback	XmCR_INPUT	ButtonPress, ButtonRelease, KeyPress, KeyRelease

A common convention we've included in this program is the double use of the `drawing_area_callback()` function. This technique is known as *function overloading*, since the same function is used by more than one source. We are using the routine as the input callback for the DrawingArea widget, as well as the activate callback for the PushButton gadget. Whenever the PushButton is activated, the callback function is invoked and passed an `XmPushButtonCallbackStruct` with the `reason` field set to `XmCR_ACTIVATE`.

It is beyond the scope of this book to discuss at length or even introduce the use of Xlib; for that, see Volume 1, *Xlib Programming Manual*. However, there are a couple of details concerning the use of Xlib functions that are noteworthy. For efficiency in use of the X protocol, Xlib drawing calls typically do not carry a lot of information about the drawing to be done. Instead, drawing characteristics such as the foreground and background colors, fill style, line weight, and so on, are defined in a graphics context (GC), which is cached in the X server. Any drawing function that wishes to use a particular GC must include the handle returned by a GC creation call.

If many different routines are going to use the same GC, the programmer should try to make the handle to it generally available. The natural tendency is to declare the GC as a global variable. However, as a program gets large, it is easy to get carried away with the use of global variables. As a result, programs tend to get over complicated and decentralized. To avoid this problem, you can use the `XmNuserData` resource (inherited from the Manager widget class) as a temporary holding area for arbitrary pointers and values. Since this program is small, it may not be worth the overhead of a call to `XtGetValues()` to avoid a global variable. It is up to you if you want to use the `XmNuserData` resource; this particular example just shows one way of avoiding global variables.

If you play with the program a little, you will soon find that you can draw right through the PushButton gadget in the DrawingArea. Because gadgets do not have windows, the DrawingArea widget indiscriminately allows you to draw through any gadget children it may be managing. Similarly, activating the PushButton clears the DrawingArea window, but it does not repaint the PushButton. None of the manager widgets, including the DrawingArea, check if the user (or the application) is overwriting or erasing gadgets. Changing the PushButton from a gadget to a widget solves the immediate problem.

However, it is generally not a good idea to use a DrawingArea widget as both a drawing canvas and as a place to have user-interface elements such as PushButtons.

For conventional geometry management involving DrawingArea widgets, you have two choices. You can write your own geometry management routine (as demonstrated for BulletinBoard widgets in Chapter 8, *Manager Widgets* or you can place the DrawingArea inside another manager that does more intelligent geometry management. The nice part about this alternative is that the other manager widgets are no more or less intelligent about graphics and repainting than the DrawingArea widget. They don't provide a callback for `Expose` events, but you can always add translations for those events, if you need them.

Redrawing a DrawingArea

In Example 11-1, when an `Expose` event or a `Resize` event occurs, the drawing is not retained and as a result the DrawingArea is always cleared. This problem was intentional for the first example because we wanted to focus on the use of the input callback routine. However, when you use the DrawingArea widget, you must always be prepared to repaint whatever is supposed to be displayed in the widget at any time.

As you may already know, most X servers support a feature called *backing store*, which saves the contents of windows, even when they are obscured by other windows, and repaints them when they are exposed. When backing store is enabled and there is enough memory available for the server, X will repaint all damaged windows without ever notifying the application that anything happened. However, you should never rely on this behavior, since you never know if the X server supports backing store, or if it has enough memory to save the contents of your windows. All applications are ultimately responsible for redrawing their windows' contents whenever necessary.

For a painting application like that in Example 11-1, the easiest way to make sure that a window can be repainted whenever necessary is to draw both into the window and into an off screen pixmap. The contents of the pixmap can be copied back into the window as needed. Example 11-2 demonstrates such a program. The off screen pixmap is copied back to the window with `XCopyArea()` to redisplay the drawing when the `XmNexposeCallback` is called.*

Example 11-2 The draw2.c program

```
/* draw2.c -- extremely simple drawing program that demonstrates
** how to draw into an off screen pixmap in order to retain the
** contents of the DrawingArea widget. This allows us to redisplay
** the widget if it needs repainting (expose events).
*/
#include <Xm/DrawingA.h>
#include <Xm/PushBG.h>
```

* `XtVaAppInitialize`() is considered deprecated in X11R6.

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

#define WIDTH 400    /* arbitrary width and height values */
#define HEIGHT 300
Pixmap pixmap; /* used to redraw the DrawingArea */

main (int argc, char *argv[])
{
    Widget         toplevel, drawing_a, pb;
    XtAppContext   app;
    GC             gc;
    void           drawing_area_callback(Widget, XtPointer, XtPointer);

    XtSetLanguageProc (NULL, NULL, NULL);
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv, NULL,
        sessionShellWidgetClass, XmNwidth, WIDTH, XmNheight, HEIGHT, NULL);

    /* Create a DrawingArea widget. */
    drawing_a = XmCreateDrawingArea (toplevel, "drawing_a", NULL, 0);

    /* add callback for all mouse and keyboard input events */
    XtAddCallback (drawing_a, XmNinputCallback, drawing_area_callback, NULL);
    XtAddCallback (drawing_a, XmNexposeCallback, drawing_area_callback, NULL);
    gc = XCreateGC (XtDisplay (drawing_a),
                    RootWindowOfScreen (XtScreen (drawing_a)), 0, NULL);
    XtVaSetValues (drawing_a, XmNuserData, gc, NULL);
    XSetForeground (XtDisplay (drawing_a), gc,
                    WhitePixelOfScreen (XtScreen (drawing_a)));

    /* create a pixmap the same size as the drawing area. */
    pixmap = XCreatePixmap (XtDisplay (drawing_a),
                            RootWindowOfScreen (XtScreen (drawing_a)),
                            WIDTH, HEIGHT,
                            DefaultDepthOfScreen (XtScreen (drawing_a)));

    /* clear pixmap with white */
    XFillRectangle (XtDisplay (drawing_a), pixmap, gc, 0, 0, WIDTH, HEIGHT);

    /* drawing is now drawn into with "black"; change the gc for future */
    XSetForeground (XtDisplay (drawing_a), gc,
                    BlackPixelOfScreen (XtScreen (drawing_a)));

    /* add a pushbutton the user can use to clear the canvas */
    pb = XmCreatePushButtonGadget (drawing_a, "Clear", NULL, 0);
    XtManageChild (pb);

    /* if activated, call same callback as XmNinputCallback. */
    XtAddCallback (pb, XmNactivateCallback, drawing_area_callback, NULL);
    XtManageChild (drawing_a);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}
```

```
/* Callback routine for DrawingArea's input and expose callbacks
** as well as the PushButton's activate callback. Determine which
** it is by testing the cbs->reason field.
*/
void drawing_area_callback (Widget widget, XtPointer client_data,
                            XtPointer call_data)
{
    static Position x, y;
    XmDrawingAreaCallbackStruct *cbs =
                            (XmDrawingAreaCallbackStruct *) call_data;
    XEvent *event = cbs->event;
    Display *dpy = event->xany.display;

    if (cbs->reason == XmCR_INPUT) {
        /* activated by DrawingArea input event -- draw lines.
        ** Button Down events anchor the initial point and Button
        ** Up draws from the anchor point to the button-up point.
        */
    if (event->xany.type == ButtonPress) {
        /* anchor initial point (i.e., save its value) */
        x = event->xbutton.x;
        y = event->xbutton.y;
    } else if (event->xany.type == ButtonRelease) {
        /* draw full line; get GC and use in XDrawLine() */
        GC gc;

        XtVaGetValues (widget, XmNuserData, &gc, NULL);
        XDrawLine (dpy, cbs->window, gc, x, y,
                   event->xbutton.x, event->xbutton.y);

        /* draw into the pixmap as well for redrawing later */
        XDrawLine (dpy, pixmap, gc, x, y,
                   event->xbutton.x, event->xbutton.y);
        x = event->xbutton.x;
        y = event->xbutton.y;
        }
    }

    if (cbs->reason == XmCR_EXPOSE || cbs->reason == XmCR_ACTIVATE) {
        GC gc;
        if (cbs->reason == XmCR_ACTIVATE) /* Clear button pushed */
            widget = XtParent (widget);
        /* get the DrawingArea widget */
        XtVaGetValues (widget, XmNuserData, &gc, NULL);

        if (cbs->reason == XmCR_ACTIVATE) {/* Clear button pushed */
            /* to clear a pixmap, reverse foreground and background */
            XSetForeground (dpy, gc, WhitePixelOfScreen (XtScreen (widget)));
            /* ...and fill rectangle the size of the pixmap */
            XFillRectangle (dpy, pixmap, gc, 0, 0, WIDTH, HEIGHT);
            /* don't forget to reset */
            XSetForeground (dpy, gc, BlackPixelOfScreen (XtScreen (widget)));
        }
        /* Note: we don't have to use WIDTH and HEIGHT--we could pull the
```

```
            ** exposed area out of the event structure, but only if the reason
            ** was XmCR_EXPOSE... make it simple for the demo; optimize as needed.
            */
            XCopyArea (dpy, pixmap, event->xany.window, gc, 0, 0,
                       WIDTH, HEIGHT, 0, 0);
        }
    }
```

A frequent problem encountered in using the DrawingArea widget is the need to redraw after every `Resize` event. When you enlarge the DrawingArea window, an `Expose` event is automatically generated since more of the window becomes exposed. But, if you shrink the window, no `Expose` event is generated since no new part of the window is being exposed.

The reason why no `Expose` event is generated when you shrink a DrawingArea widget is deep inside Xlib. The bit gravity of a window indicates where new bits are placed automatically by X when a window is resized. If you resize a window larger, then the data in the window remains in the top-left corner and the application gets a `Resize` event and an `Expose` event. The `Expose` event just identifies the newly exposed area, not the entire window. If you make the window smaller, all of the data in the window gets pushed to the top left; there is no newly exposed area, so there is no `Expose` event.

The solution is to make the window forget about bit gravity, so every `Resize` event causes all of the bits to be cleared. As a result, the `Expose` event identifies the entire window as being exposed, instead of just the newly exposed region. This technique has the side effect of generating an `Expose` event even when the window is resized smaller.

There is no routine to set the bit gravity of a window individually. It can be set only with `XChangeWindowAttributes()`, as in the following code fragment:

```
XSetWindowAttributes attrs;
attrs.bit_gravity = ForgetGravity;
XChangeWindowAttributes ( XtDisplay (drawing_area),
                          XtWindow (drawing_area), CWBitGravity, &attrs);
```

Once you do this, the DrawingArea widget gets `Expose` events when you resize it to be smaller.

Using Translations on a DrawingArea

As mentioned earlier, it is generally permissible to override or replace the default translation table of the DrawingArea widget with new translations. The only potential problem is if you plan to use the DrawingArea as a manager for other widgets and you expect it to follow the keyboard traversal mechanisms described by the *Motif Style Guide*. In fact, handling keyboard traversal is pretty much all that the default translations for the DrawingArea do. For example, the following is a subset of the default translations for the DrawingArea widget:*

```
<Key>osfSelect:            DrawingAreaInput() ManagerGadgetSelect()
<Key>osfActivate:          DrawingAreaInput() ManagerParentActivate()
<Key>osfHelp:              DrawingAreaInput() ManagerGadgetHelp()
<KeyDown>:                 DrawingAreaInput() ManagerGadgetKeyInput()
<KeyUp>:                   DrawingAreaInput()
<BtnMotion>:               ManagerGadgetButtonMotion()
<Btn1Down>:                DrawingAreaInput() ManagerGadgetArm()
<Btn1Down>,<Btn1Up>:       DrawingAreaInput() ManagerGadgetActivate()
```

These translations show that the manager widget part of the DrawingArea is responsible for tracking events for its gadget children. It is not necessary to support these translations if you are not going to use the DrawingArea to manage children. Most user-generated events also invoke `DrawingAreaInput()`, which does not do any drawing, but simply invokes the `XmNinputCallback`.

As you can see, the `BtnMotion` translation is not passed to `DrawingAreaInput()`, which means that the `XmNinputCallback` is not called for pointer motion events. When it comes to more complex drawing than that done in Example 11-2, this omission is a serious deficiency. To support rubber banding or free-hand drawing techniques, which require pointer motion events, you must install either an event handler or a translation entry to handle motion events.

The simplest approach would be to replace the translation table entry for `<BtnMotion>` events. However, this is not possible, due to a bug in the X Toolkit Intrinsics. The correct thing to do is the following:

```
String translations =
            "<Btn1Motion>: DrawingAreaInput() ManagerGadgetButtonMotion()";
...
drawing_a = XmCreateDrawingArea (main_w, "drawing_a", NULL, 0);
XtOverrideTranslations (drawing_a, XtParseTranslationTable (translations));
XtAddCallback (drawing_a, XmNinputCallback, draw, NULL);
```

With this new translation, the `XmNinputCallback` function (`draw()`) would be notified of pointer motion while Button 1 is down.

`XtOverrideTranslations()` is the preferred method for installing a new translation into the DrawingArea widget because it is non destructive. The routine only replaces translations for which identical events are specified and leaves all other translations in place. However, this routine does not work in this case because there is already a translation for the Button 1 down-up sequence in the DrawingArea translation table. In the current implementation, once Button 1 goes down, the Xt event translator waits for the Button 1 up event to match the partially finished translation. Therefore, no Button 1 motion events can be caught.If we want to get pointer motion events while the button is down, we have to resort to other alternatives.

* This translation table lists only a subset of the current translations in the DrawingArea widget; there is no guarantee that the translations will remain the same in future revisions of the toolkit.

One such alternative is to replace the entire translation table, regardless of whether we are adding new entries or overriding existing ones. This is known as a destructive override because the existing translation table is thrown out. This action has the desired effect because the offending Button 1 translation is thrown out. However, we must then take steps to re-install any other default translations that are still required. To completely replace the existing translations, the XmNtranslations resource can be set as shown in the following code fragment:

```
String translations =
            "<Btn1Motion>: DrawingAreaInput() ManagerGadgetButtonMotion()";
XtTranslations parsed_translations = XtParseTranslationTable (translations);
...
XtSetArg (args[n], XmNtranslations, parsed_translations); n++;
drawing_a = XmCreateDrawingArea (main_w, "drawing_a", args, n);
XtAddCallback (drawing_a, XmNinputCallback, draw, NULL);
```

Once you go to the trouble of replacing the translation table, you may as well install your own action functions as well. Doing so allows you to do the drawing directly from the action functions, rather than using it as an intermediate function to call an application callback. This direct-drawing approach is demonstrated in Example 11-3. The program uses pointer motion to draw lines as the pointer is dragged with the button down, rather than when the button is pressed and released. You'll notice that we have used much the same design as in Example 11-2, but have moved some of the code into different callback routines and have placed the DrawingArea widget into a MainWindow widget for flexibility. None of these changes are required nor do they enhance performance in any way. They merely point out different ways of providing the same functionality.*

Example 11-3 The free_hand.c program

```
/* free_hand.c -- simple drawing program that does freehand
** drawing. We use translations to do all the event handling
** for us rather than using the drawing area's XmNinputCallback.
*/
#include <Xm/MainW.h>
#include <Xm/DrawingA.h>
#include <Xm/PushBG.h>
#include <Xm/RowColumn.h>

/* Global variables */
GC              gc;
Pixmap          pixmap;
Dimension       width, height;

main (int argc, char *argv[])
{
    Widget          toplevel, main_w, drawing_a, pb;
    XtAppContext    app;
```

* XtVaAppInitialize() is considered deprecated in X11R6.

```
XGCValues        gcv;
void             draw(Widget, XEvent *, String *, Cardinal *);
void             redraw(Widget, XtPointer, XtPointer);
void             clear_it(Widget, XtPointer, XtPointer);
XtActionsRec     actions;
Arg              args[10];
int              n;
String           translations = /* for the DrawingArea widget */
/* ManagerGadget* functions are necessary for DrawingArea widgets
** that steal away button events from normal translation tables.
*/
                "<Btn1Down>:  draw(down)   ManagerGadgetArm()\n\
                <Btn1Up>:     draw(up)     ManagerGadgetActivate()\n\
                <Btn1Motion>: draw(motion) ManagerGadgetButtonMotion()";

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

/* Create a MainWindow to contain the drawing area */
n = 0;
XtSetArg (args[n], XmNscrollingPolicy, XmAUTOMATIC); n++;
main_w = XmCreateMainWindow (toplevel, "main_w", args, n);

/* Add the "draw" action/function used by the translation table */
actions.string = "draw";
actions.proc = draw;
XtAppAddActions (app, &actions, 1);

/* Create a DrawingArea widget. Make it 5 inches wide by 6 inches tall.
** Don't let it resize so the Clear Button doesn't force a resize.
*/
n = 0;
XtSetArg (args[n], XmNtranslations,
                   XtParseTranslationTable (translations)); n++;
XtSetArg (args[n], XmNunitType, Xm1000TH_INCHES); n++;
XtSetArg (args[n], XmNwidth, 5000); n++; /* 5 inches */
XtSetArg (args[n], XmNheight, 6000); n++; /* 6 inches */
/* remain this a fixed size */
XtSetArg (args[n], XmNresizePolicy, XmRESIZE_NONE); n++;
drawing_a = XmCreateDrawingArea (main_w, "drawing_a", args, n);

/* When scrolled, the drawing area will get expose events */
XtAddCallback (drawing_a, XmNexposeCallback, redraw, NULL);

/* convert drawing area back to pixels to get its width and height */
XtVaSetValues (drawing_a, XmNunitType, XmPIXELS, NULL);
XtVaGetValues (drawing_a, XmNwidth, &width, XmNheight, &height, NULL);

/* create a pixmap the same size as the drawing area. */
pixmap = XCreatePixmap (XtDisplay (drawing_a),
                        RootWindowOfScreen (XtScreen (drawing_a)),
                        width, height,
                        DefaultDepthOfScreen (XtScreen (drawing_a)));
```

```
    /* Create a GC for drawing (callback). Used a lot -- make global */
    gcv.foreground = WhitePixelOfScreen (XtScreen (drawing_a));
    gc = XCreateGC (XtDisplay (drawing_a),
            RootWindowOfScreen (XtScreen (drawing_a)), GCForeground, &gcv);

    /* clear pixmap with white */
    XFillRectangle (XtDisplay (drawing_a), pixmap, gc, 0, 0, width, height);

    /* drawing is now drawn into with "black"; change the gc */
    XSetForeground (XtDisplay (drawing_a), gc,
                    BlackPixelOfScreen (XtScreen (drawing_a)));
    pb = XmCreatePushButtonGadget (drawing_a, "Clear",NULL, 0);
    XtManageChild (pb);

    /* Pushing the clear button calls clear_it() */
    XtAddCallback (pb, XmNactivateCallback, clear_it, (XtPointer) drawing_a);

    XtManageChild (drawing_a);
    XtManageChild (main_w);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

/* Action procedure to respond to any of the events from the
** translation table declared in main(). This function is called
** in response to Button1 Down, Up and Motion events. Basically,
** we're just doing a freehand draw -- not lines or anything.
*/
void draw (Widget widget, XEvent *event, String *args, Cardinal *num_args)
{
    static Position x, y;
    XButtonEvent *bevent = (XButtonEvent *) event;

    if (*num_args != 1)
        XtError ("Wrong number of args!");
    if (strcmp (args[0], "down")) {
        /* if it's not "down", it must either be "up" or "motion"
        ** draw full line from anchor point to new point.
        */
        XDrawLine (bevent->display, bevent->window, gc, x, y,
        bevent->x, bevent->y);
        XDrawLine (bevent->display, pixmap, gc, x, y,
        bevent->x, bevent->y);
    }
    /* freehand is really a bunch of line segments; save this point */
    x = bevent->x;
    y = bevent->y;
}

/* Clear the window by clearing the pixmap and calling XCopyArea() */
void clear_it (Widget pb, XtPointer client_data, XtPointer call_data)
{
    Widget drawing_a = (Widget) client_data;
```

```
    XmPushButtonCallbackStruct *cbs =
                            (XmPushButtonCallbackStruct *) call_data;

    /* clear pixmap with white */
    XSetForeground (XtDisplay (drawing_a), gc,
                    WhitePixelOfScreen (XtScreen (drawing_a)));
    XFillRectangle (XtDisplay (drawing_a), pixmap, gc, 0, 0, width, height);
    /* drawing is now done using black; change the gc */
    XSetForeground (XtDisplay (drawing_a), gc,
    BlackPixelOfScreen (XtScreen (drawing_a)));
    XCopyArea (cbs->event->xbutton.display, pixmap, XtWindow (drawing_a), gc,
                0, 0, width, height, 0, 0);
}

/* redraw is called whenever all or portions of the drawing area is
** exposed. This includes newly exposed portions of the widget resulting
** from the user's interaction with the scrollbars.
*/
void redraw (Widget drawing_a, XtPointer client_data, XtPointer call_data)
{
    XmDrawingAreaCallbackStruct *cbs =
                            (XmDrawingAreaCallbackStruct *) call_data;
    XCopyArea (cbs->event->xexpose.display, pixmap, cbs->window, gc, 0, 0,
                width, height, 0, 0);
}
```

The output of the program is shown in Figure 11-2.

Figure 11-2 Output of the free_hand program

In Example 11-3, the DrawingArea widget uses the following translation string:

```
String translations =
                "<Btn1Down>:   draw(down)   ManagerGadgetArm()\n\
                 <Btn1Up>:     draw(up)     ManagerGadgetActivate()\n\
                 <Btn1Motion>: draw(motion) ManagerGadgetButtonMotion()";
```

For each of the specified events, the translation describes two actions. The `draw()` action is our own function that actually draws into the DrawingArea. The `ManagerGadget` actions are standard DrawingArea actions (inherited from the Manager widget class) for passing events to a gadget child, as described earlier. We keep them in place because we

are still using the PushButton gadget. We are not keeping the routines for managing keyboard traversal, but simply those required to arm and activate the button.

The `draw()` action routine tests whether it has been called from a button up event, a button down event, or a motion event. Since the action function is passed the event that invoked it, we could simply test the type field of the event. However, this example gives us a chance to exercise the Xt feature that supports string arguments passed to action functions. Accordingly, the `draw()` function determines what action to take by examining its `args[0]` parameter, which contains the string passed as the single parameter in the translation table. For example, `draw(up)` passes the string `"up"` as the `args[0]` parameter in response to a <`Btn1Up`> event.

Lines are drawn for both `ButtonRelease` and `ButtonMotion` events, but not for `ButtonPress` events. A line is drawn from the last anchor point to the current location of the mouse. As the pointer moves from one point to the next, the anchor point is always one step behind, so a line segment is drawn from that location to the current location. The only time that a line segment is not drawn is on the initial button press (and any motion events that occur while the button is not down). The coordinate values are relative to the current location of the pointer within the DrawingArea widget, no matter how it is positioned in the MainWindow.

The `draw()` function draws into the window and also into a pixmap. The MainWindow widget is configured to have its `XmNscrollingPolicy` set to `XmAUTOMATIC`, so ScrollBars are automatically installed over the DrawingArea when it is larger than the MainWindow, which allows the user to view different parts of the canvas interactively. Scrolling actions cause the contents of the newly exposed portions of the canvas to be erased by default. Unless we provide a mechanism by which the DrawingArea can redraw itself, scrolling the DrawingArea loses previously drawn contents. To handle this problem, we employ the same principle we used in Example 11-2. We install a pixmap that is used by both the `draw()` and `redraw()` functions.

The `redraw()` routine is installed as the callback function for the `XmNexposeCallback`. The function merely uses `XCopyArea()` to copy the pixmap onto the window of the DrawingArea. We are not concerned with the position of the DrawingArea with respect to the MainWindow in this routine. All we need to do is copy the pixmap directly into the window. X ensures that the visible portion of the window is clipped as necessary.

In this example, the `ManagerGadget` actions don't do anything unless the pointer is inside the *Clear* button, so the translation is relatively safe. However, you should be sure to remember that both actions are called. If you press Button 1 inside the PushButton and doodle around a bit before releasing it, the drawing is still done, even though the result is hidden by the gadget. In another application, the fact that actions for both the drawing area itself and its gadget children are both called might lead to indeterminate results.

The `draw()` action does not (and cannot) know if the gadget is also going to react to the button event. This problem does not exist with the standard `DrawingAreaInput()` action routine used in the previous examples because that routine is implemented by the Motif toolkit and it uses its own internal mechanisms to determine if the gadget is activated as well. If the DrawingArea does process the event on the gadget, the `DrawingAreaInput()` action knows that it should not invoke the callback function. However, this internal mechanism is not available outside of the widget code. Reordering the action functions does not help, since there is still no way to know, without making an educated guess, whether or not the DrawingArea acted upon an event on behalf of a gadget child.

As a result of this problem, `draw()` starts drawing a line, even if it starts in the middle of the PushButton, because the DrawingArea processes all of the action functions in the list. If you drag the pointer out of the gadget before releasing the mouse button, the starting point of the line is inside the gadget, but it is hidden when the gadget repaints itself. However, in this particular situation, you can do some guesswork. By installing an `XmNarmCallback` function, you can tell whether or not the DrawingArea activated a button, and by setting an internal state variable, you can decide whether or not the `draw()` action routine should do its drawing.

This confusing behavior is yet another reason why it is best not to include children in DrawingArea widgets that are intended for interactive graphics. If the DrawingArea does not have any gadget children, installing these auxiliary actions in the translation table is not necessary.

Using Color in a DrawingArea

In this section, we expand on our previous examples by incorporating color. The choice of colors is primarily supported by a function we define called `set_color()`, which takes a widget and an arbitrary color name and sets the global `GC`'s foreground color. By providing an array of colors in the form of colored PushButtons, we've got a color paint program. We have removed the PushButton gadget from the DrawingArea and created a proper control panel to the left of the DrawingArea. The program uses a RowColumn widget (see Chapter 8, *Manager Widgets*) to manage a set of eighteen colored PushButtons.* The program that demonstrates these techniques is shown in Example 11-4.†

Example 11-4 The color_draw.c program

```
/* color_draw.c -- simple drawing program using predefined colors.*/
```

* On a monochrome screen, the program runs, but the buttons are either black or white, depending on which is closer to the RGB values corresponding to the color names chosen. You can only draw with the black buttons, since the background is already white.

† `XtVaAppInitialize`() is considered deprecated in X11R6.

```
#include <Xm/MainW.h>
#include <Xm/DrawingA.h>
#include <Xm/PushBG.h>
#include <Xm/PushB.h>
#include <Xm/RowColumn.h>
#include <Xm/ScrolledW.h>
#include <Xm/Form.h>

GC gc;
Pixmap pixmap;

/* dimensions of drawing area (pixmap) */
Dimension width, height;

String colors[] = {"Black", "Red", "Green", "Blue", "White", "Navy", "Orange",
                   "Yellow", "Pink", "Magenta", "Cyan", "Brown", "Grey",
                   "LimeGreen", "Turquoise", "Violet", "Wheat", "Purple"};

main (int argc, char *argv[])
{
    Widget          toplevel, main_w, sw, rc, form, drawing_a, pb;
    XtAppContext    app;
    XGCValues g     cv;
    Arg             args[12];
    void            draw(Widget, XEvent *, String *, Cardinal *);
    void            redraw(Widget, XtPointer, XtPointer);
    void            set_color(Widget, XtPointer, XtPointer);
    void            exit(int);
    void            clear_it(Widget, XtPointer, XtPointer);
    int             i, n;
    XtActionsRec    actions;
    XtTranslations  parsed_xa;
    String          translations = /* for the DrawingArea widget */
        "<Btn1Down>:     draw(down)\n\
        <Btn1Up>:        draw(up)\n\
        <Btn1Motion>:    draw(motion)";

    XtSetLanguageProc (NULL, NULL, NULL);
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv, NULL,
                                    sessionShellWidgetClass, NULL);
    /* Create a MainWindow to contain the drawing area */
    main_w = XmCreateForm (toplevel, "main_w", NULL, 0);

    /* Create a GC for drawing (callback). Used a lot -- make global */
    gcv.foreground = WhitePixelOfScreen (XtScreen (main_w));
    gc = XCreateGC (XtDisplay (main_w),
                    RootWindowOfScreen (XtScreen (main_w)),
                    GCForeground, &gcv);

    /* Create a 3-column array of color tiles */
    n = 0;
    XtSetArg (args[n], XmNpacking, XmPACK_COLUMN);        n++;
    XtSetArg (args[n], XmNnumColumns, 3);                 n++;
    XtSetArg (args[n], XmNtopAttachment, XmATTACH_FORM);  n++;
```

```
XtSetArg (args[n], XmNleftAttachment, XmATTACH_FORM); n++;
rc = XmCreateRowColumn (main_w, "rc", args, n);

for (i = 0; i < XtNumber (colors); i++) {
    /* Create a single tile (pixmap) for each color */
    pixmap = XCreatePixmap (XtDisplay (rc),
                            RootWindowOfScreen (XtScreen (rc)), 16, 16,
                            DefaultDepthOfScreen (XtScreen (rc)));
    set_color (rc, colors[i]); /* set gc's color according to name */
    XFillRectangle (XtDisplay (main_w), pixmap, gc, 0, 0, 16, 16);
    n = 0;
    XtSetArg (args[n], XmNlabelType, XmPIXMAP); n++;
    XtSetArg (args[n], XmNlabelPixmap, pixmap); n++;
    pb = XmCreatePushButton (rc, colors[i], args, n);

    /* callback for this pushbutton sets the current color */
    XtAddCallback (pb, XmNactivateCallback, set_color,
                   (XtPointer) colors[i]);
    XtManageChild (pb);
}

XtManageChild (rc);

n = 0;
XtSetArg (args[n], XmNleftAttachment, XmATTACH_FORM);  n++;
XtSetArg (args[n], XmNtopAttachment, XmATTACH_WIDGET); n++;
XtSetArg (args[n], XmNtopWidget, rc);                  n++;
pb = XmCreatePushButton (main_w, "Quit", args, n);
XtAddCallback (pb, XmNactivateCallback, (void (*)()) exit, NULL);
XtManageChild (pb);

/* Clear button -- wait till DrawingArea is created so we can use
** it to pass as client data.
*/
n = 0;
XtSetArg (args[n], XmNleftAttachment, XmATTACH_WIDGET); n++;
XtSetArg (args[n], XmNleftWidget, pb);                  n++;
XtSetArg (args[n], XmNtopAttachment, XmATTACH_WIDGET);  n++;
XtSetArg (args[n], XmNtopWidget, rc);                   n++;
pb = XmCreatePushButton (main_w, "Clear", args, n);
XtManageChild (pb);

n = 0;
XtSetArg (args[n], XmNwidth, 300);                          n++;
XtSetArg (args[n], XmNscrollingPolicy, XmAUTOMATIC);        n++;
XtSetArg (args[n], XmNscrollBarDisplayPolicy, XmAS_NEEDED); n++;
XtSetArg (args[n], XmNtopAttachment, XmATTACH_FORM);        n++;
XtSetArg (args[n], XmNbottomAttachment, XmATTACH_FORM);     n++;
XtSetArg (args[n], XmNleftAttachment, XmATTACH_WIDGET);     n++;
XtSetArg (args[n], XmNleftWidget, rc);                      n++;
XtSetArg (args[n], XmNrightAttachment, XmATTACH_FORM);      n++;
sw = XmCreateScrolledWindow (main_w, "scrolled_win", args, n);

/* Add the "draw" action/function used by the translation table
```

```
    ** parsed by the translations resource below.
    */
    actions.string = "draw";
    actions.proc = draw;
    XtAppAddActions (app, &actions, 1);
    /* Create a DrawingArea widget. Make it 5 inches wide by 6 inches tall.
    ** Don't let it resize so the Clear Button doesn't force a resize.
    */
    parsed_xa = XtParseTranslationTable (translations);
    n = 0;
    XtSetArg (args[n], XmNunitType, Xm1000TH_INCHES); n++;
    XtSetArg (args[n], XmNwidth, 5000);               n++; /* 5 inches */
    XtSetArg (args[n], XmNheight, 6000);              n++; /* 6 inches */
    /* remain this a fixed size */
    XtSetArg (args[n], XmNresizePolicy, XmNONE);      n++;
    XtSetArg (args[n], XmNtranslations, parsed_xa);   n++;
    drawing_a = XmCreateDrawingArea (sw, "drawing_a", args, n);

    /* When scrolled, the drawing area will get expose events */
    XtAddCallback (drawing_a, XmNexposeCallback, redraw, NULL);
    XtManageChild (drawing_a);

    /* Pushing the clear button clears the drawing area widget */
    XtAddCallback (pb, XmNactivateCallback, clear_it,
                   (XtPointer) drawing_a);

    /* convert drawing area back to pixels to get its width and height */
    XtVaSetValues (drawing_a, XmNunitType, XmPIXELS, NULL);
    XtVaGetValues (drawing_a, XmNwidth, &width, XmNheight, &height, NULL);

    /* create a pixmap the same size as the drawing area. */
    pixmap = XCreatePixmap (XtDisplay (drawing_a),
                            RootWindowOfScreen (XtScreen (drawing_a)),
                            width, height,
                            DefaultDepthOfScreen (XtScreen (drawing_a)));
    /* clear pixmap with white */
    set_color (drawing_a, "White");
    XFillRectangle (XtDisplay (drawing_a), pixmap, gc, 0, 0, width, height);

    XtManageChild (sw);
    XtManageChild (main_w);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

/* Action procedure to respond to any of the events from the
** translation table declared in main(). This function is called
** in response to Button1 Down, Up and Motion events. Basically,
** we're just doing a freehand draw -- not lines or anything.
*/
void draw (Widget widget, XEvent *event, String *args, Cardinal *num_args)
{
    static Position x, y;
    XButtonEvent *bevent = (XButtonEvent *) event;
```

```
    if (*num_args != 1)
        XtError ("Wrong number of args!");
    if (strcmp (args[0], "down")) {
        /* if it's not "down", it must either be "up" or "motion"
        ** draw full line from anchor point to new point.
        */
        XDrawLine (bevent->display, bevent->window, gc, x, y,
                   bevent->x, bevent->y);
        XDrawLine (bevent->display, pixmap, gc, x, y,
                   bevent->x, bevent->y);
    }
    /* freehand is really a bunch of line segments; save this point */
    x = bevent->x;
    y = bevent->y;
}

/* Clear the window by clearing the pixmap and calling XCopyArea() */
void clear_it (Widget pb, XtPointer client_data, XtPointer call_data)
{
    Widget drawing_a = (Widget) client_data;
    XmPushButtonCallbackStruct *cbs =
                           (XmPushButtonCallbackStruct *) call_data;

    /* clear pixmap with white */
    XSetForeground (XtDisplay (drawing_a), gc,
                    WhitePixelOfScreen (XtScreen (drawing_a)));
    /* this clears the pixmap */
    XFillRectangle (XtDisplay (drawing_a), pixmap, gc, 0, 0, width, height);
    /* drawing is now done using black; change the gc */
    XSetForeground (XtDisplay (drawing_a), gc,
                    BlackPixelOfScreen (XtScreen (drawing_a)));
    /* render the newly cleared pixmap onto the window */
    XCopyArea (cbs->event->xbutton.display, pixmap, XtWindow (drawing_a),
                gc, 0, 0, width, height, 0, 0);
}

/* redraw is called whenever all or portions of the drawing area is
** exposed. This includes newly exposed portions of the widget resulting
** from the user's interaction with the scrollbars.
*/
void redraw (Widget drawing_a, XtPointer client_data, XtPointer call_data)
{
    XmDrawingAreaCallbackStruct *cbs =
                           (XmDrawingAreaCallbackStruct *) call_data;
    XCopyArea (cbs->event->xexpose.display, pixmap, cbs->window,
                gc, 0, 0, width, height, 0, 0);
}

/* callback routine for when any of the color tiles are pressed.
** This general function may also be used to set the global gc's
** color directly. Just provide a widget and a color name.
*/
void set_color (Widget widget, XtPointer client_data, XtPointer call_data)
```

```
{
    String      color = (String) client_data;
    Display     *dpy = XtDisplay (widget);
    Colormap    cmap = DefaultColormapOfScreen (XtScreen (widget));
    XColor      col, unused;

    if (!XAllocNamedColor (dpy, cmap, color, &col, &unused)) {
        char buf[32];
        sprintf (buf, "Can't alloc %s", color);
        XtWarning (buf);
        return;
    }
    XSetForeground (dpy, gc, col.pixel);
}
```

The output of the program in shown in Figure 11-3.

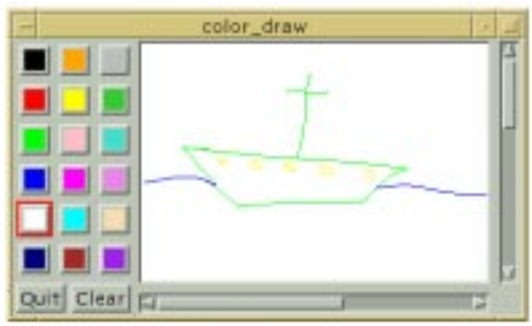


Figure 11-3 Output of the color_draw program

One thing to note about the program is that the callback routine for the *Clear* button is passed the DrawingArea widget as the client data. This technique saves us from having to declare a global variable, while still providing a handle to the DrawingArea in the callback routine.

Summary

The DrawingArea widget is probably most useful when it is used as a canvas for displaying raster images, animation, or a mixture of text and graphics. It is also well-suited for tasks that require interactive user input. The widget provides some rudimentary input mechanisms in the form of callbacks that are invoked by button events.

The translation and action tables supported by the X Toolkit Intrinsics provide a simple mechanism for notifying applications of user events such as double-mouse clicks, keyboard events, and so on. By creatively modifying the default translations and actions, you could build a rather intricate system of action functions that produces interesting graphics based on various forms of user input sequences.

However, what you can do with actions is simplistic given the complexities that are involved in true paint or draw applications. Applications that require a graphic front end should probably dig deeper into the lower levels of Xt for event handling and into Xlib for image rendering.

Exercises

There are a number of different possibilities you could explore in extending the DrawingArea widget. The following exercises are intended to shine the light down some interesting paths that you can take.

1. As we have demonstrated, a DrawingArea widget needs to be able to redisplay the contents of its window. For the programs in this chapter, we implemented redisplay by duplicating in a pixmap all of the drawing done in the window. When the window needs to be repainted, the pixmap is simply copied into it. However, this technique does not take resizing into account. If the *draw2* application is resized bigger, parts of the window are not properly redrawn because the pixmap is not resized. If you wanted to support a canvas that can grow dynamically, you also have to resize the off-screen pixmap. Modify *draw2.c* so that the pixmap resizes along with the DrawingArea. You need to add a callback for `XmNresizeCallback`. The callback should query the size of the DrawingArea, create a new pixmap, use `XCopyArea()` to copy the old pixmap into the new one, and destroy the old pixmap.
2. The resource `XmNcolormap` can be used to set and get the colormap associated with a DrawingArea widget, using `XtVaSetValues()` and `XtVaGetValues()`. Modify *color_draw.c* to use colormap values rather than predefined colors.
3. A paint program and a draw program differ in the way they internally represent their graphical displays. A paint program usually maintains a background pixmap as demonstrated by *free_hand*, whereas a draw program stores geometric information about the shapes that are drawn. For example, circles can be represented using a center (x, y coordinate) and a radius; rectangles can be represented by an origin coordinate with width and height values; and freehand drawings can be represented by a list of coordinates (line segments). Entire pictures can be represented by a list of geometric shape definitions.

   Modify *free_hand.c* or *color_draw.c* to use a list of `XSegment` structures to represent the lines that are drawn by the user. Instead of using a pixmap and `XCopyArea()` to repaint the DrawingArea widget on `Expose` events, repaint the picture by calling `XDrawSegments()` and using the data stored in the internal list of `XSegment` structures.
4. In the previous exercise, we gave you some hints about how you might build an interactive drawing application. For those of you who really want to dig into this subject, you can extend the program by giving the user a choice of geometric shapes to draw.

You need to provide a user interface to support an array of object types: arcs, circles, squares, rectangles, lines, and freehand drawings. Based on the user's choice, you have to maintain a state machine that indicates how much of a geometric figure has been drawn. Use a translation table to monitor the events that correspond to the state machine and store the coordinates of key geometric points in internal data structures. Granted, this exercise is no small feat, but it is a great way to kill a weekend!

In this chapter:
- *Labels*
- *PushButtons*
- *ToggleButtons*
- *ArrowButtons*
- *DrawnButtons*
- *Summary*
- *Exercise*

12

Labels and Buttons

This chapter contains an in-depth look at the label and button widgets provided by the Motif toolkit. These widgets are the most commonly used primitive widgets.

Labels and buttons are among the most widely used interface objects in GUI-based applications. They are also the simplest in concept and design. Labels provide the basic resources necessary to render and manage text or images (pixmaps) by controlling color, alignment, and other visual attributes. PushButtons are subclassed from Label; they extend its capabilities by adding callback routines that respond to user interaction from the mouse or keyboard. These visual and interactive features provide the cornerstone for many widgets in the Motif toolkit, such as CascadeButtons, DrawnButtons, and ToggleButtons.

This chapter also discusses ArrowButtons. While the ArrowButton is not subclassed from Label like the other buttons, it does provide a subset of the interactive capabilities of the other buttons. ArrowButtons do not contain text or graphical labels; they simply display directional arrows that point up, down, left, or right. These widgets are meant to act as companions to other interface objects whose values or displays can be controlled or changed incrementally by the user. An example might be four ArrowButtons that are used to represent directional movement for the display of a bitmap editor.

Although CascadeButtons are subclassed from the Label widget, they are specifically used in Motif menus and are not addressed in this chapter. The menu systems that are provided by Motif are separate entities and are treated separately in Chapter 4, *The Main Window*, and Chapter 20, *Interacting With the Window Manager*. Since the Motif menus use Labels and PushButtons for menu items, these widgets have certain resources that only take effect when the widgets are used in menus. These resources are not discussed in this chapter either.

Labels and buttons have a wide range of uses and they are used in many of the compound objects provided by the Motif toolkit. As a result, these widgets are discussed throughout this book. This chapter provides a basic discussion of the main resources and callbacks used by the objects. It also provides examples of common usage and attempts to address problem areas.

Labels

Labels are simply props for the stage. They are not intended to respond to user interaction, although a help callback can be attached in case the HELP key is pressed. It is equally common to find Labels displaying either text or graphics, yet they cannot display both simultaneously in the conventional sense.

Since Labels can display text, it may not always be obvious whether to use a Label or a Text widget to display textual information. The *Motif Style Guide* suggests that Labels should always be used when non editable text is displayed, even if the text is longer than what you might think of as a label. If a Label is large, you can always place it in the work area of an automatic ScrolledWindow widget, as discussed in Chapter 10, *ScrolledWindows and ScrollBars*. Even if the text is expected to change frequently, your needs can often be accommodated by a Label widget or gadget.

Another issue that affects the choice between a Label widget and a Text widget is the ability to select the text. Even if you have text that is not editable by the user, you may wish to allow the user to select all or part of the text. The Label widget acts as a drag source for drag and drop operations, which means that the full text of a Label can be manipulated using drag and drop.* However, this capability does not allow the user to manipulate only part of the text. For that type of interaction, and with previous versions of the toolkit, you need to use a Text widget rather than a Label to provide selection capabilities.

Labels have a number of added visual advantages over Text widgets. The text in a Label can be greyed out when it is insensitive and it can display text using multiple fonts. The Text widgets, however, do not support multiple fonts†. An insensitive Text widget also greys out its text. Labels are also lighter-weight objects than Text widgets. There is little overhead in maintaining or displaying a Label and there is no need to handle event processing on a Label to the same degree as for a Text widget. All things considered, we would recommend using Label widgets over Text widgets for read-only information, except where the user needs to be able to select and copy the value.

However, when it comes to interactive objects, Labels are not the best choice. In most cases where you want to allow the user to click on a Label, it is more appropriate to use a PushButton or a ToggleButton, since they are designed to support user interaction. Furthermore, users who are familiar with other Motif applications will not expect to have to interact with Labels. In short, the best thing to do with Label widgets is simple and obvious: use them to display labels.

---

* All of the button subclasses of Label inherit the drag source capability, so the text labels for PushButtons and ToggleButtons can also be manipulated using drag and drop. From Motif 2.0, dragging from a Label may be disabled if the `XmDisplay` resource `XmNenableUnselectableDrag` is `False`.

† The Motif 2.0 CSText widget had a compound text interface, and as such could display multiple fonts. However, this widget had serious performance and other problems, and was removed from the widget set in Motif 2.1.

There are a number of resources associated with Labels that are used by other Motif objects (or by widget classes that are subclassed from Label). For example, since Labels (and PushButtons) are used extensively as menu items in menus, they can have accelerators, mnemonics, and other visual resources set to provide the appropriate functionality for menus. These resources do not apply to Labels (and PushButtons) that are not used as menu items, so we do not discuss them here.

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

Creating a Label

Applications that use Labels must include the header file <*Xm/Label.h*>, which defines the `xmLabelWidgetClass` type. This type is a pointer to the actual widget structure used by `XtVaCreateWidget`() or `XtVaCreateManagedWidget()` routines. Motif as usual defines a convenience function, and thus you can create a Label in any of the following ways:

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

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

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

Since Labels do not have children, there is very little reason in terms of performance to create them as unmanaged widgets first and then manage them later. For consistency, we will prefer the Motif convenience functions, and subsequently manage the widget at the appropriate point.

Label gadgets are also available. Recall that a gadget is a windowless object that relies on its parent to provide it with events generated either by the system or by the user†.

The Label gadget is an entirely different class from its widget counterpart. To use the gadget variant, you must include the header file <*Xm/LabelG.h*> and use the

* Whether a Label receives `Help` events depends on the input policy the user is using and whether or not keyboard traversal is on. Since it may not be possible for the user to use the HELP key on Labels, we don't recommend providing help callbacks for them.

† Gadgets in Motif 1.2 also rely on their parent for inherited visual characteristics. From Motif 2.0, however, a gadget can be painted independently of its parent.

xmLabelGadgetClass pointer in XtVaCreateManagedWidget() or XtVaCreateWidget() calls, otherwise use the Motif convenience routine XmCreateLabelGadget(), as in the following examples:

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


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


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

Text Labels

A Label widget or gadget can display either text or an image. The XmNlabelType resource controls the type of label that is displayed; the resource can be set to XmSTRING or XmPIXMAP. The default value is XmSTRING, so if you want to display text in a Label, you do not need to set this resource explicitly.

The resource that specifies the string that is displayed in a Label is XmNlabelString. The value for this resource must be a Motif compound string; common C character strings are not allowed. The following code fragment shows the appropriate way to specify the text for a Label:

```
Widget label;
Arg args[...];
int n= 0;
XmString str = XmStringCreateLocalized ("A Label");

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

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

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

This method of specifying the label string for the widget is much simpler than using a compound string. It avoids the overhead of creating and destroying a compound string, which is expensive in terms of allocating and freeing memory. The problem with the name of the widget shown above is that it is illegal as a widget name. Technically, widget names

should only be composed of alphanumerics (letters and numbers), hyphens, and underscores. Characters such as space, dot (.), and the asterisk (*) are disallowed because they make it impossible for the user to specify these widgets in resource files. On the other hand, using names that contain these characters in your code can be to your advantage if you want to try to prevent users from externally changing the resource values of certain widgets. You can achieve the same result by hard-coding the label or by using an illegal widget name. The first method is more elegant, so the decision you make here should be well-informed.

If you are going to hard-code the label string, you can avoid the overhead of creating a compound string by using the XtVaTypedArg feature of Xt, as shown in the following example:

```
label = XtVaCreateManagedWidget ( "widget_name", xmLabelWidgetClass, parent,
                                  XtVaTypedArg, XmNlabelString, XmRString,
                                  "A Label", 8, /* strlen("A Label") + 1 */
                                  NULL);
```

The C string "A Label" (which is 7 chars long, plus 1 NULL byte) is automatically converted into a compound string by the toolkit using a pre-installed type converter. This method can also be used to change the label for a widget using XtVaSetValues().

Since compound strings are dynamically created and destroyed, you cannot statically declare an argument list that contains a pointer to a compound string. For example, it would be an error to do the following:

```
static Arg list[] = {... XmNlabelString,
                         XmStringCreateLocalized ("A label"), ...};
label = XmCreateLabel (parent, "name", list, XtNumber (list));
```

This technique causes an error because you cannot create a compound string in a statically declared array. For a complete discussion of compound strings, see Chapter 25, *Compound Strings*.

Images as Labels

A Label widget or gadget can display an image instead of text by setting the XmNlabelType resource to XmPIXMAP. As a result of this resource setting, the Label displays the pixmap specified for the XmNlabelPixmap resource. Example 12-1 demonstrates how pixmaps can be used as labels.*

Example 12-1 The pixmaps.c program

```
/* pixmaps.c -- Demonstrate simple label gadgets in a row column
** Each command line argument represents a bitmap filename. Try
** to load the corresponding pixmap and store in a RowColumn.
```

* XtVaAppInitialize() is considered deprecated in X11R6.

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

main (int argc, char *argv[])
{
    XtAppContext   app;
    Pixel          fg, bg;
    Widget         toplevel, rowcol, pb;
    Arg            args[6];
    int            n;
    int            int_sqrt();

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

    if (argc < 2) {
        puts ("Specify bitmap filenames.");
        exit (1);
    }
    /* create a RowColumn that has an equal number of rows and
    ** columns based on the number of pixmaps it is going to
    ** display (this value is in "argc").
    */
    n = 0;
    XtSetArg (args[n], XmNpacking, XmPACK_COLUMN);      n++;
    XtSetArg (args[n], XmNnumColumns, int_sqrt (argc)); n++;
    rowcol = XmCreateRowColumn (toplevel, "rowcol", args, n);
    /* Get the foreground and background colors of the rowcol to make
    ** all the pixmaps appear using a consistent color.
    */
    XtVaGetValues (rowcol, XmNforeground, &fg, XmNbackground, &bg, NULL);

    while (*++argv) {
        Pixmap pixmap = XmGetPixmap (XtScreen (rowcol), *argv, fg, bg);
        if (pixmap == XmUNSPECIFIED_PIXMAP)
            printf ("Couldn't load %s\n", *argv);
        else {
            n = 0;
            XtSetArg (args[n], XmNlabelType, XmPIXMAP); n++;
            XtSetArg (args[n], XmNlabelPixmap, pixmap); n++;
            pb = XmCreateLabelGadget (rowcol, *argv, args, n);
            XtManageChild (pb);
        }
    }
    XtManageChild (rowcol);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

/* get the integer square root of n -- used to determine the number
** of rows and columns of pixmaps to use in the RowColumn widget.
*/
```

```
int int_sqrt (register int n)
{
    register int i, s = 0, t;
    for (i = 15; i >= 0; i--) {
        t = (s | (1 << i));
        if (t * t <= n)
            s = t;
    }
    return s;
}
```

The program displays a two-dimensional array of pixmaps based on the bitmap files listed on the command line. For example, the following command produces the output shown in Figure 12-1.

```
% pixmaps flagup letters wingdogs xlogo64 calculator tie_fighter
```

Figure 12-1 Output of the pixmaps program

To optimize the use of space by the RowColumn widget, the number of rows and columns is set to the square root of the number of images. For example, if there are nine pixmaps to load, there should be a 3×3 grid of images. Since the number of files to be loaded corresponds to the number of arguments in `argv`, `argc` is passed to `int_sqrt()` to get the integer square root of its value. This value tells us the number of columns to specify for the `XmNnumColumns` resource of the RowColumn.

The bitmap files are read using `XmGetPixmap()`, which is a function that creates a pixmap from the specified file. This file must be in X11 bitmap format. Since the function needs foreground and background colors for the pixmap, we use the colors of the RowColumn. If the specified file cannot be found or if it does not contain a bitmap, the function returns the constant `XmUNSPECIFIED_PIXMAP`.* If this error condition is returned, the program skips the file and goes on to the next one. For more detailed information on `XmGetPixmap()` and other supporting functions, see Chapter 3, *Overview of the Motif Toolkit*.

* `XmUNSPECIFIED_PIXMAP` is not `0` or `NULL`. Many people have a tendency to test for these values upon return of functions that return opaque objects. The literal value is `2`.

Label Sensitivity

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

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

Label Alignment

Within the boundaries of a Label widget or gadget, the text or image that is displayed can be left justified, right justified, or centered. The alignment depends on the value of the XmNalignment resource†, which can have one of the following values:

XmALIGNMENT_BEGINNING XmALIGNMENT_END XmALIGNMENT_CENTER

The default value is XmALIGNMENT_CENTER, which causes the text or pixmap to be centered vertically and horizontally within the widget or gadget. The XmALIGNMENT_BEGINNING and XmALIGNMENT_END values refer to the left and right edges of the widget or gadget when the value for XmNlayoutDirection is set to XmLEFT_TO_RIGHT. If the text used within a Label is read from left-to-right (the default), the beginning of the string is on the left. However, if the text used is read from right-to-left, the alignment values are inverted, as should be the value for XmNlayoutDirection. These values also apply to Labels that display pixmaps.

If you have a set of Labels that are associated with strings of text that are right justified, all of the Labels should use the same alignment and string direction settings for consistency. One way to handle this situation is to set the resources universally (as a class-based resource) for all Labels and subclasses of Labels. For example, if your application is written

* In Motif 1.2, the Label will not display a pixmap when it is insensitive and the XmNlabelInsensitivePixmap is XmUNSPECIFIED_PIXMAP. Any pixmap specified for the resource is *not* stippled by the toolkit: the programmer has to construct or supply an appropriate stippled pixmap herself.

† In Motif 2.0 and later, XmNstringDirection is superseded by the XmNlayoutDirection resource.

for a language that displays text from right-to-left, you may choose to have the following lines in the application defaults file:

```
*XmLabel*.layoutDirection: RIGHT_TO_LEFT
*XmLabelGadget.layoutDirection: RIGHT_TO_LEFT
```

Note that the resource must be set for both the widget and gadget classes. You should also be aware that setting the layout direction does not cause the compound strings for the Labels to be automatically converted to the right direction. Similarly, a Label that uses a compound string with a right-to-left string direction does not automatically set the XmNlayoutDirection resource appropriately. These are internationalization issues if you are thinking of supporting languages that are justified either left-to-right or right-to-left.

The RowColumn manager widget can also be used to enforce consistency by controlling the geometry management of its children. If you are using a RowColumn to lay out a group of Labels (or objects subclassed from Label, such as PushButtons), you can tell the RowColumn to align each of its children in a consistent manner using the XmNentryAlignment resource. This resource takes the same values as the XmNalignment resource for Labels. If the parent of a Label widget or gadget is a RowColumn with its XmNisAligned resource set to True, the XmNalignment resource of each of the Label children is forced to the same value as the XmNentryAlignment resource.

You should note that the alignment is only enforced when the RowColumn resource XmNrowColumnType is XmWORK_AREA. If you are using a RowColumn to arrange components in your application, its type should always be a work area. The other types of the widget are used by the internals of Motif for creating special objects like MenuBars and PulldownMenus. If you set the XmNentryAlignment resource for other types of RowColumn widgets, you may or may not see the alignment effects.

There is also a RowColumn resource that affects the vertical alignment of its children that are Labels, subclasses of Label, and Text widgets. The XmNentryVerticalAlignment resource can take one of the following values:

```
XmALIGNMENT_BASELINE_BOTTOM          XmALIGNMENT_BASELINE_TOP
XmALIGNMENT_CONTENTS_BOTTOM          XmALIGNMENT_CONTENTS_TOP
XmALIGNMENT_CENTER
```

The resource only takes effect when the children of the RowColumn are arranged in rows, which means that the XmNorientation is XmHORIZONTAL. The default value is XmALIGNMENT_CENTER, which causes the center of all of the children in a row to be aligned.

Multi-line and Multi-font Labels

The fonts used within a Label are directly associated with the rendition element tags used in the compound string specified for the XmNlabelString resource. The XmNrenderTable* resource for a Label specifies the mapping between rendition tags and fonts that are used when displaying the text. Since a compound string may use multiple character sets, a Label can display any number of fonts, as specified in the XmNlabelString for the Label. A compound string may also contain embedded newlines and tabs, and color specifications. Example 12-2 shows the use of a Label to display a single compound string that contains a monthly calendar.†

Example 12-2 The xcal.c program

```
/* xcal.c -- display a monthly calendar. The month displayed is a
** single Label widget whose text is generated from the output of
** the "cal" program found on any UNIX machine. popen() is used
** to run the program and read its output. Although this is an
** inefficient method for getting the output of a separate program,
** it suffices for demonstration purposes. A List widget displays
** the months and the user can provide the year as argv[1].
*/
#include <stdio.h>
#include <X11/Xos.h>
#include <Xm/Xm.h>
#include <Xm/List.h>
#include <Xm/Frame.h>
#include <Xm/LabelG.h>
#include <Xm/RowColumn.h>
#include <Xm/SeparatoG.h>

int            year;
XmStringTable  ArgvToXmStringTable(int, char **);
void           FreeXmStringTable(XmStringTable);
char           *months[] =  {"January", "February", "March", "April", "May",
                            "June", "July", "August", "September", "October",
                             "November", "December"};

main (int argc, char *argv[])
{
    Widget         toplevel, frame, rowcol, label, w;
    XtAppContext   app;
    extern void    set_month(Widget, XtPointer, XtPointer);
    XmRenderTable  render_table;
    XmRendition    rendition;
```

* In Motif 2.0 and later, the XmFontList type is obsolete, replaced by the XmRenderTable. Accordingly, the XmNfontList resource is deprecated. For backwards compatibility, a font lists is internally re-implemented as a render table.

† XtVaAppInitialize() is considered deprecated in X11R6. XmFontListEntryCreate(), XmFontListAppendEntry(), XmFontListCreate() and XmFontListAdd() are all deprecated from Motif 2.0 onwards. The XmFontList is deprecated, and replaced with the XmRenderTable and XmRendition objects.

```
Arg            args[10];
int            n;
XmStringTable  strs;
int            month_no;

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

/* Create a render table based on the fonts we're using. These are the
** fonts that are going to be hardcoded in the Label and List widgets.
*/

n = 0;
XtSetArg (args[n], XmNfontName, "-*-courier-bold-r-*--18-*"); n++;
XtSetArg (args[n], XmNfontType, XmFONT_IS_FONT);              n++;
XtSetArg (args[n], XmNloadModel, XmLOAD_IMMEDIATE);           n++;
rendition = XmRenditionCreate (toplevel, "tag1", args, n);
render_table = XmRenderTableAddRenditions (NULL, &rendition, 1,
                                           XmMERGE_NEW);
XmRenditionFree (rendition);

n = 0;
XtSetArg (args[n], XmNfontName, "-*-courier-medium-r-*--18-*"); n++;
XtSetArg (args[n], XmNfontType, XmFONT_IS_FONT);                n++;
XtSetArg (args[n], XmNloadModel, XmLOAD_IMMEDIATE);             n++;
rendition = XmRenditionCreate (toplevel, "tag2", args, n);
render_table = XmRenderTableAddRenditions (render_table, &rendition,
                                           1,XmMERGE_NEW);
XmRenditionFree (rendition);

if (argc > 1) {
    month_no = 1;year = atoi (argv[1]);
} else {
    extern long time(long *);
    long t = time ((long *) 0);
    struct tm *today = localtime (&t);
    year = 1900 + today->tm_year;
    month_no = today->tm_mon+1;
}
/* The RowColumn is the general layout manager for the application.
** It contains two children: a Label gadget that displays the calendar
** month, and a ScrolledList to allow the user to change the month.
*/
n = 0;
XtSetArg (args[n], XmNorientation, XmHORIZONTAL); n++;
rowcol = XmCreateRowColumn (toplevel, "rowcol", args, n);

/* enclose the month in a Frame for decoration. */
frame = XmCreateFrame (rowcol, "frame", NULL, 0);

n = 0;
XtSetArg (args[n], XmNalignment, XmALIGNMENT_BEGINNING); n++;
XtSetArg (args[n], XmNrenderTable, render_table);        n++;
```

```
    label = XmCreateLabelGadget (frame, "month", args, n);
    XtManageChild (label);
    XtManageChild (frame);

    /* create a list of month names */
    strs = ArgvToXmStringTable (XtNumber (months), months);
    w = XmCreateScrolledList (rowcol, "list", NULL, 0);
    XtVaSetValues (w,
                   XmNitems, strs,
                   XmNitemCount, XtNumber (months),
                   XmNrenderTable, render_table,
                   NULL);
    FreeXmStringTable (strs);
    XmRenderTableFree (render_table);
    XtAddCallback (w, XmNbrowseSelectionCallback, set_month,
                   (XtPointer) label);
    XtManageChild (w);
    XmListSelectPos (w, month_no, True);

    /* initialize month */
    XtManageChild (rowcol);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

/* callback function for the List widget -- change the month */
void set_month (Widget w, XtPointer client_data, XtPointer call_data)
{
    register FILE          *pp;
    extern FILE            *popen();
    char                   text[BUFSIZ];
    register char          *p = text;
    XmString               str;
    Widget                 label = (Widget) client_data;
    XmListCallbackStruct   *list_cbs = (XmListCallbackStruct *) call_data;

    /* Ask UNIX to execute the "cal" command and read its output */
    sprintf (text, "cal %d %d", list_cbs->item_position, year);

    if (!(pp = popen (text, "r"))) {
        perror (text);
        return;
    }
    *p = 0;

    while (fgets (p, sizeof (text) - strlen (text), pp))
        p += strlen (p);
    pclose (pp);

    /* display the month using the "tag1" rendition from the
    ** Label gadget's XmNrenderTable
    */
    str = XmStringGenerate ((XtPointer) text, "tag1",
                            XmCHARSET_TEXT, NULL);
```

```
    XtVaSetValues (label, XmNlabelString, str, NULL);
    XmStringFree (str);
}

/* Convert an array of string to an array of compound strings */
XmStringTable ArgvToXmStringTable (int argc, char **argv)
{
    XmStringTable new =
                    (XmStringTable) XtMalloc ((argc+1) * sizeof (XmString));
    if (!new)
        return (XmStringTable) 0;
    new[argc] = (XmString) 0;
    while (--argc >= 0)
        new[argc] = XmStringGenerate ((XtPointer) argv[argc], "tag2",
                                      XmCHARSET_TEXT, NULL);
    return new;
}

/* Free the table created by ArgvToXmStringTable() */
void FreeXmStringTable (XmStringTable argv)
{
    register int i;
    if (!argv)
        return;
    for (i = 0; argv[i]; i++)
        XmStringFree (argv[i]);
    XtFree ((char *) argv);
}
```

The output of this program is shown in Figure 12-2.

Figure 12-2 Output of the xcal program

The principal function in Example 12-2 is `set_month()`. In this function, we call `popen()` to run the `UNIX` program *cal* and read its input into a buffer. Since we know ahead of time about how much text we are going to read, `text` is declared with ample space (`BUFSIZ`). Each line is read consecutively until `fgets()` returns `NULL`, at which time we close the opened process using `pclose()` and convert the text buffer into a compound string. This compound string specifies a render table element tag and it includes newlines because `fgets()` does not strip newline characters from the strings it retrieves.

The program displays the calendar for the month corresponding to the selected item in the List, but only as a single Label widget. If we wanted to display individual days using different fonts (with Sundays greyed out, for example), then the text buffer would have to be parsed. In this case, separate compound strings would be created using a different rendition for the Sunday dates only. Since this exercise is more about manipulating compound strings than it is about Label widgets, we refer you to Chapter 24, *Render Tables*, and Chapter 25, *Compound Strings*, for a detailed discussion of the use of multiple fonts in compound strings. If you want to provide the user with the ability to select individual days from the month displayed, you must parse the dates from the text buffer and you probably want to use separate PushButton widgets for each date. See the Appendix A, *Additional Example Programs*, for an example of this technique.

PushButtons

Since the PushButton is subclassed from Label, a PushButton can do everything that a Label can. However, unlike Labels, PushButtons can interact with the user and invoke functions internal to the underlying application through callback routines. This interactivity is the principal difference between PushButtons and Labels. There are other visual differences, but these are adjusted automatically by the PushButton widget using Label resources.

<Xm/PushB.h> and *<Xm/PushBG.h>* are the header files for PushButton widgets and gadgets, respectively. These objects can be created using `XtCreateWidget()`, `XtVaCreateManagedWidget()`, or the appropriate Motif convenience routine, as in the following code fragments:

```
Widget pushb_w = XtVaCreateWidget ("name", xmPushButtonWidgetClass, parent,
                                   resource-value-list, NULL);
Widget pushb_g = XtVaCreateWidget ("name", xmPushButtonGadgetClass, parent,
                                   resource-value-list, NULL);

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

PushButton Callbacks

The major callback routine associated with the PushButton widget is the `XmNactivateCallback`. The functions associated with this resource are called whenever the user activates the PushButton by pressing the left mouse button over it or by pressing the SPACEBAR when the widget has the keyboard focus.

The other callback routines associated with the PushButton are the `XmNarmCallback` and the `XmNdisarmCallback`. Each function in an arm callback list is called whenever the

user presses the left mouse button when the pointer is over the PushButton. When the PushButton is armed, the top and bottom shadows are inverted and the background of the button changes to the arm color. The arm callback does not indicate that the button has been released. If the user releases the mouse button within the widget, then the activate callback list is invoked. The arm callback is always called before the activate callback, whether or not the activate callback is even called.

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

The activate callback function is by far the most useful of the PushButton callbacks. It is generally unnecessary to register arm and disarm callback functions, unless your application has a specific need to know when the button is pushed and released, even if it is not activated. Example 12-3 demonstrates the use of the various PushButton callbacks.*

Example 12-3 The pushb.c program

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

#include <Xm/PushB.h>

main (int argc, char *argv[])
{
    XtAppContext   app;
    Widget         toplevel, button;
    void           my_callback(Widget, XtPointer, XtPointer);
    XmString       btn_text;
    Arg            args[2];

    XtSetLanguageProc (NULL, NULL, NULL);
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv, NULL,
                                    sessionShellWidgetClass, NULL);
    btn_text = XmStringCreateLocalized ("Push Here");
    XtSetArg (args[0], XmNlabelString, btn_text);
    button = XmCreatePushButton (toplevel, "button", args, 1);
    XmStringFree (btn_text);
```

* XtVaAppInitialize() is considered deprecated in X11R6.

```
    XtAddCallback (button, XmNarmCallback, my_callback, NULL);
    XtAddCallback (button, XmNactivateCallback, my_callback, NULL);
    XtAddCallback (button, XmNdisarmCallback, my_callback, NULL);
    XtManageChild (button);

    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

void my_callback (Widget w, XtPointer client_data, XtPointer call_data)
{
    XmPushButtonCallbackStruct *cbs =
                                (XmPushButtonCallbackStruct *) call_data;

    if (cbs->reason == XmCR_ARM)
        printf ("%s: armed\n", XtName (w));
    else if (cbs->reason == XmCR_DISARM)
        printf ("%s: disarmed\n", XtName (w));
    else
    printf ("%s: pushed %d times\n", XtName (w), cbs->click_count);
}
```

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

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

The `reason` parameter is set to `XmCR_ACTIVATE`, `XmCR_ARM`, or `XmCR_DISARM` depending on the callback that invoked the callback routine. We use this value to decide what action to take in the callback routine. The event that caused the callback routine to be invoked is referenced by the `event` field.

The value of the `click_count` field reflects how many times the PushButton has been clicked repeatedly. A repeated button click is one that occurs during a predefined time segment since the last button click. Repeated button clicks can only be done using the mouse. The time segment that determines whether a button click is repeated is defined by the resource `multiClickTime`*. This resource is not defined in the widget class hierarchy but on a per-display basis; the value should be left to the user to specify independently from the application. You can get or set this value using the functions `XtGetMultiClickTime()` or `XtSetMultiClickTime()`. The time interval is used by Xt's translation manager to determine when multiple events are interpreted as a repeat event. The default value is 200 milliseconds (1/5 of a second).

* There is no definition of this resource in any public header file, Motif, Xt, or otherwise.

Multiple Button Clicks

Unfortunately, there is no way to determine whether you are about to receive multiple button clicks from a PushButton. Each time the user activates the PushButton, the arm callback is invoked, followed by the activate callback, followed by the disarm callback. These three callbacks are invoked regardless of whether multiple clicks have occurred.

The best way to determine whether multiple button clicks have occurred would be for the disarm callback to be called only when there are no more button clicks queued. Under this scenario, the same callback function can be used to determine the end of a multiple button click sequence. However, since the Motif toolkit does not operate this way, we must approach the task of handling multiple button clicks differently. We handle the situation by setting up our own timeout routines independently of Motif and handling multiple clicks through the timeout function. Even though we are going to use an alternate method for handling multiple clicks, we can still use the `click_count` parameter in the callback structure provided by the PushButton callback routine. Our technique is demonstrated in Example 12-4.*

Example 12-4 The multi_click.c program

```
/* multi_click.c -- demonstrate handling multiple PushButton clicks.
** First, obtain the time interval of what constitutes a multiple
** button click from the display and pass this as the client_data
** for the button_click() callback function. In the callback, single
** button clicks set a timer to expire on that interval and call the
** function process_clicks(). Double clicks remove the timer and
** just call process_clicks() directly.
*/

#include <Xm/PushB.h>
XtAppContext app;

main (int argc, char *argv[])
{
    Widget     toplevel, button;
    void       button_click(Widget, XtPointer, XtPointer);
    XmString   btn_text;
    int        interval;
    Arg        args[2];

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

    /* get how long for a double click */
    interval = XtGetMultiClickTime (XtDisplay (toplevel));
    printf ("Interval = %d\n", interval);
```

* `XtVaAppInitialize`() is considered deprecated in X11R6.

```
    btn_text = XmStringCreateLocalized ("Push Here");
    XtSetArg (args[0], XmNlabelString, btn_text);
    button = XmCreatePushButton (toplevel, "button", args, 1);
    XtManageChild (button);
    XmStringFree (btn_text);
    XtAddCallback (button, XmNactivateCallback, button_click,
                   (XtPointer) interval);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

/* Process button clicks. Single clicks set a timer, double clicks
** remove the timer, and extended clicks are ignored.
*/
void button_click (Widget w, XtPointer client_data, XtPointer call_data)
{
    static XtIntervalId    id;
    void                   process_clicks(XtPointer, XtIntervalId *);
    int                    interval = (int) client_data;
    XmPushButtonCallbackStruct *cbs =
                            (XmPushButtonCallbackStruct *) call_data;

    if (cbs->click_count == 1)
        id = XtAppAddTimeOut (app, (unsigned long) interval,
                              process_clicks, (XtPointer) False);
    else if (cbs->click_count == 2) {
        XtRemoveTimeOut (id);
        process_clicks ((XtPointer) True, (XtIntervalId *) 0);
    }
}

/* This function won't be called until we've established whether
** or not a single or a double click has occurred.
*/
void process_clicks (XtPointer client_data, XtIntervalId *id)
{
    int double_click = (int) client_data;

    if (double_click)
        puts ("Double click");
    else
        puts ("Single click");
}
```

The program displays the same basic PushButton widget. First, it obtains the time interval that constitutes a multiple button click from the display. This value is passed as the `client_data` to the PushButton's callback function, `button_click()`. When the user first clicks on the PushButton, the callback function is called, and since it is a single-click at this point, a timer is set to expire on the given time interval. If the timer expires, the function `process_clicks()` is called with `False` as its parameter, which means that a single-click has indeed occurred. However, if a second button click occurs before the timer expires, the timer is removed and `process_clicks()` is called directly with `True` as its

data, to indicate that a double-click has occurred. The function `process_clicks()` can be any function that processes single, double, or multiple clicks, depending on how you modify the example we've provided.

If you run Example 12-4, you may find that you get mixed messages about whether an action is a single or double mouse click. A multiple mouse click means that the user has both pressed and released the mouse button more than once. It is very common for a user to intend to double click on a button only to find that she really invoked a double press; she quickly pressed the mouse button twice, but she failed to release it before the required time interval. This problem makes it difficult to interpret double (multiple) button clicks. It is important that you inform the user of the proper double-clicking method in any accompanying documentation you provide with your application, as attempting to program around this problem will definitely cause you great distress.

If you are going to use multiple button clicks for PushButtons, it is important that the multiple-click actions perform a more global version of the single-click actions. The reason for this recommendation is that if the user intends to perform a double click but doesn't click fast enough, the single-click action is invoked instead of the double-click action. If the two actions are completely different, it can make an application difficult to use.You might also consider displaying some visual cue to the user about the availability of double-click actions. For example, you could use a multi-lined label in a PushButton, where the first line indicates the single-click action and the second line specifies the double-click action. If you use this technique, make sure that your documentation informs the user how to invoke either of the two actions.

While double-clicking is a popular interface technique among application programmers and it is certainly useful for computers with single-button mice, it may not be the best interface for all occasions. Possible error conditions may arise when the user is unfamiliar with single and double-clicking techniques. Users often trip on mouse buttons, causing unintentional multiple clicks. Also, users frequently intend to do one double click yet succeed in doing two single clicks. As a result, they get very upset because the application invokes the wrong action twice as opposed to the right action once. Rather than subjecting your users to possible misinterpretation, it may be better to define an alternate method for providing separate actions for the same PushButton widget.

For example, you could define an action for a SHIFT-modified button click. This action is easy enough for the user to do, it is less subject to ambiguity or accidental usage, and it is much easier to program. The callback function only needs to check the `event` data structure and see if the SHIFT key is down when the button is activated.

The PushButton looks for and reports multiple button-click actions by default, so if you are not interested in multiple button clicks, you should set the resource `XmNmultiClick` to `XmMULTICLICK_DISCARD`. When multiple clicks are discarded, only the first of a series

of clicks are processed; the rest are discarded without notifying the callback routine. To turn multiple clicks back on, set the resource to `XmMULTICLICK_KEEP`.

ToggleButtons

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

Individually, a ToggleButton might be used to indicate whether a file should be opened in overwrite mode or append mode, or whether a mail application should update a folder upon process termination. But for the most part, it is when ToggleButtons are grouped together that they become interesting components of a user interface. A RadioBox is a group of ToggleButtons in which only one may be on at any given time. Like the old AM car radios, when one button is pressed in, all of the others are popped out. A CheckBox is a group of ToggleButtons in which each ToggleButton may be set independently of the others. In a RadioBox the selection indicator is represented by a diamond shape, and in a CheckBox it is represented by a square. In either case, when the button is on, the indicator appears to be pressed in, and when it is off, the indicator appears to be popped out. The indicator can also be configured to internally display a cross or check (tick) mark†, and there are resources specifically to configure the color of the indicator in the on and off state‡.

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

Creating ToggleButtons

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

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

* The range of indicators is extended in Motif 2.0: Motif 1.2 can only display squares and diamonds.

† Crosses, Checks and the like are only available from Motif 2.0 onwards.

‡ Coloration for the off state can only be specified in Motif 2.0 and later.

```
Widget toggle = XmCreateToggleButton ( parent, "name", resource-value-array,
                                       resource-value-count);
```

ToggleButtons are also available in the form of gadgets. To use a ToggleButton gadget, you must include the header file *<Xm/ToggleBG.h>*. ToggleButton gadgets may be created as follows:

```
Widget toggle = XtVaCreateWidget ( "name", xmToggleButtonGadgetClass, parent,
                                   resource-value-list, NULL);
Widget toggle = XmCreateToggleButtonGadget ( parent, "name",
                                  resource-value-array, resource-value-count);
```

As we'll show you later in this section, it is also possible to create ToggleButtons at the same time as you create their RowColumn parent. This technique is commonly used when you create a RadioBox or a CheckBox.

Figure 12-3 shows an example of several different ToggleButtons in various states.

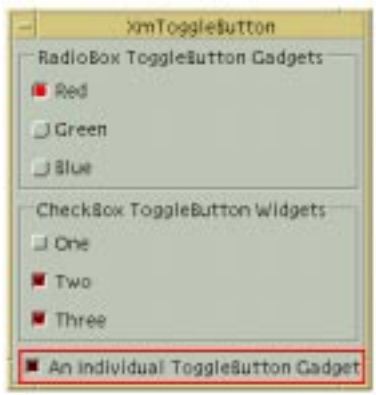


Figure 12-3 ToggleButton widgets and gadgets

ToggleButton Resources

Since ToggleButtons are fairly simple objects, there are only a few resources associated with them aside from those inherited from the Label class. Probably one of the most important of these resources is `XmNindicatorType`, which controls the general shape of the selection indicator that shows whether the ToggleButtons are part of a CheckBox or a RadioBox. Table 12-1 lists the available possibilities and their meanings.*

Table 12-1 The various settings for the XmNindicatorType resource

XmN_OF_MANY	A square button
XmONE_OF_MANY	A round or diamond shaped button
XmONE_OF_MANY_ROUND	A round button
XmONE_OF_MANY_DIAMOND	A diamond-shaped button

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

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

Whilst XmNindicatorType configured the general geometric shape of the indicator, the resource XmNindicatorOn configures the contents*. Table 12-2 lists the possibilities:

Table 12-2 The XmNindicatorOn resource and associated values

XmINDICATOR_NONE	No indicator
XmINDICATOR_FILL	A check box, or box
XmINDICATOR_BOX	A Shadowed Box
XmINDICATOR_CHECK	A check (tick) mark
XmINDICATOR_CHECK_BOX	A check (tick) enclosed in a box
XmINDICATOR_CROSS	A cross
XmINDICATOR_CROSS_BOX	A cross enclosed in a box

The value XmINDICATOR_FILL depends upon the XmDisplay object XmNenableToggleVisual resource. If False, the toggle indicator is a box, which is the Motif 1.2 appearance. If True, the indicator has a check mark.

Toggles are usually thought of as Boolean in nature: they are either on, or off. A Motif 2.x toggle can, however, be tri-state. The third state is the *indeterminate* state, which is neither on, nor off. To configure a Toggle for three states, the XmNtoggleMode resource is set to XmTOGGLE_INDETERMINATE; the default, XmTOGGLE_BOOLEAN, is the normal two-state toggle. In order to programmatically set the Toggle into a given state, the XmNset

* XmONE_OF_MANY_ROUND, XmONE_OF_MANY_DIAMOND are only available from Motif 2.0 onwards.

* XmNindicatorOn in Motif 1.2 is a simple Boolean: display the indicator, or not. We feel that extending the meaning to also encompass the visual appearance itself is possibly confusing, and it is no longer intuitive that this is the resource to set if you want a Cross in a box. There should possibly have been an XmNindicator-Style resource instead of overloading XmNindicatorOn.

resource is used: XmSET, XmUNSET, and XmINDETERMINATE are the relevant values*. Figure 12-4 shows some Toggles in the three states.

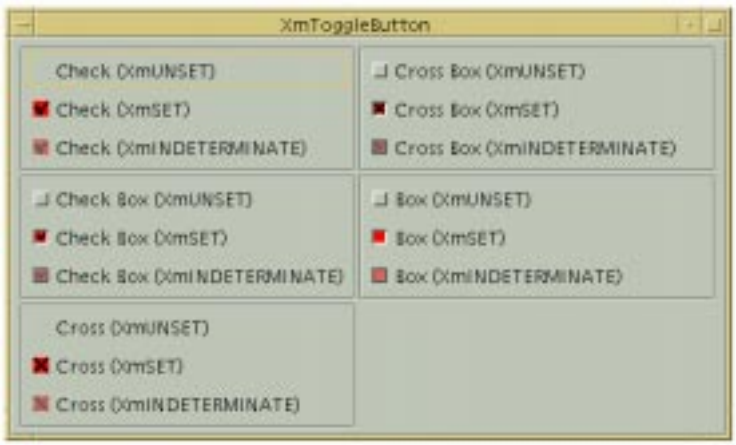


Figure 12-4 ToggleButton states and indicator appearance

Many of the remaining resources are intended mostly for fine-tuning the details of the indicator shape. These details are straightforward and do not require a great deal of discussion. For example, the XmNindicatorSize resource can be used to set the width and height of the indicator. There is nothing magical about these sorts of resources or their side effects, so most are either set automatically by the ToggleButton or they should be left to the user to configure for herself.

ToggleButton Pixmaps

The XmNselectPixmap resource specifies the pixmap to use when a ToggleButton is on (the resource XmNset has the value XmSET). The selected pixmap only applies if the XmNlabelType resource is set to XmPIXMAP. XmNlabelType is a Label class resource, but it applies to ToggleButtons since they are subclassed from Label. The resource XmNindeterminatePixmap specifies the pixmap to use when the Toggle is in the indeterminate state. Example 12-5 demonstrates the creation of a ToggleButton and the use of the XmNselectPixmap and XmNindeterminatePixmap resources.†

Example 12-5 The toggle.c program

```
/* toggle.c -- demonstrate a simple toggle button.
*/
```

* In Motif 1.2, XmNset is a Boolean-valued resource. for backwards compatibility, XmSET is equivalent to True, XmUNSET is equivalent to False.

† XtVaAppInitialize() is considered deprecated in X11R6. XmNindeterminatePixmap is available from Motif 2.0 and later.

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

void toggled (Widget widget, XtPointer client_data, XtPointer call_data)
{
    XmToggleButtonCallbackStruct *state =
                            (XmToggleButtonCallbackStruct *) call_data;
    printf ("%s: %s\n", XtName (widget),
                        state->set == XmSET? "on" :
                        state->set == XmOFF ? "off" : “indeterminate”);
}

main (int argc, char *argv[])
{
    Widget           toplevel, rowcol, toggle;
    XtAppContext     app;
    Pixmap           on, off, unknown;
    Pixel            fg, bg;
    Arg              args[6];
    int              n;

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

    n = 0;
    XtSetArg (args[n], XmNorientation, XmHORIZONTAL); n++;
    rowcol = XmCreateRowColumn (toplevel, "rowcol", args, n);
    XtVaGetValues (rowcol, XmNforeground, &fg, XmNbackground, &bg,   NULL);
    on = XmGetPixmap (XtScreen (rowcol), "switch_on.xbm", fg, bg);
    off = XmGetPixmap (XtScreen (rowcol), "switch_off.xbm", fg, bg);
    unknown = XmGetPixmap (XtScreen (rowcol),
                           "switch_unknown.xbm", fg, bg);
    if (on == XmUNSPECIFIED_PIXMAP  ||
        off == XmUNSPECIFIED_PIXMAP ||
        unknown == XmUNSPECIFIED_PIXMAP) {
            puts ("Couldn't load pixmaps");
            exit (1);
    }

    n = 0;
    XtSetArg (args[n], XmNlabelType, XmPIXMAP);                n++;
    XtSetArg (args[n], XmNtoggleMode, XmTOGGLE_INDETERMINATE); n++;
    XtSetArg (args[n], XmNlabelPixmap, off);                   n++;
    XtSetArg (args[n], XmNselectPixmap, on);                   n++;
    XtSetArg (args[n], XmNindeterminatePixmap, unknown);       n++;

    toggle = XmCreateToggleButton (rowcol, "toggle", args, n);
    XtAddCallback (toggle, XmNvalueChangedCallback, toggled, NULL);
    XtManageChild (toggle);

    toggle = XmCreateToggleButton (rowcol, "toggle",args, n);
    XtAddCallback (toggle, XmNvalueChangedCallback, toggled, NULL);
    XtManageChild (toggle);
```

```
    toggle = XmCreateToggleButton (rowcol, "toggle",args, n);
    XtAddCallback (toggle, XmNvalueChangedCallback, toggled, NULL);
    XtManageChild (toggle);

    XtManageChild (rowcol);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}
```

The output for this program is shown in Figure 12-5. The button on the left shows the ToggleButton when it is in the XmUNSET state, the button in center is the XmINDETERMINATE state, and the button on the right shows it in the XmSET state. The pixmaps illustrate the movement of a simple mechanical switch. Since the pixmaps make the state of the toggle clear, the square indicator is not really necessary. It can be turned off by setting XmNindicatorOn to XmINDICATOR_NONE (its default value is XmINDICATOR_FILL).

Figure 12-5 Output of the toggle program

In order to create the pixmaps for the ToggleButtons, we use the function XmGetPixmap(), which is a general-purpose pixmap loading and caching function. The function needs a foreground and background color for the pixmap it creates, so we retrieve and use the colors from the RowColumn that is the parent of the ToggleButton. XmGetPixmap() loads the pixmaps stored in the files *switch_on.xbm*, *switch_off.xbm*, and *switch_unknown.xbm* in the current directory.* Those files contain the following bitmap definitions:

```
#define switch_on_width 16
#define switch_on_height 16
static unsigned char switch_on_bits[] = {
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x18, 0x00, 0x3c,
        0x00, 0x1e, 0x00, 0x0f, 0x80, 0x07, 0xc0, 0x03, 0xff, 0xff, 0xff, 0xff,
        0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
};

#define switch_off_width 16
#define switch_off_height 16
static unsigned char switch_off_bits[] = {
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x18, 0x00, 0x3c, 0x00,
        0x78, 0x00, 0xf0, 0x00, 0xe0, 0x01, 0xc0, 0x03, 0xff, 0xff, 0xff, 0xff,
        0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
};
```

* The fact that the pixmap files happen to reside in the current directory is not necessarily the recommended method for using XmGetPixmap(). For a complete discussion of the function, see Chapter 3, *Overview of the Motif Toolkit*.

```
#define switch_unknown_width 16
#define switch_unknown_height 16
static unsigned char switch_unknown_bits[] = {
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x80, 0x01, 0x80, 0x01, 0x80, 0x01,
        0x80, 0x01, 0x80, 0x01, 0x80, 0x01, 0x80, 0x01, 0xff, 0xff, 0xff, 0xff,
        0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
};
```

The XmNselectInsensitivePixmap resource can be used to specify a pixmap to be used when the widget or gadget is insensitive, but in a selected state. When a ToggleButton is insensitive, the user cannot change its value interactively. Similarly, XmNindeterminateInsensitivePixmap can be used to display the indeterminate state when the Toggle is insensitive*.

ToggleButton Callbacks

The primary callback routine associated with the ToggleButton is the XmNvalueChangedCallback, which is invoked when the value of the ToggleButton changes. The ToggleButton also has arm and disarm callbacks that are analogous to the callbacks in PushButtons. The callback structure associated with the ToggleButton callback routines is XmToggleButtonCallbackStruct, which is defined as follows:

```
typedef struct {
    int        reason;
    XEvent     *event;
    int        set;
} XmToggleButtonCallbackStruct;
```

When the value of the ToggleButton has changed, the reason field is set to XmCR_VALUE_CHANGED and the set field indicates the current state of the widget.

You can determine the state of a ToggleButton at any time using either XmToggleButtonGetState() or XmToggleButtonGadgetGetState(). These functions take the following form:

```
Boolean XmToggleButtonGetState (Widget toggle_w)
Boolean XmToggleButtonGadgetGetState (Widget toggle_w)
```

Both of the routines return the state of the specified ToggleButton. XmToggleButtonGetState() determines if the *toggle_w* parameter is a widget or a gadget, so you can use the routine on either a ToggleButton widget or a ToggleButton gadget. XmToggleButtonGadgetSetState() can only be used on a gadget.

You can explicitly set the state of a ToggleButton using similar functions: XmToggleButtonSetState() and XmToggleButtonGadgetSetState(). These functions take the following form:

* XmNindeterminateInsensitivePixmap is only available from Motif 2.0 onwards.

```
void XmToggleButtonSetState ( Widget toggle_w, Boolean state,
                              Boolean notify)
void XmToggleButtonGadgetSetState ( Widget toggle_w, Boolean state;
                                    Boolean notify)
```

The *state* argument specifies the state of the ToggleButton. The *notify* parameter allows you to specify whether or not the XmNvalueChangedCallback of the ToggleButton is called when the state is changed. Just like the corresponding get function, XmToggleButtonSetState() determines if its parameter is a widget or gadget internally, so you can use it on either a ToggleButton widget or a ToggleButton gadget. XmToggleButtonGadgetSetState() can only be used on a gadget.

The XmToggleButtonGetState() and XmToggleButtonSetState() functions assume the Motif 1.2 model: the Toggle is a Boolean switch. It is not possible to query or set the Toggle into an indeterminate state using the routines. For a tri-state Toggle, the following functions set the widget state:

```
Boolean XmToggleButtonSetValue ( Widget toggle_w,
                                 XmToggleButtonState state,
                                 Boolean notify)
Boolean XmToggleButtonGadgetSetValue ( Widget toggle_w;
                                       XmToggleButtonState state;
                                       Boolean notify)
```

If the Toggle has the resource XmNtoggleMode set to XmTOGGLE_INDETERMINATE, XmToggleButtonSetValue() and XmToggleButtonGadgetSetValue() set the Toggle to the requested state, call the XmNvalueChangedCallback if *notify* is True, and thereafter return the value True. If XmNtoggleMode is set otherwise, the routines simply return False.

One important point to make about ToggleButtons is that, unlike PushButtons and DrawnButtons, the callback is not typically used to take an action in the application. This point becomes clearer with groups of ToggleButtons, which are commonly used to set the state of various variables. When the user has set the state as desired, she might tell the application to apply the settings by clicking on an associated PushButton. For this reason, the callback routine for a ToggleButton may simply set the state of a global variable; the value can then be used by other application functions.

Of course, like almost every object in Motif, a ToggleButton can be put to many uses. For example, a single ToggleButton could be used to swap the foreground and background colors of a window as soon as the user selects the button. An application that controls a CD player could have a *Pause* button represented by a ToggleButton.

RadioBoxes

When a group of ToggleButtons are used as part of an interface, it is in the form of a RadioBox or a CheckBox. The primary difference between the two is the selection of the

ToggleButtons within. In a RadioBox, only one item may be selected at a time (analogous to old-style AM car radios). You push one button and the previously set button pops out. Examples of exclusive settings in a RadioBox might be baud rate settings for a communications program or U.S. versus European paper sizes in the page setup dialog of a word processing program.

A RadioBox is implemented using a combination of ToggleButton widgets or gadgets and a RowColumn manager widget. As discussed in Chapter 8, *Manager Widgets*, the RowColumn widget is a general-purpose composite widget that manages the layout of its children. The RowColumn has special resources that allow it to act as a RadioBox for a group of ToggleButtons.

In a RadioBox, only one of the buttons may be set at any given time. This functionality is enforced by the RowColumn when the resource `XmNradioBehavior` is set to `True`. For true RadioBox effect, the `XmNradioAlwaysOne` resource can also be set to tell the RowColumn that one of the ToggleButtons should always be set. Whenever `XmNradioBehavior` is set, the RowColumn automatically sets the `XmNindicatorType` resource to `XmONE_OF_MANY` and the `XmNvisibleWhenOff` resource to `True` for all of its ToggleButton children. Furthermore, the `XmNisHomogeneous` resource on the RowColumn is forced to `True` to ensure that no other kinds of widgets can be contained in that RowColumn instance.

Motif provides the convenience function `XmCreateRadioBox()` to automatically create a RowColumn widget that is configured as a RadioBox. This routine creates a RowColumn widget with `XmNisHomogeneous` set to `True`, `XmNentryClass` set to `xmToggleButtonGadgetClass`, `XmNradioBehavior` set to `True`, and `XmNpacking` set to `XmPACK_COLUMN`. Keep in mind that unless `XmNisHomogeneous` is set to `True`, there is nothing restricting a RadioBox from containing other classes as well as ToggleButtons. Whether the RowColumn is homogeneous or not, the toggle behavior is not affected. Although the Motif convenience function sets the homogeneity, it is not a requirement. For example, you might want a RadioBox to contain a Label, or perhaps even some other control area, like a Command widget.

Example 12-6 contains a program that creates and uses a RadioBox.*

Example 12-6 The radio.c program

```
/* radio.c -- demonstrate a simple radio box. Create a
** box with 3 toggles: "one", "two" and "three". The callback
** routine prints the most recently selected choice. Maintain
** a global variable that stores the most recently selected.
*/
#include <Xm/ToggleBG.h>
#include <Xm/RowColumn.h>
```

* `XtVaAppInitialize`() is considered deprecated in X11R6.

```
int toggle_item_set;

void toggled (Widget widget, XtPointer client_data, XtPointer call_data)
{
    int which = (int) client_data;
    XmToggleButtonCallbackStruct *state =
                        (XmToggleButtonCallbackStruct *) call_data;
    printf ("%s: %s\n", XtName (widget),
                    state->set == XmSET ? "on" :
                    state->set == XmOFF ? "off" : “indeterminate”);
    if (state->set == XmSET)
        toggle_item_set = which;
    else
        toggle_item_set = 0;
}

main (int argc, char *argv[])
{
    Widget         toplevel, radio_box, one, two, three;
    XtAppContext   app;

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

    radio_box = XmCreateRadioBox (toplevel, "radio_box", NULL, 0);

    one = XmCreateToggleButtonGadget (radio_box, "One", NULL, 0);
    XtAddCallback (one, XmNvalueChangedCallback, toggled, (XtPointer) 1);
    XtManageChild (one);

    two = XmCreateToggleButtonGadget (radio_box, "Two", NULL, 0);
    XtAddCallback (two, XmNvalueChangedCallback, toggled, (XtPointer) 2);
    XtManageChild (two);

    three = XmCreateToggleButtonGadget (radio_box, "Three", NULL, 0);
    XtAddCallback (three, XmNvalueChangedCallback, toggled, (XtPointer) 3);
    XtManageChild (three);

    XtManageChild (radio_box);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}
```

The program creates three ToggleButtons inside of a RadioBox. When the user selects one of the buttons, the previously-set widget is toggled off, and the `XmNvalueChangedCallback` routine is called. Notice that the routine is called twice for each selection: the first time to notify that the previously set widget has been turned off, and the second time to notify that the newly set widget has been turned on. The output of the program is shown in Figure 12-6.

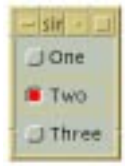


Figure 12-6 Output of the simple_radio program

The global variable `toggle_item_set` indicates which of the three selections is on. The value of `toggle_item_set is` accurate at any given time because it is either set to the most currently selected object or it is set to `0`. In a real application, this global variable would be used to store the state of the buttons, so that other application functions could reference them.

You should beware of lengthy callback lists, however. If you have more than one function in the callback list for the ToggleButtons (unlike the situation shown above), the entire list is going to be called twice. A zero value for `toggle_item_set` indicates that you are in the first of two phases of the toggling mechanism. In this case, you can fall through your callback lists, as the list is called again with the value set to the recently selected toggle item.

Motif provides another RadioBox creation routine, `XmVaCreateSimpleRadioBox()`, for creating simple RadioBoxes. If a RadioBox only has one callback associated with it and you only need to know which button has been selected, this routine may be used. The form of the function is:

```
XmVaCreateSimpleRadioBox (Widget parent, String name, int button_set,
                          void (*callback)(), ..., NULL)
```

In addition to the specified parameters, the function also accepts a `NULL` terminated list of resource-value pairs that apply to the RowColumn widget that acts as the RadioBox. You can specify any normal RowColumn resources in this list, as well as the value `XmVaRADIOBUTTON`, which is a convenient method for specifying a button that is to be created inside the RadioBox. This parameter is followed by four additional arguments: a *`label`* of type `XmString`, a *`mnemonic`* of type `XmKeySym`, an *`accelerator`* of type `String`, and *`accelerator_text`* (also of type `XmString`) that is used to display the accelerator in the widget. You can use `XmVaRADIOBUTTON` multiple times in the same call to `XmVaCreateSimpleRadioBox()`, so that you can create an entire group of ToggleButtons in one function call.

Example 12-7 contains an example of `XmVaCreateSimpleRadioBox()`. This program is functionally identical to the previous example.*

* `XtVaAppInitialize`() is considered deprecated in X11R6.

Example 12-7 The simple_radio.c program

```
/* simple_radio.c -- demonstrate a simple radio box by using
** XmVaCreateSimpleRadioBox(). Create a box with 3 toggles:
** "one", "two" and "three". The callback routine prints
** the most recently selected choice.
*/

#include <Xm/RowColumn.h>

void toggled (Widget widget, XtPointer client_data, XtPointer call_data)
{
    int which = (int) client_data;
    XmToggleButtonCallbackStruct *state =
                            (XmToggleButtonCallbackStruct *) call_data;
    printf ("%s: %s\n", XtName (widget),
                        state->set == XmSET? "on" :
                        state->set == XmOFF ? "off" : “indeterminate”);
}

main (int argc, char *argv[])
{
    Widget       toplevel, radio_box;
    XtAppContext app;
    XmString     one, two, three;

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

    one = XmStringCreateLocalized ("One");
    two = XmStringCreateLocalized ("Two");
    three = XmStringCreateLocalized ("Three");

    radio_box = XmVaCreateSimpleRadioBox (toplevel,
                                "radio_box",
                                0, /* the initial choice */
                                toggled, /* the callback routine */
                                XmVaRADIOBUTTON, one, NULL, NULL, NULL,
                                XmVaRADIOBUTTON, two, NULL, NULL, NULL,
                                XmVaRADIOBUTTON, three, NULL, NULL, NULL,
                                NULL);

    XmStringFree (one);
    XmStringFree (two);
    XmStringFree (three);

    XtManageChild (radio_box);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}
```

CheckBoxes

A CheckBox is similar to a RadioBox, except that there is no restriction on how many items may be selected at once. A word processing program might use a CheckBox for nonexclusive settings, such as whether font smoothing, bitmap smoothing, or both, should be applied.

Like RadioBoxes, CheckBoxes are implemented using RowColumn widgets and ToggleButton children. To allow multiple items to be selected, the `XmNradioBehavior` resource is set to `False`. The convenience routine `XmVaCreateSimpleCheckBox()` works just like the radio box creation routine, except that it turns off the `XmNradioBehavior` resource. Rather than using this function, we can simply create a common RowColumn widget and add ToggleButton children. With this technique, we have more direct control over the resources that are set in the RowColumn, since we can specify exactly which ones we want using the varargs interface for creating the widget.

Example 12-8 demonstrates how to create a CheckBox using a RowColumn widget.*

Example 12-8 The toggle_box.c program

```
/* toggle_box.c -- demonstrate a home-brew ToggleBox. A static
** list of strings is used as the basis for a list of toggles.
** The callback routine toggled() is set for each toggle item.
** The client data for this routine is set to the enumerated
** value of the item with respect to the entire list. This value
** is treated as a bit which is toggled in "toggles_set" -- a
** mask that contains a complete list of all the selected items.
** This list is printed when the PushButton is selected.
*/

#include <Xm/ToggleBG.h>
#include <Xm/PushBG.h>
#include <Xm/SeparatoG.h>
#include <Xm/RowColumn.h>

unsigned long toggles_set; /* has the bits of which toggles are set */
char *strings[] = {"One", "Two", "Three", "Four", "Five",    "Six", "Seven",
                   "Eight", "Nine", "Ten"};
/* A RowColumn is used to manage a ToggleBox (also a RowColumn) and
** a PushButton with a separator gadget in between.
*/
main (int argc, char *argv[])
{
    Widget         toplevel, rowcol, toggle_box, w;
    XtAppContext   app;
    void           toggled(Widget, XtPointer, XtPointer);
    void           check_bits(Widget, XtPointer, XtPointer);
    int            i;
```

* `XtVaAppInitialize`() is considered deprecated in X11R6.

```
    Arg            args[4];

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

    i = 0;
    XtSetArg (args[i], XmNpacking, XmPACK_TIGHT); i++;
    rowcol = XmCreateRowColumn (toplevel, "rowcolumn", args, i);

    i = 0;
    XtSetArg (args[i], XmNpacking, XmPACK_COLUMN); i++;
    XtSetArg (args[i], XmNnumColumns, 2);          i++;
    toggle_box = XmCreateRowColumn (rowcol, "togglebox", args, i);

    /* simply loop through the strings creating a widget for each one */
    for (i = 0; i < XtNumber (strings); i++) {
        w = XmCreateToggleButtonGadget (toggle_box, strings[i], NULL, 0);
        XtAddCallback (w, XmNvalueChangedCallback, toggled, (XtPointer) i);
        XtManageChild (w);
    }

    w = XmCreateSeparatorGadget (rowcol, "sep",NULL, 0);
    XtManageChild (w);
    w = XmCreatePushButtonGadget (rowcol, "Check Toggles",NULL, 0);
    XtAddCallback (w, XmNactivateCallback, check_bits, NULL);
    XtManageChild (w);

    XtManageChild (rowcol);
    XtManageChild (toggle_box);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

/* callback for all ToggleButtons.*/
void toggled (Widget widget, XtPointer client_data, XtPointer call_data)
{
    int bit = (int) client_data;
    XmToggleButtonCallbackStruct *toggle_data =
                                (XmToggleButtonCallbackStruct *) call_data;

    if (toggle_data->set == XmSET)
        /* if the toggle button is set, flip its bit */
        toggles_set |= (1 << bit);
    else /* if the toggle is "off", turn off the bit. */
        toggles_set &= ~(1 << bit);
}

void check_bits (Widget widget, XtPointer client_data, XtPointer call_data)
{
    int i;

    printf ("Toggles set:");
```

```
        for (i = 0; i < XtNumber (strings); i++)
            if (toggles_set & (1<<i))
                printf (" %s", strings[i]);
        putchar ('\n');
    }
```

The output of this program is shown in Figure 12-7.

Figure 12-7 Output of the toggle_box program

This example is similar to the previous RadioBox examples, except that since more than one of the buttons may be set at a time in a CheckBox, we can no longer use `toggle_item_set` the way we did in the previous examples. Instead, we are going to change its name to `toggles_set` and its type to `unsigned long`. This time we are going to use the variable as a *mask*, which means that its individual bits have meaning, rather than the combined value of the variable. The bits indicate which of the ToggleButtons have been set. Each time a ToggleButton changes its value, the callback routine flips the corresponding bit in the mask. We can therefore determine at any given time which buttons are set and which are not.*

The PushButton in the program provides a way to check the state of all of the ToggleButtons. The callback routine for the PushButton prints the strings of those buttons that are selected by looping through the *toggles_set* variable and checking for bits that have been set.

One interesting aspect of this program is that it works just as well if the CheckBox is a RadioBox. To test this statement, we can run the program again with the `XmNradioBehavior` resource set to `True` via the `-xrm` command-line option:

```
toggle_box -xrm "*radioBehavior: True"
```

The result is shown in Figure 12-8.

* The `unsigned long` type can only represent up to 32 ToggleButtons. If more buttons are used within the CheckBox, a new mechanism is needed, although the basic design presented here can still be used.

Figure 12-8 Output of the toggle_box program with XmNradioBehavior set True

As you can see, simply changing this single RowColumn resource completely changes the appearance of all the ToggleButtons.

ArrowButtons

An ArrowButton is just like a PushButton, except that it only displays a directional arrow symbol. The arrow can point up, down, left, or right. Motif provides both widget and gadget versions of the ArrowButton; the associated header files are *<Xm/ArrowB.h>*and *<Xm/ArrowBG.h>*. Example 12-9 shows a program that creates four ArrowButtons, one for each direction.*

Example 12-9 The arrow.c program

```
/* arrow.c -- demonstrate the ArrowButton widget.
** Have a Form widget display 4 ArrowButtons in a
** familiar arrangement.
*/

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

main (int argc, char *argv[])
{
    XtAppContext   app;
    Widget         toplevel, form, arrow;
    Arg            args[6];
    int            n;
    XrmDatabase    xrm_db;

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

* `XtVaAppInitialize`() is considered deprecated in X11R6.

```
xrm_db = XrmGetDatabase(XtDisplay (toplevel));
/* Rather than listing all these resources in an app-defaults file,
** add them directly to the database for this application only. This
** would be virtually equivalent to hard-coding values, since these
** resources will override any other specified external to this file.
*/

XrmPutStringResource (&xrm_db,
                      "*form*topAttachment", "attach_position");
XrmPutStringResource (&xrm_db,
                      "*form*leftAttachment", "attach_position");
XrmPutStringResource (&xrm_db,
                      "*form*rightAttachment", "attach_position");
XrmPutStringResource (&xrm_db,
                      "*form*bottomAttachment","attach_position");

n = 0;
XtSetArg (args[n], XmNfractionBase, 3); n++;
form = XmCreateForm (toplevel, "form",args, n);

n = 0;
XtSetArg (args[n], XmNtopPosition, 0);             n++;
XtSetArg (args[n], XmNbottomPosition, 1);          n++;
XtSetArg (args[n], XmNleftPosition, 1);            n++;
XtSetArg (args[n], XmNrightPosition, 2);           n++;
XtSetArg (args[n], XmNarrowDirection, XmARROW_UP); n++;
arrow = XmCreateArrowButtonGadget (form, "arrow1", args, n);
XtManageChild (arrow);

n = 0;
XtSetArg (args[n], XmNtopPosition, 1);               n++;
XtSetArg (args[n], XmNbottomPosition, 2);            n++;
XtSetArg (args[n], XmNleftPosition, 0);              n++;
XtSetArg (args[n], XmNrightPosition, 1);             n++;
XtSetArg (args[n], XmNarrowDirection, XmARROW_LEFT); n++;
arrow = XmCreateArrowButtonGadget (form, "arrow2", args, n);
XtManageChild (arrow);

n = 0;
XtSetArg (args[n], XmNtopPosition, 1);                n++;
XtSetArg (args[n], XmNbottomPosition, 2);             n++;
XtSetArg (args[n], XmNleftPosition, 2);               n++;
XtSetArg (args[n], XmNrightPosition, 3);              n++;
XtSetArg (args[n], XmNarrowDirection, XmARROW_RIGHT); n++;
arrow = XmCreateArrowButtonGadget (form, "arrow3", args, n);
XtManageChild (arrow);

n = 0;
XtSetArg (args[n], XmNtopPosition, 2);               n++;
XtSetArg (args[n], XmNbottomPosition, 3);            n++;
XtSetArg (args[n], XmNleftPosition, 1);              n++;
XtSetArg (args[n], XmNrightPosition, 2);             n++;
XtSetArg (args[n], XmNarrowDirection, XmARROW_DOWN); n++;
arrow = XmCreateArrowButtonGadget (form, "arrow3", args, n);
```

```
    XtManageChild (arrow);

    XtManageChild (form);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}
```

Figure 12-9 shows the output of this program.

Figure 12-9 Output of the arrow program

The size of the arrow-shaped image is calculated dynamically based on the size of the widget itself. If the widget is resized for some reason, the directional arrow grows or shrinks to fill the widget. The `XmNarrowDirection` resource controls the direction of the arrow displayed by an ArrowButton. This resource may have one of the following values:

`XmARROW_UP` `XmARROW_DOWN` `XmARROW_LEFT` `XmARROW_RIGHT`

ArrowButtons are useful if you want to provide redundant interface methods for certain widgets. For example, you could use ArrowButtons to move the viewport of a ScrolledWindow. Redundancy, when used appropriately, can be an important part of a graphical user interface. Many users may not adapt well to certain interface controls, such as PulldownMenus in MenuBars or keyboard accelerators, while they are perfectly comfortable with iconic controls such as ArrowButtons and PushButtons displaying pixmaps. ArrowButtons are also useful if you want to build your own interface for an object that is not part of the Motif widget set.

ArrowButton widgets and gadgets work in the same way as PushButtons. ArrowButtons have an `XmNactivateCallback`, an `XmNarmCallback`, an `XmNdisarmCallback`, and a `XmNmultiClick` resource. The callback routines all take a parameter of type `XmArrowButtonCallbackStruct`, which is defined as follows:

```
typedef struct {
    int          reason;
    XEvent       *event;
    int          click_count;
} XmArrowButtonCallbackStruct;
```

This callback structure is identical to the one used for PushButtons.

ArrowButtons are commonly used to increment and decrement a value, a position, or another type of data by some arbitrary amount. If the amount being incremented or decremented is sufficiently small in comparison to the total size of the object, it is convenient for the user if you give her the ability to change the value quickly. For example, we can emulate the activate callback routine being called continuously when the user holds down the mouse button over an ArrowButton widget. This functionality is not a feature of the ArrowButton; it is something we have to add ourselves. To implement this feature, we use an Xt timer as demonstrated in Example 12-10.*

Example 12-10 The arrow_timer.c program

```
/* arrow_timer.c -- demonstrate continuous callbacks using
** ArrowButton widgets. Display up and down ArrowButtons and
** attach arm and disarm callbacks to them to start and stop timer
** that is called repeatedly while the button is down. A label
** that has a value changes either positively or negatively
** by single increments while the button is depressed.
*/

#include <Xm/ArrowBG.h>
#include <Xm/Form.h>
#include <Xm/RowColumn.h>
#include <Xm/LabelG.h>

XtAppContext   app;
Widget         label;
XtIntervalId   arrow_timer_id;

typedef struct value_range {
    int value, min, max;
} ValueRange;

main (int argc, char *argv[])
{
    Widget       w, toplevel, rowcol;
    void         start_stop(Widget, XtPointer, XtPointer);
    ValueRange   range;
    Arg          args[6];
    int          n;
    XmString     xms;

    XtSetLanguageProc (NULL, NULL, NULL);

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

    n = 0;
    XtSetArg (args[n], XmNorientation, XmHORIZONTAL); n++;
    rowcol = XmCreateRowColumn (toplevel, "rowcol", args, n);
```

* XtVaAppInitialize() is considered deprecated in X11R6.

```
    n = 0;
    XtSetArg (args[n], XmNarrowDirection, XmARROW_UP); n++;
    w = XmCreateArrowButtonGadget (rowcol, "arrow_up", args, n);
    XtAddCallback (w, XmNarmCallback, start_stop, (XtPointer) 1);
    XtAddCallback (w, XmNdisarmCallback, start_stop, (XtPointer) 1);
    XtManageChild (w);

    n = 0;
    XtSetArg (args[n], XmNarrowDirection, XmARROW_DOWN); n++;
    w = XmCreateArrowButtonGadget (rowcol, "arrow_dn", args, n);
    XtAddCallback (w, XmNarmCallback, start_stop, (XtPointer) -1);
    XtAddCallback (w, XmNdisarmCallback, start_stop, (XtPointer) -1);
    XtManageChild (w);

    range.value = 0;
    range.min = -50;
    range.max = 50;

    n = 0;
    xms = XmStringCreateLocalized("3");
    XtSetArg (args[n], XmNlabelString, xms); n++;
    XtSetArg (args[n], XmNuserData, &range); n++;
    label = XmCreateLabelGadget (rowcol, "label", args, n);
    XmStringFree (xms);
    XtManageChild (label);

    XtManageChild (rowcol);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

/* start_stop is used to start or stop the incremental changes to
** the label's value. When the button goes down, the reason is
** XmCR_ARM and the timer starts. XmCR_DISARM disables the timer.
*/

void start_stop (Widget w, XtPointer client_data, XtPointer call_data)
{
    int incr = (int) client_data;
    XmArrowButtonCallbackStruct *cbs =
                            (XmArrowButtonCallbackStruct *) call_data;
    void change_value(XtPointer, XtIntervalId *);

    if (cbs->reason == XmCR_ARM)
        change_value ((XtPointer) incr, (XtIntervalId *) 1);
    else if (cbs->reason == XmCR_DISARM)
        XtRemoveTimeOut (arrow_timer_id);
}

/* change_value is called each time the timer expires. This function
** is also used to initiate the timer. The "id" represents that timer
** ID returned from the last call to XtAppAddTimeOut(). If id == 1,
** the function was called from start_stop(), not a timeout. If the value
```

```
** has reached its maximum or minimum, don't restart timer, just return.
** If id == 1, this is the first timeout so make it be longer to allow
** the user to release the button and avoid getting into the "speedy"
** part of the timeouts.
*/
void change_value (XtPointer client_data, XtIntervalId *id)
{
    ValueRange    *range;
    char          buf[8];
    int           incr = (int) client_data;
    XmString      xms;

    XtVaGetValues (label, XmNuserData, &range, NULL);

    if (range->value + incr > range->max || range->value + incr < range->min)
        return;

    range->value += incr;
    sprintf (buf, "%d", range->value);
    xms = XmStringCreateLocalized (buf);
    XtVaSetValues (label, XmNlabelString, xms, NULL);
    XmStringFree (xms);

    arrow_timer_id = XtAppAddTimeOut (app,
                    (unsigned long) (id == (XtIntervalId *) 1? 500: 100),
                    change_value, incr);
}
```

The output of this program is shown in Figure 12-10.

Figure 12-10 Output of the arrow_timer program

The program creates up and down ArrowButtons and attaches arm and disarm callbacks that start and stop an internal timer. Each time the timer expires, the value displayed by the Label changes incrementally by one. The timer remains on as long as the button is down. We know that the button has been released when the disarm event occurs.

The function responsible for this behavior is `start_stop()`; it is installed for both the arm and disarm callback. When the button is pressed, the `reason` is `XmCR_ARM`, and the timer starts. When the button is released, the disarm callback is invoked, the `reason` is `XmCR_DISARM`, and the timer is disabled. The `start_stop()` routine initiates the timer by calling `change_value()`. Each time the timer expires, `change_value()` is also called, which means that the function is called repeatedly while the button is pressed. The

`id` represents the ID of the timer that recently expired from the last call to `XtAppAddTimeOut()`. If the value is one, the function was called from `start_stop()`, not as a timeout. We don't restart the timer if the value has reached its maximum or minimum value. If `id` is one, we know that this is the initiating call, so we make the first timeout last longer to allow the user to release the button before getting into the "speedy" timeouts. Otherwise, the time out occurs every 100 milliseconds.

If you experiment with the program, you can get a feel for how the functions work and modify some of the hard-coded values, such as the timeout values. While we demonstrate this technique with ArrowButtons, it can also be applied to a PushButton or any other widget that provides arm and disarm callbacks.*

DrawnButtons

DrawnButtons are similar to PushButtons, except that they also have callback routines for `Expose` and `ConfigureNotify` events. Whenever a DrawnButton is exposed or resized, the corresponding callback routine is responsible for redisplaying the contents of the button. The widget does not handle its own repainting. These callbacks are invoked any time the widget needs to redraw itself, even if it is a result of a change to a resource such as `XmNshadowType`, `XmNshadowThickness`, or the foreground or background color of the widget.

The purpose of the DrawnButton is to allow you to draw into it while maintaining complete control over what the widget displays. Unlike with a PushButton, you are in control of the repainting of the surface area of the widget, not including the bevelled edges that give it a 3D effect. To provide a dynamically changing pixmap using a PushButton widget, you would have to change the `XmNlabelPixmap` resource using `XtVaSetValues()`. Unfortunately, this method results in an annoying flickering effect because the PushButton redisplays itself entirely whenever its pixmap changes. By using the DrawnButton widget, you can dynamically change its display by rendering graphics directly onto the window of the widget using any Xlib routines such as `XDrawLine()` or `XCopyArea()`. This tight control may require more work on your part, but the feedback to the user is greatly improved over the behavior of the PushButton.

DrawnButtons are created similarly to PushButtons and ArrowButtons. However, because the widget provides you with its own drawing area, there is no corresponding gadget version of this object. The associated header file is *<Xm/DrawnB.h>* and it must be included by files that create the widget. Example 12-11 shows a simple example of how a DrawnButton can be created.†

---

* The kind of input arrangement outlined in Example 12-10 is more likely to be implemented as a SpinBox in Motif 2.0 (or, in Motif 2.1, a SimpleSpinBox).

† `XtVaAppInitialize()` is considered deprecated in X11R6.

Example 12-11 The drawn.c program

```
/* drawn.c -- demonstrate the DrawnButton widget by drawing a
** common X logo into its window. This is hardly much different
** from a PushButton widget, but the DrawnButton isn't much
** different, except for a couple more callback routines...
*/

#include <Xm/DrawnB.h>
#include <Xm/BulletinB.h>

Pixmap pixmap;

main (int argc, char *argv[])
{
    XtAppContext   app;
    Widget         toplevel, bb, button;
    Pixel          fg, bg;
    Dimension      ht, st;
    void           my_callback(Widget, XtPointer, XtPointer);

    XtSetLanguageProc (NULL, NULL, NULL);

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

    bb = XmCreateBulletinBoard (toplevel, "bb", NULL, 0);
    XtVaGetValues (bb, XmNforeground, &fg, XmNbackground, &bg, NULL);
    pixmap = XmGetPixmap (XtScreen (bb), "xlogo64", fg, bg);
    button = XmCreateDrawnButton (bb, "button", NULL, 0);
    XtManageChild (button);

    XtVaGetValues (button, XmNhighlightThickness, &ht,
                   XmNshadowThickness, &st, NULL);
    XtVaSetValues (button, XmNwidth, 2 * ht + 2 * st + 64,
    XmNheight, 2 * ht + 2 * st + 64, NULL);

    XtAddCallback (button, XmNactivateCallback, my_callback, NULL);
    XtAddCallback (button, XmNexposeCallback, my_callback, NULL);
    XtAddCallback (button, XmNresizeCallback, my_callback, NULL);

    XtManageChild (bb);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

void my_callback (Widget w, XtPointer client_data, XtPointer call_data)
{
    XmDrawnButtonCallbackStruct *cbs =
                                (XmDrawnButtonCallbackStruct *) call_data;

    if (cbs->reason == XmCR_ACTIVATE)
        printf ("%s: pushed %d times\n", XtName (w), cbs->click_count);
    else if (cbs->reason == XmCR_EXPOSE) {
```

```
            Dimension ht, st;

        XtVaGetValues (w, XmNhighlightThickness, &ht,
        XmNshadowThickness, &st, NULL);
        XtVaSetValues (w, XmNwidth, 2 * ht + 2 * st + 64,
                        XmNheight, 2 * ht + 2 * st + 64, NULL);

        XCopyArea (XtDisplay (w), pixmap, XtWindow (w),
                XDefaultGCOfScreen (XtScreen (w)), 0, 0, 64, 64, ht + st, ht + st);
        }
        else /* XmCR_RESIZE */
            puts ("Resize");
    }
```

The program simply displays the X Window System logo as shown in Figure 12-11.

Figure 12-11 Output of the drawn program

A single callback routine, `my_callback()`, is specified for the `XmNactivateCallback`, `XmNexposeCallback`, and `XmNresizeCallback` callbacks. The callback structure associated with the DrawnButton is called the `XmDrawnButtonCallbackStruct`, which is defined as follows:

```
typedef struct {
    int          reason;
    XEvent       *event;
    Window       window;
    int          click_count;
} XmDrawnButtonCallbackStruct;
```

The `window` field of the structure is the window ID of the DrawnButton widget. This value is the same as that returned by `XtWindow()`. The `my_callback()` callback routine checks the value of the `reason` field to determine which action to take. The `reason` can be one of the following values:

```
XmCR_ACTIVATE                XmCR_ARM                XmCR_DISARM
XmCR_EXPOSE                  XmCR_RESIZE
```

When the `reason` is `XmCR_EXPOSE`, the callback routine handles drawing the X Window System logo in the DrawnButton. Since the widget takes care of drawing its own highlight and shadow, we have to be careful not to draw over these areas.

Since all of the rendering in a DrawnButton is the responsibility of the application, you must decide whether you want to render the graphics differently when the button is

insensitive. Since the DrawnButton is subclassed from the Label class, you can provide a XmNlabelPixmap and XmNlabelInsensitivePixmap if you like, but in this case you might as well use a PushButton instead of a DrawnButton.

In Chapter 26, *Signal Handling*, we present an example that shows how DrawnButtons can be used to construct an *application manager**. An application manager is a program that contains a set of icons, where each icon corresponds to a program. When the user pushes one of the buttons, the corresponding program is run. The button deactivates itself so that only one instance of each application can run at a time. There is no particular reason for this design restriction aside from the fact that it demonstrates the use of the visual resources of the DrawnButton widget.

The XmNpushButtonEnabled resource of the DrawnButton indicates whether or not the DrawnButton should look and act like a PushButton. When the value is False (the default), the DrawnButton displays whatever contents you put in it as well as a shadow border. The style of the shadow is specified by the XmNshadowType resource, which can be set to one of the following values:

XmSHADOW_IN	XmSHADOW_OUT
XmSHADOW_ETCHED_IN	XmSHADOW_ETCHED_OUT

When XmNpushButtonEnabled is False, the button does not provide any feedback to the user when the button is activated.

When the value of XmNpushButtonEnabled is set to True, the DrawnButton behaves like a PushButton and does provide feedback to the user when the button is activated. The shadow border for the button is always drawn in the XmSHADOW_IN style, regardless of the setting of the XmNshadowType resource. When the button is activated, the shadow is reversed, just as for a PushButton.

Summary

The Label class acts as a superclass for more widgets than any other widget in the Motif toolkit and as a result, its use is rather broad. We have presented the fundamentals of Labels, PushButtons, ToggleButtons, ArrowButtons, and DrawnButtons in this chapter. For additional information on these widgets, especially their uses in menu systems, see Chapter 4, *The Main Window*, and Chapter 19, *Menus*. Examples of all these widgets are also liberally spread throughout the rest of the book.

* This is not the natural manner of performing this task in Motif 2.0 or later: the Container and IconGadget combination are specifically designed for this task.

Exercise

The following exercise is intended to stimulate and encourage other creative uses of labels and buttons.

1. Generic X windows have a background pixmap property that can be set using `XSetWindowBackgroundPixmap()`.* Whenever the background pixmap is set, the image is tiled on the window. If the window is larger than the image, the image is replicated in a checkerboard fashion until the window's background is filled; if the window is the same size or smaller than the image, the image is centered in the window. The image is automatically rendered into the window appropriately by the server whenever necessary. Since widgets have windows, the X Toolkit Intrinsics provides a resource for the Core widget class that allows you to set the background pixmap using `XtNbackgroundPixmap`. (Motif's `XmNbackgroundPixmap` resource is identical except that the naming convention provides consistency among resource names.) Write a program that displays a Label that contains both graphics and a text label by setting both `XmNlabelString` and `XmNbackgroundPixmap` to appropriate values.

* See Volume 1, for details on `XSetWindowBackgroundPixmap`().

In this chapter:

- *Creating a List Widget*
- *Using ScrolledLists*
- *Manipulating Items*
- *Positioning the List*
- *Navigating the List*
- *List Callback Routines*
- *Summary*
- *Exercises*

13

The List Widget

This chapter describes another control that the user can manipulate. The List widget displays a number of text choices that the user can select interactively.

Almost every application needs to display lists of choices to the user. This task can be accomplished in many ways, depending on the nature of the choices. For example, a group of ToggleButtons is ideal for displaying configuration settings that can be individually set and unset and then applied all at once. A list of commands can be displayed in a PopupMenu, or for a more permanent command palette, a RowColumn or Form widget can manage a group of PushButton widgets. But for displaying a list of text choices, such as a list of files to be opened or a list of fonts to be applied to text, the List widget is the optimal choice.

A List widget displays a single column of text choices that can be selected or deselected using either the mouse or the keyboard. Each choice is represented by a single-line text element specified as a compound string. Figure 13-1 shows a typical List widget.

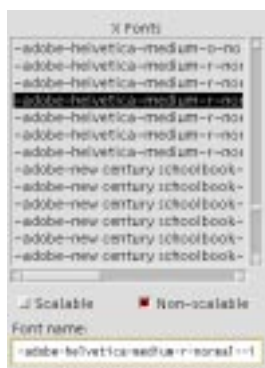


Figure 13-1 A List widget with a selected item

Internally, the List widget operates on an array of compound strings that are defined by the application. (See Chapter 13, *The List Widget*, for a discussion of how to create and manage compound strings. Each string is an element of the array, with the first position starting at one, as opposed to position zero, which is used in C-style arrays. The user can

select a particular choice by clicking and releasing the left mouse button on the item. All of the items in the list are available to the user for selection at all times; you cannot make individual items unselectable. What happens when an item is selected is up to the application callback routines invoked by the List widget.

A List widget is typically a child of a ScrolledWindow, so that the List is displayed with ScrollBars attached to it. The selection mechanism for the List does not change, so the user can still select items as before, but the user can now use the ScrollBars to adjust the items in the list that are visible.

The List widget supports four different selection policies:

- In *single selection* mode, selecting an item toggles its selection state and deselects any other selected item. Single selection Lists should be used when only one of many choices maybe selected at a time, although under this policy there may also be no items selected. Some possible uses for a single selection List include choosing a font family or style for text input and choosing a color for a bitmap editor.
- In *browse selection* mode, selecting a new item deselects any other selected item, but there can never be a state where no items are selected. From the user's perspective, browse selection is similar to single selection, except that there is an initial selected item. There are also differences with respect to callback routines. This issue is addressed in later in this chapter.
- In *multiple selection* mode, any number of items can be selected atone time. When an item is selected, the selection state of the item is toggled; the selection states of the rest of the items are not changed. The List can be in a state where none of the items are selected or all of the items are selected. Multiple selection mode is advantageous in situations where an action may be taken on more than one item at a time, such as in an electronic mail application, where the user might choose to delete, save, or print multiple messages simultaneously.
- In *extended selection* mode, the user can select discontiguous ranges of items. This selection policy is an extension of the multiple selection policy that provides more flexibility.

Creating a List Widget

Using List widgets is fairly straightforward. An application that uses the List widget must include the header file *<Xm/List.h>*. This header file declares the types of the public List functions and the widget class name `xmListWidgetClass`. A List widget can be created as shown in the following code fragment:

```
Widget list = XmCreateList ( parent, "name", resource-value-array,
                             resource-value-count);
Widget list = XtCreateWidget ( "name", xmListWidgetClass, parent,
                               resource-value-list, NULL);
```

Example 13-1 shows a program that creates a simple List widget.*

Example 13-1 The simple_list.c program

```
/* simple_list.c -- introduce the List widget. Lists present
** a number of compound strings as choices. Therefore, strings
** must be converted before set in lists. Also, the number of
** visible items must be set or the List defaults to 1 item.
*/
#include <Xm/List.h>

char *months[] = {"January", "February", "March", "April", "May", "June",
            "July", "August", "September", "October", "November", "December"};

main (int argc, char *argv[])
{
    Widget          toplevel, list;
    XtAppContext    app;
    int             i, n = XtNumber (months);
    XmStringTable   str_list;
    Arg             args[4];

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

    str_list = (XmStringTable) XtMalloc (n * sizeof (XmString));

    for (i = 0; i < n; i++)
        str_list[i] = XmStringCreateLocalized (months[i]);

    i = 0;
    XtSetArg (args[i], XmNvisibleItemCount, n); i++;
    XtSetArg (args[i], XmNitemCount, n);        i++;
    XtSetArg (args[i], XmNitems, str_list);     i++;
    list = XmCreateList (toplevel, "Hello", args, i);

    for (i = 0; i < n; i++)
        XmStringFree (str_list[i]);
    XtFree ((char *) str_list);

    XtManageChild (list);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}
```

The program simply creates a List widget as the child of the `top level` widget. The List contains the names of the months as its choices. The output of the program is shown in Figure 13-2.

* `XtVaAppInitialize`() is considered deprecated in X11R6.

Figure 13-2 Output of the simple_list program

The selection policy of the List is controlled by the XmNselectionPolicy resource. The possible values for this resource are:

XmSINGLE_SELECT	XmBROWSE_SELECT
XmMULTIPLE_SELECT	XmEXTENDED_SELECT

XmBROWSE_SELECT is the default selection policy for the List widget. Since this policy is the one that we want to use, we do not need to set the XmNselectionPolicy resource. You should be aware that the user could change this policy with a resource specification. If you want to enforce this selection policy, you can program defensively and hard-code the value for XmNselectionPolicy, despite its default.

The program demonstrates the use of three basic elements of the List widget: the list of items, the number of items in the list, and the number of visible items. Because the items in a List must be compound strings, each of the choices must be converted from a C string to a compound string. The application allocates an array of XmStrings, creates a compound string for each month name, and stores the string in the str_list. The List widget is created with str_list as the value for the XmNitems resource and XmNitemCount is set to n.

Just like other widgets that use compound strings, the List widget copies the entire table of compound strings into its own internal storage. As a result, the list of strings needs to be freed after you have used it to set the XmNitems resource. When you set the items using this resource, you also need to set the XmNitemCount resource to specify the number of items in the list. If this resource is not set, the List does not know how many items to copy. The value of XmNitemCount should never be larger than the number of items in XmNitems. If the value for XmNitemCount is less than the number of items, the additional items are not put in the list.

To retrieve the list of items, you can call XtVaGetValues() on these resources, as shown in the following code fragment:

```
extern Widget    list;
XmStringTable    choices;
int              n_choices;
XtVaGetValues (list, XmNitems, &choices, XmNitemCount, &n_choices, NULL);
```

Since the items that the area returned are compound strings, you must convert them to C-style strings if you need to use any of the standard C library functions to view or manipulate the strings. You can also use any of the compound string functions described in Chapter 25, *Compound Strings*, for this purpose. Since we used `XtVaGetValues()` to obtain the values for the resources, the returned data should, as always, be considered read-only. You should not change any of the items in the list or attempt to free them (or the pointer to them) when you are done examining their values.

Example 13-1 also makes use of the `XmNvisibleItemCount` resource, which sets the height of the list to match the number of items that should be visible. If you want all the items to be visible, you simply set the value to the total number of items in the list. Setting the visible item count to a higher value is acceptable, assuming that the list is expected to grow to at least that size. If you want to set the number of visible items to be less than the number of items actually in the list, you should use a ScrolledList as described in the next section.

Using ScrolledLists

Most applications use List widgets in conjunction with ScrolledWindows. By creating a List widget as the child of a ScrolledWindow, we create what Motif calls a ScrolledList. The ScrolledList is not a widget, but a compound object. While this chapter describes most of the common resources and functions that deal with ScrolledLists, more detailed information about ScrolledWindows and ScrollBars can be found in Chapter 10, *ScrolledWindows and ScrollBars*.

A ScrolledList is built from two widget classes, so we could create and manage the widgets separately. However, since ScrolledLists are used so frequently, Motif provides a convenience function to create this compound object. `XmCreateScrolledList()` takes the following form:

```
Widget XmCreateScrolledList ( Widget parent, char *name, ArgList arglist,
                              Cardinal argcount)
```

The *`arglist`* parameter is an array of size *`argcount`* that contains resources to be passed to both the ScrolledWindow widget and the List widget. Generally, the two widgets use different resources that are specific to the widgets themselves, so there isn't any confusion about which resources apply to which widget. However, common resources, such as Core resources, are interpreted by both widgets, so caution is advised. If you want to set some resources on one widget, while ensuring that the values are not set on the other widget, you should avoid passing the values to the convenience routine. Instead, you can set resources separately by using the routine `XtVaSetValues()` on each widget individually.

XmCreateScrolledList() returns the List widget; if you need a handle to the ScrolledWindow, you can use XtParent() on the List widget. When you use the convenience routine, you need to manage the object explicitly with XtManageChild().

ScrolledLists are useful because they can display a portion of the entire list provided by the widget. For example, we can modify the previous example, *simple_list.c*, to use a ScrolledList by using the following code fragment:

```
...
/* Create the ScrolledList */
list_w = XmCreateScrolledList (toplevel, "Months", NULL, 0);
/* set the items, the item count, and the visible items */
XtVaSetValues (list_w, XmNitems, str_list, XmNitemCount, n,
                    XmNvisibleItemCount, 5, NULL);
/* Convenience routines don't create managed children */
XtManageChild (list_w);
...
```

The size of the viewport into the entire List widget is controlled by the XmNvisibleItemCount resource. The resource calculates its default value based on the XmNheight of the List. We set the resource to 5. The output resulting from these changes is shown in Figure 13-3.

Figure 13-3 Output of the simple_list program modified to use a ScrolledList

The XmNscrollBarDisplayPolicy and XmNlistSizePolicy resources control the display of the ScrollBars in a ScrolledList widget. The value for XmNscrollBarDisplayPolicy controls the display of the vertical ScrollBar; the resource can be set to either XmAS_NEEDED (the default) or XmSTATIC. If the policy is XmAS_NEEDED, when the entire list is visible, the vertical ScrollBar is not displayed. When the resource is set to XmSTATIC, the vertical ScrollBar is always displayed. The XmNlistSizePolicy resource reflects how the ScrolledList manages its horizontal ScrollBar. The default setting is XmVARIABLE, which means that the ScrolledList attempts to grow horizontally to contain its widest item and a horizontal ScrollBar is not displayed. This policy may present a problem if the parent of the ScrolledList constrains its horizontal size. If the resource is set to XmRESIZE_IF_POSSIBLE, the ScrolledList displays a horizontal ScrollBar only if it cannot resize itself accordingly. If the value XmCONSTANT is used, the horizontal ScrollBar is displayed at all times, whether it is needed or not.

The size of a ScrolledList is ultimately controlled by its parent. In most cases, a manager widget such as a RowColumn or Form allows its children to be any size they request. If a

ScrolledList is a child of a Form widget, its size is whatever you specify with either the XmNheight resource or the XmNvisibleItemCount. However, certain constraints, such as the XmNresizePolicy in a Form widget, may affect the height of its children unexpectedly. For example, if you set XmNresizePolicy to XmRESIZE_NONE, the ScrolledList widget's height request is ignored, which makes it look like XmNvisibleItemCount is not working.

The List widget accepts keyboard input to select items in the list, browse the list, and scroll the list. Like all other Motif widgets, the List has translation functions that facilitate this process. The translations are hard-coded into the widget and we do not recommend attempting to override this list with new translations. For ScrolledLists, the List widget automatically sets the ScrollBar's XmNtraversalOn resource to False so that the ScrollBar associated with the ScrolledList does not get keyboard input. Instead, the List widget handles the input that affects scrolling. We recommended that you do not interfere with this process, so users are not confused by different applications on the desktop behaving in different ways.

If a List widget is sensitive, all of the items in the List are selectable. If it is insensitive, none of them are selectable. You cannot set certain items to be insensitive to selection at any given time. Furthermore, you cannot set the entire List to be insensitive and allow the user to manipulate the ScrollBars. It is not entirely possible to make a read-only List widget; the user always has the ability to select items in the List, providing that it is sensitive. Of course, you can always choose not to hook up callback procedures to the widget, but this can lead to more confusion than anything else because if the user selects an object and the toolkit provides the visual feedback acknowledging the action, the user will expect the application to respond as well.

Manipulating Items

From the programmer's perspective, much of the power of the List widget comes from being able to manipulate its items. The toolkit provides a number of convenience functions for dealing with the items in a List. While the items are accessible through the XmNitems resource, the convenience routines are designed to deal with many common operations, such as adding items to the List, removing items, and locating items.

Adding Items

The entire list of choices may not always be available at the time the List is created. In fact, it is not uncommon to have no items available for a new list. In these situations, items can be added to the list dynamically using the following toolkit functions: XmListAddItem(), XmListAddItemsUnselected(), XmListAddItems(), and XmListAddItemsUnselected(). These functions take the following form:

```
void XmListAddItem (Widget list_w, XmString item, int position)
```

```
void XmListAddItemUnselected (Widget list_w, XmString item, int position)
void XmListAddItems ( Widget list_w, XmString *items, int item_count,
                      int position)
void XmListAddItemsUnselected ( Widget list_w, XmString *items,
                                int item_count, int position)
```

These routines allow you to add one or more items to a List widget at a specified position. Remember that list positions start at `1`, not `0`. The position `0` indicates the last position in the List; specifying this position appends the item or items to the end of the list. If the new item(s) are added to the list in between existing items, the rest of the items are moved down the list.

The difference between `XmListAddItem()` and `XmListAddItemUnselected()` is that `XmListAddItem()` compares each new item to each of the existing items. If a new item matches an existing item and if the existing item is selected, the new item is also selected. `XmListAddItemUnselected()` simply adds the new item without performing this check. In most situations, it is clear which routine you should use. If you know that the new item does not already exist, you should add it unselected. If the List is a single selection list, you should add new items as unselected. The only time that you should really add new items to the list using `XmListAddItem()` is when there could be duplicate entries, the list supports multiple selections, and you explicitly want to select all new items whose duplicates are already selected. The same is true of the routines that add multiple items.

Example 13-2 shows how items can be added to a ScrolledList dynamically using `XmListAddItemUnselected()`.*

Example 13-2 The alpha_list.c program

```
/* alpha_list.c -- insert items into a list in alphabetical order.
*/

#include <Xm/List.h>
#include <Xm/RowColumn.h>
#include <Xm/TextF.h>

main (int argc, char *argv[])
{
    Widget         toplevel, rowcol, list_w, text_w;
    XtAppContext   app;
    Arg            args[5];
    int            n = 0;
    void           add_item(Widget, XtPointer, XtPointer);

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

* `XtVaAppInitialize`() is considered deprecated in X11R6. `XmStringGetLtoR`() is deprecated from Motif 2.0 onwards. `XmStringUnparse`() is only available from Motif 2.0 onwards.

```
    rowcol = XmCreateRowColumn (toplevel, "rowcol", NULL, 0);
    XtSetArg (args[n], XmNvisibleItemCount, 5); n++;
    list_w = XmCreateScrolledList (rowcol, "scrolled_list", args, n);
    XtManageChild (list_w);

    n = 0;
    XtSetArg (args[n], XmNcolumns, 25); n++;
    text_w = XmCreateTextField (rowcol, "text", args, n);
    XtAddCallback (text_w, XmNactivateCallback, add_item,
                   (XtPointer) list_w);
    XtManageChild (text_w);

    XtManageChild (rowcol);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

/* Add item to the list in alphabetical order. Perform binary
** search to find the correct location for the new item position.
** This is the callback routine for the TextField widget.
*/
void add_item (Widget text_w, XtPointer client_data, XtPointer call_data)
{
    Widget     list_w = (Widget) client_data;
    char       *text, *newtext = XmTextFieldGetString (text_w);
    XmString   str, *strlist;
    int        u_bound, l_bound = 0;

    /* newtext is the text typed in the TextField widget */
    if (!newtext || !*newtext) {
        /* non-null strings must be entered */
        XtFree (newtext); /* XtFree() checks for NULL */
        return;
    }

    /* get the current entries (and number of entries) from the List */
    XtVaGetValues (list_w, XmNitemCount, &u_bound,
                         XmNitems, &strlist, NULL);
    u_bound--;

    /* perform binary search */
    while (u_bound >= l_bound) {
        int i = l_bound + (u_bound - l_bound) / 2;

        /* convert the compound string into a regular C string */
        if (!(text = (char *) XmStringUnparse (strlist[i],
                                XmFONTLIST_DEFAULT_TAG,
                                XmCHARSET_TEXT,
                                XmCHARSET_TEXT,
                                NULL, 0,
                                XmOUTPUT_ALL)))
            break;

        if (strcmp (text, newtext) > 0)
```

```
            u_bound = i - 1; /* newtext comes before item */
        else
            l_bound = i + 1; /* newtext comes after item */
        XtFree (text);/* XmStringUnparse() allocates memory */
    }

    str = XmStringCreateLocalized (newtext);
    XtFree (newtext);

    /* positions indexes start at 1, so increment accordingly */
    XmListAddItemUnselected (list_w, str, l_bound+1);
    XmStringFree (str);
    XmTextFieldSetString (text_w, "");
}
```

In Example 13-2, the ScrolledList is created with no items. However, we do specify `XmNvisibleItemCount`, in anticipation of items being added to the list. A TextField widget is used to prompt for strings that are added to the list using the `add_item()` callback. This function performs a binary search on the list to determine the position where the new item is to be added. A binary search can save time, as it is expensive to scan an entire List widget and convert each compound string into a C string. When the position for the new item is found, it is added using `XmListAddItemUnselected()`. The output of this program is shown in Figure 13-4.

Figure 13-4 Output of the alpha_list program

Finding Items

It is often useful to be able to determine whether or not a List contains a particular item. The simplest function for determining whether a particular item exists is `XmListItemExists()`, which takes the following form:

```
Boolean XmListItemExists (Widget list_w;, XmString item)
```

This function performs a linear search on the list for the specified item. If you are maintaining your list in a particular order, you may want to search the list yourself using another type of search to improve performance. The List's internal search function does not convert the compound strings to C strings. The search routine does a direct byte-by-byte comparison of the strings using `XmStringByteCompare()`, which is much more efficient than converting the compound strings to C strings for comparison. However, the linear search is still slower than a binary search by orders of magnitude. And unfortunately,

XmStringByteCompare() does not return which string is of greater or lesser value. The routine just returns whether the strings are different, so we cannot use it to alphabetize the items in a List.

If you need to know the position of an item in the List, you can use XmListItemPos(). This routine takes the following form:

```
int XmListItemPos (Widget list_w, XmString item)
```

This function returns the position of the first occurrence of *item* in the List, with 1 being the first position. If the function returns 0, the element is not in the List. If a List contains duplicate entries, you can find all of the positions of a particular item using XmListGetMatchPos(), which takes the following form:

```
Boolean XmListGetMatchPos ( Widget list_w, XmString item, int **pos_list,
                            int *pos_cnt)
```

This function returns True if the specified item is found in the List in one or more locations. The *pos_list* parameter is allocated to contain the array of positions of the item and the number of items found is returned in *pos_cnt*. When you are done using *pos_list*, you should free it using XtFree(). The function returns False if there are no items in the List, if memory cannot be allocated for *pos_list*, or if the specified item isn't in the List. In these cases, *pos_list* does not point to allocated space and should not be referenced or freed and the value of *pos_cnt* is not specified. The following code fragment shows the use of XmListGetMatchPos() to get the positions of an item in a List:

```
extern Widget    list_w;
int              *pos_list;
int              pos_cnt, i;
char             *choice = "A Sample Text String";
XmString         str = XmStringCreateLocalized (choice);
if (!XmListGetMatchPos (list_w, str, &pos_list, &pos_cnt))
    XtWarning ("Can't get items in list");
else {
    printf ("%s exists in positions %d:", choice, pos_cnt);

    for (i = 0; i < pos_cnt; i++)
        printf (" %d", pos_list[i]);
    puts ("");
    XtFree ((char *) pos_list);
}
```

Replacing Items

There are also a number of functions for replacing items in a List. To replace a contiguous sequence of items, use either XmListReplaceItemsPos() or XmListReplaceItemsPosUnselected(). These functions take the following form:

```
void XmListReplaceItemsPos ( Widget list_w, XmString *new_items,
                             int item_count, int position;)
```

```
void XmListReplaceItemsPosUnselected ( Widget list_w, XmString *new_items,
                                       int item_count, int position)
```

These functions replace the specified number of items with the new items starting at *position*. The difference between the two functions is the same as the difference between the List routines that add items selected and unselected.

You can also replace arbitrary elements in the list with new elements, using XmListReplaceItems() or XmListReplaceItemsUnselected(). These routines take the following form:

```
void XmListReplaceItems ( Widget list_w, XmString *old_items,
                          int item_count, XmString *new_items)
void XmListReplaceItemsUnselected ( Widget list_w, XmString *old_items,
                                    int item_count, XmString *new_items)
```

These functions work by searching the entire list for each element in *old_items*. Every occurrence of each element that is found is replaced with the corresponding element from *new_items*. The search continues for each element in *old_items* until *item_count* has been reached. The difference between the two functions is the same as the difference between the List routines that add items selected and unselected.

There is another routine that allows you to replace items in a List based upon position. The XmListReplacePositions() routine takes the following form:

```
void XmListReplacePositions ( Widget list_w, int *pos_list,
                              XmString *new_items, int item_count)
```

This routine replaces the item at each position specified in *pos_list* with the corresponding item in *new_items* until *item_count* has been reached.

Deleting Items

You can delete items from a List widget in many ways. First, to delete a single item, you can use either XmListDeleteItem() or XmListDeletePos(). These functions take the following form:

```
void XmListDeleteItem (Widget list_w, XmString item)
void XmListDeletePos  (Widget list_w, int position)
```

XmListDeleteItem() finds the given item and deletes it from the list, while XmListDeletePos() removes an item directly from the given position. If you know the position of an item, you can avoid creating a compound string and use XmListDeletePos(). After an item is deleted, the items following it are moved up one position.

You can delete multiple items using either XmListDeleteItems(), XmListDeleteItemsPos(), or XmListDeletePositions(). These routines take the following form:

```
void XmListDeleteItems (Widget list_w, XmString *items, int item_count)
void XmListDeleteItemsPos (Widget list_w, int item_count, int position)
void XmListDeletePositions (Widget list_w, int *pos_list, int pos_count)
```

XmListDeleteItems() deletes each of the items in the *items array* from the List; there are *item_count* strings in the array. You must create and initialize this array before calling the function and you must free it afterwards. If you already know the positions of the items you want to delete, you can avoid creating an array of compound strings and use either of the routines XmListDeleteItemsPos() and XmListDeletePositions(). XmListDeleteItemsPos() deletes *item_count* items from the List starting at *position*. XmListDeletePositions() deletes the item at each position specified in *pos_list* until *item_count* has been reached.

You can delete all of the items in a List widget using XmListDeleteAllItems(). This routine takes the following form:

```
void XmListDeleteAllItems (Widget list_w)
```

Selecting Items

Since the main purpose of the List widget is to allow a user to make a selection from a set of choices, one of the most important tasks for the programmer is to determine which items have been selected by the user. In this section, we present an overview of the resources and functions available to set or get the actual items that are selected in the List widget. Later in this chapter we discuss how to determine the items that are selected by the user when they are selected. The resources and functions used to set and get the selected items in the List widget are directly analogous to those that set the actual items in the list. Just as XmNitems represents the entire list, the XmNselectedItems resource represents the list of selected items. The XmNselectedItemCount resource specifies the number of items that are selected.

There are convenience routines that allow you to modify the items that are selected in a List. The functions XmListSelectItem() and XmListSelectPos() can be used to select individual items. These functions take the following form:

```
void XmListSelectItem (Widget list_w, XmString item, Boolean notify)
void XmListSelectPos (Widget list_w, int position, Boolean notify)
```

These functions cause the specified item to be selected. If you know the position in the list of the item to be selected, you should use XmListSelectPos() rather than XmListSelectItem(). The latter routine uses a linear search to find the specified item. The search can take a long time in a large list, which can affect performance if you are performing frequent list operations.

When the specified item is selected, any other items that have been previously selected are deselected, except when XmNselectionPolicy is set to XmMULTIPLE_SELECT. In this case, the specified item is added to the list of selected items. Even though the extended

selection policy allows multiple items to be selected, the previous selection is deselected when one of these routines is called. If you want to add an item to the list of selected items in an extended selection list, you can set the selection policy to XmMULTIPLE_SELECT, use one of the routines, and then set the selection policy back to XmEXTENDED_SELECT.

The *notify* parameter indicates whether or not the callback routine for the List widget should be called. If your callback routine does special processing of list items, then you can avoid having redundant code by passing True. As a result, the callback routine is called just as if the user had made the selection himself. If you are calling either of these functions from the callback routine, you probably want to pass False to avoid a possible infinite loop.

There are no functions available for selecting multiple items at the same time. To select multiple items, use XtVaSetValues() and set the XmNselectedItems and XmNselectedItemCount resources to the entire list of selected items. From Motif 2.0 onwards, it is also possible to set the XmNselectedPositions and XmNselectedPositionCount resources. Another alternative is to temporarily set XmNselectionPolicy to XmMULTIPLE_SELECT. You can call the above routines repeatedly to select the desired items individually and then set the selection policy back to XmEXTENDED_SELECT.

Items can be deselected in the same manner that they are selected using the routines XmListDeselectItem() and XmListDeselectPos(). These functions take the following form:

```
void XmListDeselectItem (Widget list_w, XmString item)
void XmListDeselectPos (Widget list_w, int position)
```

These routines modify the list of selected items, but they do not have a *notify* parameter, so they do not invoke the callback routine for the List. You can deselect all items in the list by calling XmListDeselectAllItems(), which takes the following form:

```
void XmListDeselectAllItems (Widget list_w)
```

There are also convenience routines that allow you to check on the selected items in a List. You can use XmListPosSelected() to determine whether an item is selected. It takes the following form:

```
Boolean XmListPosSelected (Widget list_w, int position)
```

The routine returns True if the item at the specified position is selected and False otherwise. You can get the positions of all of the selected items in a List using XmListGetSelectedPos(), which takes the following form:

```
Boolean XmListGetSelectedPos (Widget list_w, int **pos_list, int *pos_cnt)
```

This in Motif 2.0 and later does little more than return the value of the XmNselectedPositions and XmNselectedPositionCount resources, which as an alternative may as well be fetched directly using XtGetValues().

The use of the XmListGetSelectedPos() function is identical to that of XmListGetMatchPos(). The *pos_list* parameter is allocated to contain the array of positions of selected items and the number of items selected is returned in *pos_cnt*. When you are done using *pos_list*, you should free it using XtFree(). The function returns False if there are no selected items in the List or if memory cannot be allocated for *pos_list*. In these cases, *pos_list* does not point to allocated space and should not be referenced or freed and the value of *pos_cnt* is not specified.

An Example

In this section, we pull together all of the functions we have described in the preceding sections. This example builds on *alpha_list.c*, the program that adds items that are input by the user to a ScrolledList in alphabetical order. Using another Text widget, the user can also search for items in the list. The searching method uses regular expression pattern-matching functions intrinsic to UNIX systems. Example 13-3 shows the new application.*

Example 13-3 The search_list.c program

```
/* search_list.c -- search for items in a List and select them
*/

#include <stdio.h>
#include <Xm/List.h>
#include <Xm/LabelG.h>
#include <Xm/Label.h>
#include <Xm/RowColumn.h>
#include <Xm/PanedW.h>
#include <Xm/TextF.h>

main (int argc, char *argv[])
{
    Widget         toplevel, rowcol, list_w, text_w, label_w;
    XtAppContext   app;
    Arg            args[5];
    int            n = 0;
    XmString       label;
    void           add_item(Widget, XtPointer, XtPointer);
    void           search_item(Widget, XtPointer, XtPointer);

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

    rowcol = XmCreatePanedWindow (toplevel, "rowcol", NULL, 0);

    label = XmStringCreateLocalized ("List:");
    n = 0;
```

* XtVaAppInitialize() is considered deprecated in X11R6. XmStringGetLtoR() is deprecated from Motif 2.0. XmStringUnparse() is only available from Motif 2.0 onwards.

```
    XtSetArg (args[n], XmNlabelString, label); n++;
    label_w = XmCreateLabel (rowcol, "list_lable", args, n);
    XmStringFree (label);
    XtManageChild (label_w);

    n = 0;
    XtSetArg (args[n], XmNvisibleItemCount, 10);              n++;
    XtSetArg (args[n], XmNselectionPolicy, XmEXTENDED_SELECT); n++;
    list_w = XmCreateScrolledList (rowcol, "scrolled_list", args, n);
    XtManageChild (list_w);

    label = XmStringCreateLocalized ("Add:");
    n = 0;
    XtSetArg (args[n], XmNlabelString, label); n++;
    label_w = XmCreateLabel (rowcol, "add_label", args, n);
    XmStringFree (label);
    XtManageChild (label_w);

    n = 0;
    XtSetArg (args[n], XmNcolumns, 25); n++;
    text_w = XmCreateTextField (rowcol, "add_text",args, n);
    XtAddCallback (text_w, XmNactivateCallback, add_item,
                        (XtPointer) list_w);
    XtManageChild (text_w);

    label = XmStringCreateLocalized ("Search:");
    n = 0;
    XtSetArg (args[n], XmNlabelString, label); n++;
    label_w = XmCreateLabel (rowcol, "search_label", args, n);
    XmStringFree (label);
    XtManageChild (label_w);

    n = 0;
    XtSetArg (args[n], XmNcolumns, 25); n++;
    text_w = XmCreateTextField (rowcol, "search_text", args, n);
    XtAddCallback (text_w, XmNactivateCallback, search_item,
                        (XtPointer) list_w);
    XtManageChild (text_w);

    XtManageChild (rowcol);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

/* Add item to the list in alphabetical order. Perform binary
** search to find the correct location for the new item position.
** This is the callback routine for the Add: TextField widget.
*/
void add_item (Widget text_w, XtPointer client_data, XtPointer call_data)
{
    Widget     list_w = (Widget) client_data;
    char       *text, *newtext = XmTextFieldGetString (text_w);
    XmString   str, *strlist;
    int        u_bound, l_bound = 0;
```

```
    if (!newtext || !*newtext) {
        /* non-null strings must be entered */
        XtFree (newtext);
        return;
    }

    XtVaGetValues (list_w, XmNitemCount, &u_bound,
                   XmNitems, &strlist, NULL);
    u_bound--;

    /* perform binary search */
    while (u_bound >= l_bound) {
        int i = l_bound + (u_bound - l_bound)/2;

        if (!(text = (char *) XmStringUnparse (strlist[i],
                                               XmFONTLIST_DEFAULT_TAG,
                                               XmCHARSET_TEXT,
                                               XmCHARSET_TEXT,
                                               NULL, 0,
                                               XmOUTPUT_ALL)))
            break;
        if (strcmp (text, newtext) > 0)
            u_bound = i-1; /* newtext comes before item */
        else
            l_bound = i+1; /* newtext comes after item */
        XtFree (text);
    }

    str = XmStringCreateLocalized (newtext);
    XtFree (newtext);

    /* positions indexes start at 1, so increment accordingly */
    XmListAddItemUnselected (list_w, str, l_bound+1);
    XmStringFree (str);
    XmTextFieldSetString (text_w, "");
}

/* find the item in the list that matches the specified pattern */
void search_item (Widget text_w, XtPointer client_data,
                  XtPointer call_data)
{
    Widget       list_w = (Widget) client_data;
    char         *exp, *text, *newtext = XmTextFieldGetString (text_w);
    XmString     *strlist, *selectlist = NULL;
    int          matched, cnt, j = 0;
#ifndef SYSV
    extern char  *re_comp();
#endif /* SYSV */

    if (!newtext || !*newtext) {
        /* non-null strings must be entered */
        XtFree (newtext);
        return;
```

```
    }

    /* compile expression into pattern matching library */
#ifdef SYSV
    if (!(exp = regcmp (newtext, NULL))) {
        printf ("Error with regcmp(%s)\n", newtext);
        XtFree (newtext);
        return;
    }
#else /* BSD */
    if (exp = re_comp (newtext)) {
        printf ("Error with re_comp(%s): %s\n", newtext, exp);
        XtFree (newtext);
        return;
    }
#endif /* SYSV */

    /* get all the items in the list... we're going to search each one */
    XtVaGetValues (list_w, XmNitemCount, &cnt, XmNitems, &strlist, NULL);

    while (cnt--) {
        /* convert item to C string */
        if (!(text = (char *) XmStringUnparse (strlist[cnt],
                                               XmFONTLIST_DEFAULT_TAG,
                                               XmCHARSET_TEXT,
                                               XmCHARSET_TEXT,
                                               NULL, 0,
                                               XmOUTPUT_ALL)))
        break;

        /* do pattern match against search string */
#ifdef SYSV
        /* returns NULL if match failed */
        matched = regex (exp, text, NULL) != NULL;
#else /* BSD */
        /* -1 on error, 0 if no-match, 1 if match */
        matched = re_exec (text) > 0;
#endif /* SYSV */
        if (matched) {
            selectlist = (XmString *) XtRealloc ((char *) selectlist,
                            (j+1) * (sizeof (XmString *)));
            selectlist[j++] = XmStringCopy (strlist[cnt]);
        }
        XtFree (text);
    }
#ifdef SYSV
    free (exp);
    /* this must be freed for regcmp() */
#endif /* SYSV */

    XtFree (newtext);

    /* set the actual selected items to be those that matched */
    XtVaSetValues (list_w, XmNselectedItems, selectlist, XmNselectedItemCount,
                   j, NULL);
```

```
        while (j--)
            XmStringFree (selectlist[j]);

        XmTextFieldSetString (text_w, "");
    }
```

The output of this program is shown in Figure 13-5. The TextField widget that is used to search for items in the List widget works identically to the one that is used to add new items. Its callback routine, `search_item()`, searches the list for the specified pattern. The version of UNIX you are running (System V or BSD) dictates which kind of regular expression matching is done. System V machines use the function `regcmp()` to compile the pattern and `regex()` to search for the pattern within another string, while BSD UNIX systems use the function `sre_comp()` and `re_exec()` to do the same thing.*

Figure 13-5 Output of the search_list program

The items in the list are retrieved using `XtVaGetValues()` and the `strlist` parameter. This variable points to the internal list used by the List widget, so it is important that we do not change any of these elements or free these pointers when we are through with them. Changing the value of `XmNselectedItems` causes the internal list to change. Since the internal list is referenced by `strlist`, it is important to copy any values that we want to use elsewhere. If the pattern matches a list item, the item is copied using `XmStringCopy()` and is later added to the List's `XmNselectedItems`.

* Systems that support both BSD and System V may support one, the other, or both methods of regular expression handling. You should consult your system's documentation for more information on these functions.

Positioning the List

The items within a List can be positioned such that an arbitrary element is placed at the top or bottom of the List. If the List is being used as part of a ScrolledList, the item is placed at the top or bottom of the viewport of the ScrolledWindow. To position a particular item at the top or bottom of the window, use either `XmListSetItem()` or `XmListSetBottomItem()`. These routines take the following form:

```
void XmListSetItem (Widget list_w, XmString item)
void XmListBottomItem (Widget list_w, XmString item)
```

Both of these functions require an `XmString` parameter to reference a particular item in the list. However, if you know the position of the item, you can use `XmListSetPos()` or `XmListSetBottomPos()` instead. These functions take the following form:

```
void XmListSetPos (Widget list_w, int position)
void XmListSetBottomPos (Widget list_w, int position)
```

The *position* parameter can be set to `0` to specify that the last item be positioned at the bottom of the viewport. Through a mixture of resource values and simple calculations, you can position any particular item anywhere in the list. For example, if you have an item that you want to be sure is visible, but you are not concerned about where in the viewport it is displayed, you can write a function to make the item visible. Example 13-4 shows the `MakePosVisible()` routine, which makes sure that the item at a specified position is visible.

Example 13-4 The MakePosVisible() routine

```
void MakePosVisible (Widget list_w, int item_no)
{
    int top, visible;

    XtVaGetValues (list_w, XmNtopItemPosition, &top,
                        XmNvisibleItemCount, &visible, NULL);

    if (item_no < top)
        XmListSetPos (list_w, item_no);
    else if (item_no >= top + visible)
        XmListSetBottomPos (list_w, item_no);
}
```

The function gets the number of visible items and the position of the item at the top of the viewport. The `XmNtopItemPosition` resource stores this information. If the item comes before top, `item_no` is set to the top of the List using `XmListSetPos()`. If it comes after `top + visible`, the item is set at the bottom of the List using `XmListSetBottomPos()`. If you don't know the position of the item in the List, you can write a function that makes a specified item visible, as shown in Example 13-5.

Example 13-5 The MakeItemVisible() routine

```
void MakeItemVisible (Widget list_w, XmString item)
{
    int item_no = XmListItemPos (list_w, item);

    if (item_no > 0)
        MakePosVisible (list_w, item_no);
}
```

The `MakeItemVisible()` routine simple gets the position of the given item in the list using `XmListItemPos()` and calls `MakePosVisible()`.

There are some other routines that deal with positions in a List widget. The `XmListGetKbdItemPos()` and `XmListSetKbdItemPos()` routines retrieve and set the item in the List that has the location cursor. These routines take the following form:

```
int XmListGetKbdItemPos (Widget list_w)
Boolean XmListSetKbdItemPos (Widget list_w, int position)
```

`XmListGetKbdItemPos()` returns the position of the item that has the location cursor, while `XmListSetKbdItemPos()` provides a way to specify the position of this item.

The `XmListPosToBounds()` and `XmListYToPos()` functions in provide a way to translate list items to x, y coordinates and vice versa. `XmListPosToBounds()` returns the bounding box of the item at a specified position in a List. This routine takes the following form:

```
Boolean XmListPosToBounds ( Widget          list_w,
                            int             position,
                            Position        *x,
                            Position        *y,
                            Dimension       *width,
                            Dimension       *height)
```

This routine returns `True` if the item at the specified position is visible and `False` otherwise. If the item is visible, the return parameters specify the bounding box of the item. This information can be useful if you need to perform additional event processing or draw special graphics for the item. The `XmListYToPos()` routine returns the position of the List item at a specified y-coordinate. This function takes the following form:

```
int XmListYToPos (Widget list_w, Position y)
```

The position information returned by this routine can be useful if you are processing events that report a pointer position and you need to convert the location of the event into an item position.

Navigating the List

In Motif 2.0 and later, the user can navigate through items in the List simply by typing characters which match the first character of an item. The resource `XmNmatchBehavior`

controls this aspect of the List. Match behavior is enabled by default. To disable, set XmNmatchBehavior to XmNONE, and to re-enable, use the value XmQUICK_NAVIGATE.

Navigation proceeds cyclically: when the user types a character, it is compared against List items starting below the current item. If no item below the current item matches the keyboard input, the search proceeds from the top of the List. If a match is found, the matching item becomes the new current item. Figure 13-6 shows how this all works.

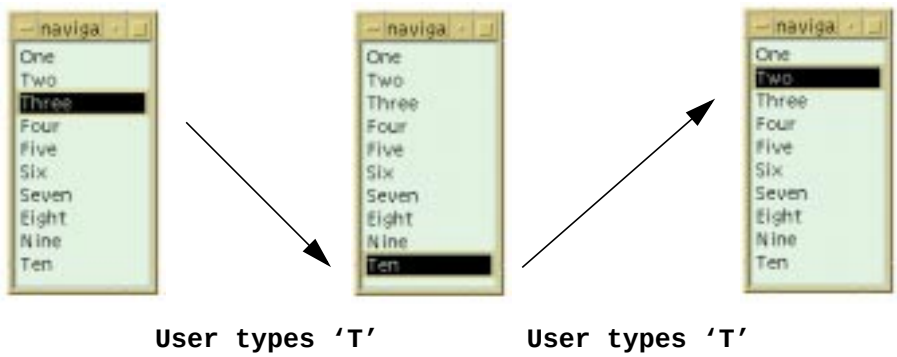


Figure 13-6 XmNmatchBehavior set to XmQUICK_NAVIGATE

If no match is found, XBell() is called automatically: there is no way to configure the List otherwise. Note that matching is case sensitive: in the example, typing a lower case "t" would not have matched against any items in the List.

List Callback Routines

While the callback routines associated with the List widget are not affected by whether the List is scrollable, they do depend on the selection policy currently in use. There is a separate callback resource for each selection policy, plus a callback for the default action. The default action is invoked when the left mouse button is double-clicked on an item or the RETURN key is pressed. The callback resources are:

```
XmNbrowseSelectionCallback
XmNextendedSelectionCallback
XmNsingleSelectionCallback
XmNdefaultActionCallback
XmNmultipleSelectionCallback
```

The Default Action

In all of the selection modes there is the concept of the default *action*. This term refers to the action that is taken when the user double clicks the left mouse button on an item or presses the RETURN key when an item has the location cursor. The default action always indicates that the active item should be selected, regardless of the selection policy. The XmNdefaultActionCallback is invoked for the default action.

The default selection is activated when the user double clicks on a List item. The time interval between two consecutive button clicks determines whether the clicks are interpreted as individual clicks or as a double click. You can set or get the time interval using the XmNdoubleClickInterval resource. The value is stored as milliseconds, so a value of 500 is half a second. If the resource is not set, the value of the multiClickTime resource is used instead. This resource is a fundamental X resource that is understood by all X applications; it is not an Xt or Motif toolkit resource. You should let the user specify the double-click interval in a resource file; the value should be set using the more global multiClickTime resource.

Browse and Single Selection Callbacks

The browse and single selection modes only allow the selection of a single item. The browsing mode is regarded as a simpler interface for the user. Interactively, browse selection allows the user to drag the selection over many items; the selection is not made till the mouse button is released. In the single selection mode, the selection is made as soon as the mouse button is pressed. For browse selection, the callback list associated with the XmNbrowseSelectionCallback is used, while the XmNsingleSelectionCallback is used for the single selection mode.

Keyboard traversal in the List is also different between the two modes. If the user uses the keyboard to move from one item to the next in single selection mode, the XmNsingleSelectCallback is not invoked until the SPACEBAR is pressed. In browse selection, the XmNbrowseSelectionCallback is invoked for each item the user traverses. Since these two modes for the List widget are visually similar, your treatment of the callbacks is very important for maintaining consistency between Lists that use different selection modes.

A simple example of using callbacks with a List widget is shown in Example 13-6.*

Example 13-6 The browse.c program

```
/* browse.c -- specify a browse selection callback for a simple List. */

#include <Xm/List.h>

char *months[] = {"January", "February", "March", "April", "May", "June",
                  "July", "August", "September", "October", "November",
                  "December"};

main (int argc, char *argv[])
{
    Widget         toplevel, list_w;
    XtAppContext   app;
```

* XtVaAppInitialize() is considered deprecated in X11R6. XmStringGetLtoR() is deprecated in Motif 2.0 and later.

```
    int            i, n = XtNumber (months);
    XmStringTable  str_list;
    void           sel_callback(Widget, XtPointer, XtPointer);

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

    str_list = (XmStringTable) XtMalloc (n * sizeof (XmString *));
    for (i = 0; i < n; i++)
        str_list[i] = XmStringCreateLocalized (months[i]);

    list_w = XmCreateScrolledList (toplevel, "months", NULL, 0);
    XtVaSetValues (list_w, XmNvisibleItemCount, n,
                           XmNitemCount, n,
                           XmNitems, str_list, NULL);
    XtManageChild (list_w);
    XtAddCallback (list_w, XmNdefaultActionCallback, sel_callback, NULL);
    XtAddCallback (list_w, XmNbrowseSelectionCallback, sel_callback, NULL);

    for (i = 0; i < n; i++)
        XmStringFree (str_list[i]);
    XtFree ((char *) str_list);

    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

void sel_callback (Widget list_w, XtPointer client_data, XtPointer call_data)
{
    XmListCallbackStruct *cbs = (XmListCallbackStruct *) call_data;
    char *choice;

    if (cbs->reason == XmCR_BROWSE_SELECT)
        printf ("Browse selection -- ");
    else
        printf ("Default action -- ");

    choice = (char *) XmStringUnparse (cbs->item,
                                       XmFONTLIST_DEFAULT_TAG,
                                       XmCHARSET_TEXT,
                                       XmCHARSET_TEXT,
                                       NULL, 0,
                                       XmOUTPUT_ALL);
    printf ("selected item: %s (%d)\n", choice, cbs->item_position);
    XtFree (choice);
}
```

For this example, we modified our previous example that uses a ScrolledList to display the months of the year. We have added the same callback routine, `sel_callback()`, to the `XmNbrowseSelectionCallback` and `XmNdefaultActionCallback` resources. Since the default action may happen for any List widget, it is advisable to set this callback, even if there are other callbacks. The callback routine prints the type of action performed by the

user and the selection that was made. The callback structure is used to get information about the nature of the List widget and the selection made.

The List callbacks provide a callback structure of type XmListCallbackStruct, which is defined as follows:

```
typedef struct {
    int          reason;
    XEvent       *event;
    XmString     item;
    int          item_length;
    int          item_position;
    XmString     *selected_items;
    int          selected_item_count;
    int          *selected_item_positions;
    char         selection_type;
    char         auto_selection_type;*
} XmListCallbackStruct;
```

The reason field specifies the reason that the callback was invoked, which corresponds to the type of action performed by the user. The possible values for this field are:

```
XmCR_BROWSE_SELECT              XmCR_DEFAULT_ACTION
XmCR_EXTENDED_SELECT            XmCR_MULTIPLE_SELECT
XmCR_SINGLE_SELECT
```

The reason field is important with List callbacks because not all of the fields in the callback structure are valid for every reason. For the browse and single selection policies, the reason, event, item, item_length, and item_position fields are valid. For the default action, all of the fields are valid. List items are stored as compound strings in the callback structure, so to print an item using printf(), we must convert the string with the compound string function XmStringUnparse().

Multiple Selection Callback

When XmNselectionPolicy is set to XmMULTIPLE_SELECT, multiple items can be selected in the List widget. When the user selects an item, its selection state is toggled. Each time the user selects an item, the callback routine associated with the XmNmultipleSelectionCallback is invoked. Example 13-7 shows the sel_callback() routine that could be used with a multiple selection List.†

Example 13-7 The sel_callback() routine for a multiple selection list

```
void sel_callback ( Widget list_w, XtPointer client_data,
                    XtPointer call_data)
{
    XmListCallbackStruct  *cbs = (XmListCallbackStruct *) call_data;
```

* The auto_selection_type field is only available from Motif 2.0 onwards.

† XmStringGetLtoR() is deprecated from Motif 2.0.

```
        char                *choice;
        int                 i;

        if (cbs->reason == XmCR_MULTIPLE_SELECT) {
            printf ("Multiple selection -- %d items selected:\n",
                            cbs->selected_item_count);

            for (i = 0; i < cbs->selected_item_count; i++) {
                choice = (char *) XmStringUnparse (cbs->selected_items[i],
                                                   XmFONTLIST_DEFAULT_TAG,
                                                   XmCHARSET_TEXT,
                                                   XmCHARSET_TEXT,NULL, 0,
                                                   XmOUTPUT_ALL);
                printf ("%s (%d)\n", choice, cbs->selected_item_positions[i]);
                XtFree (choice);
            }
        } else {
            choice = (char *) XmStringUnparse (cbs->item,
                                                   XmFONTLIST_DEFAULT_TAG,
                                                   XmCHARSET_TEXT,
                                                   XmCHARSET_TEXT,
                                                   NULL, 0,
                                                   XmOUTPUT_ALL);
            printf ("Default action -- selected item %s (%d)\n",
                                    choice, cbs->item_position);
            XtFree (choice);
        }
    }
```

The routine tests the callback structure's `reason` field to determine whether the callback was invoked as a result of a multiple selection action or the default action. When the `reason` is `XmCR_MULTIPLE_SELECT`, we print the list of selected items by looping through `selected_items` and `selected_item_positions`. With this reason, all of the fields in the callback structure except `selection_type` are valid. If the `reason` is `XmCR_DEFAULT_ACTION`, there is only one item selected, since the default selection action causes all of the other items to be deselected.

Extended Selection Callback

With the extended selection model, the user has the greatest flexibility to select and deselect individual items or ranges of items. The `XmNextendedSelectionCallback` is invoked whenever the user makes a selection or modifies the selection. Example 13-8 demonstrates the `sel_callback()` routine that could be used with an extended selection List.*

Example 13-8 The sel_callback() routine for extended selection

```
void sel_callback ( Widget list_w, XtPointer client_data,
                    XtPointer call_data)
```

* `XmStringGetLtoR`() is deprecated in Motif 2.0.

```
{
    XmListCallbackStruct  *cbs = (XmListCallbackStruct *) call_data;
    char                  *choice;
    int                   i;

    if (cbs->reason == XmCR_EXTENDED_SELECT) {
        if (cbs->selection_type == XmINITIAL)
            printf ("Extended selection -- initial selection: ");
        else if (cbs->selection_type == XmMODIFICATION)
            printf ("Extended selection -- modification of selection: ");
        else /* selection type = XmADDITION */
            printf ("Extended selection -- additional selection: ");

        printf ("%d items selected\n", cbs->selected_item_count);

        for (i = 0; i < cbs->selected_item_count; i++) {
            choice = (char *) XmStringUnparse (cbs->selected_items[i],
                                               XmFONTLIST_DEFAULT_TAG,
                                               XmCHARSET_TEXT,
                                               XmCHARSET_TEXT,
                                               NULL, 0,
                                               XmOUTPUT_ALL);
            printf ("%s (%d)\n", choice, cbs->selected_item_positions[i]);
            XtFree (choice);
        }
    } else {
        choice = (char *) XmStringUnparse (cbs->item,
                                           XmFONTLIST_DEFAULT_TAG,
                                           XmCHARSET_TEXT,
                                           XmCHARSET_TEXT,
                                           NULL, 0,
                                           XmOUTPUT_ALL);
        printf ("Default action -- selected item %s (%d)\n",
        choice, cbs->item_position);
        XtFree (choice);
    }
}
```

Most of the callback routine is the same as it was for multiple selection mode. With an extended selection callback, the `selection_type` field is also valid. This field can have the following values:

```
XmINITIAL          XmMODIFICATION         XmADDITION
```

The `XmINITIAL` value indicates that the selection is an initial selection for the List. All previously-selected items are deselected and the items selected with this action comprise the entire list of selected items. The value is `XmMODIFICATION` when the user modifies the selected list by using the SHIFT key in combination with a selection action. In this case, the selected item list contains some items that were already selected before this action took place. `XmADDITION` indicates that the items that are selected are in addition to what was previously selected. The user can select additional items by using the CTRL key in combination with a selection action. Regardless of the value for `selection_type`, the

selected_items and selected_item_positions fields always reflect the set of currently selected items.

Automatic Selection

A List which is configured for browse or extended selection does not normally invoke callbacks until the user completes the selection by releasing the mouse. This means that notification of selection change does not take place until after the event. If the resource XmNautomaticSelection is set to XmAUTO_SELECT, however, notification is immediate as soon as the user moves over a new item in the List. Automatic selection is disabled by setting XmNautomaticSelection to XmNO_AUTO_SELECT.* The *auto_selection_type* field in the XmListCallbackStruct indicates the state of the change to the selection. The field takes the following values:

XmAUTO_UNSET	XmAUTO_BEGIN	XmAUTO_MOTION
XmAUTO_NO_CHANGE	XmAUTO_CHANGE	XmAUTO_CANCEL

Summary

The List widget is a powerful user interface tool that has a simple design. The programming interface to the widget is mostly mechanical. The List allows you to present a vast list of choices to the user, although the choices themselves must be textual in nature. Lists are not suitable for all situations however, as they cannot display choices other than text (pixmaps cannot be used as selection items). Even with these shortcomings, the List widget is still a visible and intuitive object that can be used in designing a graphical user interface.

In Motif 1.2, individual List items could not be colored independently. With the advent of Render Tables in Motif 2.0, this is no longer the case. See Chapter 24, *Render Tables*, for more information on this subject. Example 24-1 is a sample application which creates a multi-colored ScrolledList.

Exercises

The following exercises expand on some of the concepts presented in this chapter.

1. Write a program that reads each word from the file */usr/dict/words* into a ScrolledList. Provide a TextField widget whose callback routine searches for the word typed into it from the entries in the List. Once found, make the List widget scroll so that each item is centered in the ScrolledList's viewport. (Hint: convert the C string from the Text-

* In Motif 1.2, XmNautomaticSelection is a simple Boolean resource. In Motif 2.0 and later, the resource changes to the enumerated type. For backwards compatibility, XmNO_AUTO_SELECT is equivalent to False, XmAUTO_SELECT to True.

Field into a compound string and use one of the List search routines to find the element.)

2. ScrolledLists frequently confuse the unsuspecting programmer who forgets that the parent of the List widget is a ScrolledWindow. For example, if you create a ScrolledList as a child of a Form widget, and want to specify attachment constraints on the ScrolledList, you should set these resources on the ScrolledWindow, not the List widget. Write a program that places two ScrolledList widgets next to each other in a single Form widget. (For more information on the role of the ScrolledWindow widget in a ScrolledList object, see the similar discussion on ScrolledText objects in Chapter 18, *Text Widgets*, and more discussion in Chapter 10, *ScrolledWindows and ScrollBars*.)
3. Consider two List widgets whose items are somewhat dependent on one another.For example, the one List contains login names and the other List contains the corresponding user-IDs.Write a program where the `XmNdefaultActionCallback` routine for each list selects the dependent/corresponding item in the other list.Since the user ID for “root” is always `0`,selecting “root” from the login name list should cause the item `0`in the user-ID list to be selected.

In this chapter:

- *Creating a ComboBox*
- *ComboBox Resources*
- *ComboBox Functions*
- *ComboBox Callbacks*
- *Summary*
- *Exercises*

14

The ComboBox Widget

A ComboBox is a component which combines direct textual entry with the convenience of list selection. The user can either type directly into a TextField, or choose from a set of predefined values available from a popup ScrolledList. The ComboBox widget was introduced in Motif 2.0, and is thus not available in earlier versions of the toolkit.

The Motif ComboBox is a manager widget. It automatically creates a TextField and ScrolledList for direct textual entry and list selection respectively. The ComboBox layout can be configured in a limited number of ways, depending upon whether the list of items is to be permanently visible, or whether the ScrolledList is hidden and only displayed on user request. Figure 14-1 displays a ComboBox with the List set to be permanently visible.

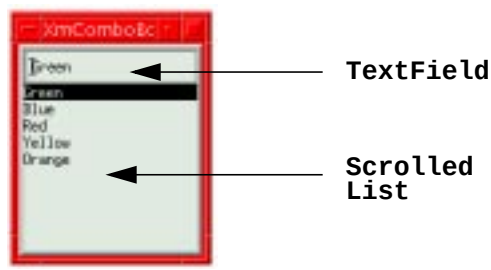


Figure 14-1 A ComboBox widget with the List permanently visible

In this configuration, the ScrolledList is simply placed directly underneath the ComboBox, and is sized so that the width of the List is the same as the ComboBox built-in TextField. The user can either type directly into the TextField, or select an item from the List. Selecting from the ScrolledList automatically places the selected item into the TextField, replacing any previous contents. This configuration is perhaps not the usual one associated with a ComboBox. More familiar is the arrangement shown in Figure 14-2. The ComboBox in this case does not display the Scrolled List until requested by the user. In order to display the List, the user clicks on an arrow which the ComboBox adds to the side of the TextField.

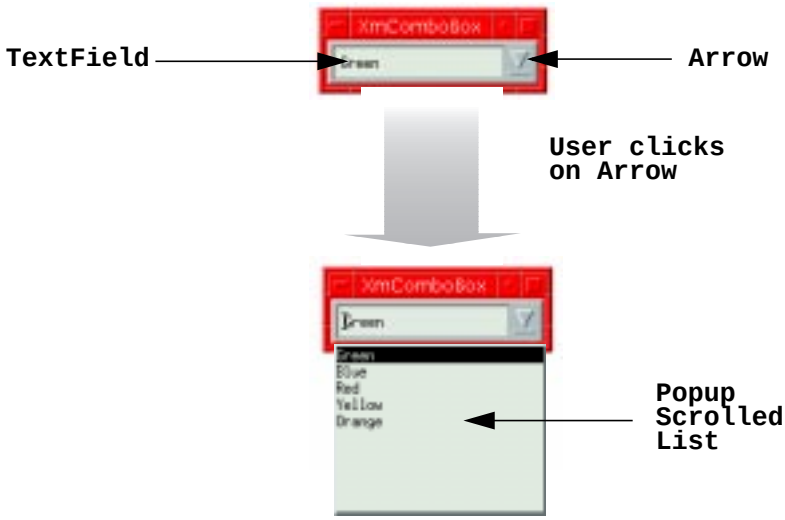


Figure 14-2 A ComboBox widget with the List visible on request

There are three basic kinds of ComboBox supported by Motif. The default ComboBox creates a permanently visible Scrolled List, and a TextField which is editable, which means that the user is not restricted in choice by the set of items in the List. A Drop-down-ComboBox, on the other hand, hides the Scrolled List until required, but again the TextField is editable. A ComboBox which is configured as a Drop-down-List also hides the built-in Scrolled List until required, but this time the TextField is not editable, and thus the user must choose from the Scrolled List in order to change the current selection.

The ComboBox is not meant to be used as a general purpose manager, and you are not expected to add any additional children to the widget over and above the built-in components.

Creating a ComboBox

Creating a ComboBox is performed in precisely the same kind of way that other Motif widgets are instantiated. Applications must include the header file associated with the widget class, which in this case is <*Xm/ComboBox.h*>. A ComboBox can be created in one of the following ways, either using a Motif convenience routine, or the general purpose Xt methods; note that the ComboBox supports more than one convenience routine for its creation:

```
Widget combo = XmCreateComboBox ( parent, "name", resource-value-array,
                                  resource-value-count);
...
Widget combo = XmCreateDropDownComboBox ( parent, "name",
                                  resource-value-array, resource-value-count);
```

```
...
Widget combo = XmCreateDropDownList ( parent, "name", resource-value-array,
                                      resource-value-count);
...
Widget combo = XtCreateWidget ( "name", xmComboBoxWidgetClass, parent,
                                resource-value-list, NULL);
```

The parent can be a Shell or any manager widget. Since the geometry management involved in creating a ComboBox is minimal, the widget could be created as managed using XtCreateManagedWidget() or similar without incurring a significant performance overhead. See Chapter 8, *Manager Widgets*, for a discussion of when manager widgets should be created in the managed or unmanaged state. The resource-value parameters control the behavior and visual effects of the ComboBox, as well as its built-in TextField and List children.

The most important resource for configuring the layout of the ComboBox is XmNcomboBoxType. If the value is XmCOMBO_BOX, the ComboBox creates a permanently visible Scrolled List, and the built-in TextField is set to be editable. If the value is XmDROP_DOWN_COMBO_BOX, the Scrolled List is hidden until required, an arrow is drawn by the ComboBox to facilitate List popup, and the TextField is set to be editable. The value XmDROP_DOWN_LIST is similar to XmDROP_DOWN_COMBO_BOX, except that the TextField is not editable, thus forcing the user to display and subsequently select from the Scrolled List in order to change the current selection. The XmNcomboBoxType resource is a create-only attribute: you cannot change the value dynamically, and must specify the type within the *resource-value-array* argument to the widget creation routine if you are using a general purpose Xt widget creator. The default value is XmCOMBO_BOX; this might be considered somewhat inconvenient since the usual requirement is for a ComboBox with a hidden List.

However, Motif does provide three convenience routines to create the widget, and these internally set the XmNcomboBoxType resource appropriately for the required configuration. The function XmCreateComboBox() is equivalent to setting XmNcomboBoxType to XmCOMBO_BOX, which displays the List permanently. For a hidden List, use either XmCreateDropDownComboBox() or XmCreateDropDownList(), depending on whether you need the TextField to be editable. XmCreateDropDownComboBox() is equivalent to setting XmNcomboBoxType to XmDROP_DOWN_COMBO_BOX, which creates an editable TextField. XmCreateDropDownList() creates a read-only TextField, and is equivalent to setting XmNcomboBoxType to XmDROP_DOWN_LIST.

Example 14-1 creates a ComboBox for selecting from a range of color names.*

* XtVaOpenApplication() and the SessionShell widget class are only available in X11R6. XmCreateDropDownList() is only available from Motif 2.0 onwards.

Example 14-1 The simple_combobox.c program

```
/* simple_combobox.c -- demonstrate the combobox widget */

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

/* the list of colors */
char *colors[] = { "red", "green", "blue", "orange", "purple",
                   "pink", "white", "black", "yellow" };

main (int argc, char *argv[])
{
    Widget           toplevel, combo;
    XtAppContext     app;
    Arg              args[4];
    int              count = XtNumber (colors);
    int              i, n;
    XmStringTable    str_list;

    /* initialize the toolkit */
    XtSetLanguageProc (NULL, NULL, NULL);
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv, NULL,
                                    sessionShellWidgetClass, NULL);

    /* create the List items */
    str_list = (XmStringTable) XtMalloc (count * sizeof (XmString *));

    for (i = 0; i < count; i++)
        str_list[i] = XmStringCreateLocalized (colors[i]);

    /* create the combobox */
    n = 0;
    XtSetArg (args[n], XmNitems, str_list);  n++;
    XtSetArg (args[n], XmNitemCount, count); n++;
    combo = XmCreateDropDownList (toplevel, "combo", args, n);

    for (i = 0; i < n; i++)
        XmStringFree (str_list[i]);
    XtFree ((XtPointer) str_list);

    XtManageChild (combo);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}
```

In the example, we create the ComboBox using the convenience routine XmCreateDropDownList(). If we wanted the user to be able to directly supply a different value to the colors we add to the ComboBox, we would use the routine XmCreateDropDownComboBox() instead. The appearance of the application is identical in this case; only the editability of the built-in TextField changes. We specify the contents of the ComboBox built-in List through the XmNitems and XmNitemCount resources. These are *mirrored* resources: the ComboBox arranges to set the items on the List on our

behalf, and thus we do not need to gain access to the underlying List component in order to program the set of available choices. The XmNitems resource specifies an array of compound strings, which topic is covered in detail in Chapter 25, *Compound Strings*.

The output from the program is given in Figure 14-3.

Figure 14-3 Output of the simple_combobox program

ComboBox Resources

The ComboBox provides what is known as a *mirror*; that is, for each of the important resources which can be set on the built-in TextField and Scrolled List children there is an equivalent implemented in the ComboBox resource table itself, so that access to the underlying children is not necessary for the majority of tasks.

List Resources

The contents of the ComboBox popup List can be set using the XmNitems and XmNitemCount resources of the ComboBox. XmNitems specifies an array of compound strings, and XmNitemCount is the number of such strings in the array. The following code fragment shows how to create a ComboBox which displays the names of the days of the week:

```
extern Widget parent;

char *month_names[] = { "Monday", "Tuesday", "Wednesday", "Thursday",
                        "Friday", "Saturday", "Sunday" };

int          count, i, n;
XmString     *xms;
Arg          args[4];
Widget       combo;

count = XtNumber (month_names);
xms = (XmString *) XtMalloc ((unsigned) count * sizeof (XmString));

for (i = 0; i < count; i++) {
    xms[i] = XmStringCreateLocalized (month_names[i]);
}
```

```
n = 0;
XtSetArg (args[n], XmNitems, xms);       n++;
XtSetArg (args[n], XmNitemCount, count); n++;
combo = XmCreateComboBox (parent, "days", args, n);

for (i = 0; i < count; i++)
    XmStringFree (xms[i]);
XtFree ((char *) xms);
```

The vertical size of the ComboBox List is controlled through the XmNvisibleItemCount resource. If the visible item count is less than the number of items in the list, the List automatically displays ScrollBars for navigating to hidden items.

Just as a List supports automatic keyboard selection through the XmNmatchBehavior resource, so this is mirrored in the ComboBox resource set. If the value is XmQUICK_NAVIGATE, the user can select an item in the List simply by typing the first character of the item. This behaviour is disabled if the value is XmNONE.

The widget ID of the built-in List is available through the ComboBox XmNlist resource. You may fetch, but not set the value of this resource.

TextField Resources

The TextField contains the current selection. This can be fetched or set through the XmNselectedItem resource. Note that this resource specifies a compound string, even though the internals of the Motif Text widgets are not compound-string based.

The width of the built-in TextField is controlled by the XmNcolumns resource. This also controls the width of the ComboBox itself, although allowance should be made for the additional space required to draw an arrow.

The widget ID of the built-in TextField can be fetched using the XmNtextField resource. This resource is read-only in nature.

ComboBox Resources

The type of the ComboBox is specified through the XmNcomboBoxType resource. The possible values are:

```
XmCOMBO_BOX          XmDROP_DOWN_COMBO_BOX          XmDROP_DOWN_LIST
```

Note that these are create-only resources, so you need to decide in advance whether or not to allow the user to type directly into the TextField.

The way in which the ComboBox reports the current selection is controlled through a resource called XmNpositionMode. Normally, a List is manipulated by specifying indexes starting at 1. That is to say, the first item in a List is at position 1, the second item at position 2, and so forth. Position zero is reserved to mean "the end of the list", whatever that index may be. However, the CDE DtComboBox widget was based around a different

model: the first item is at position zero, the second at position 1, and so forth. Since CDE pre-dates the release of Motif 2.x, applications may well have included the DtComboBox widget well before the Motif ComboBox became available. For this reason, the resource XmNpositionMode was implemented. If XmNpositionMode is XmONE_BASED, the ComboBox reports positions in its callbacks starting at position 1, which is consistent with normal Motif List behavior. If on the other hand, XmNpositionMode is XmZERO_BASED, positions are reported offset from zero for CDE compatibility. Note that this resource does not affect any of the ComboBox functions described within the next section: it only affects the way in which the various elements of a ComboBox XmNselectionCallback is interpreted, and the value of the resource XmNselectedPosition.

The index of the current selection in the List of choices is given by the XmNselectedPosition resource. As described in the previous paragraph, the interpretation of this resource depends upon the XmNpositionMode value. If XmNpositionMode is XmONE_BASED, an XmNselectedPosition of 1 means that the first item in the List is the selected item. In this mode, a position of zero is reserved to mean that no List items were selected. If XmNpositionMode is XmZERO_BASED, an XmNselectedPosition of zero means that the first List item is selected, a position of 1 means the second was selected, and so forth. It is difficult in these circumstances to work out if no List items were selected other than by directly querying the XmNselectedItems of the internal List itself.

ComboBox Functions

Just as you can program the contents of a List widget either by manipulating the XmNitems and XmNselectedItem resources or by calling any of a whole range of routines for selectively adding or deleting items, so also the ComboBox comes with a set of convenience routines for changing its List contents over and above any manipulations to the ComboBox XmNitems resource you may care to make.*

Adding ComboBox Items

An item can be inserted into the ComboBox built-in List through the function XmComboBoxAddItem(), which takes the following form:

```
void XmComboBoxAddItem ( Widget       combo,
                         XmString     item,
                         int          position,
                         Boolean      unique)
```

* With the exception of XmComboBoxUpdate(), all of the ComboBox convenience routines are only available as of Motif 2.1. XmComboBoxUpdate() is available from Motif 2.0.

The *position* parameter specifies an index into the ComboBox List where the *item* is to be inserted. The first item in the List is at position 1. If *position* is zero, *item* is appended to the bottom of the List. If *unique* is True, *item* is only inserted if it does not already exist in the ComboBox List.

Deleting ComboBox Items

You can delete an item at a given position in the ComboBox List through the function XmComboBoxDeletePos(). This routine has the following prototype:

```
void XmComboBoxDeletePos (Widget combo, int position)
```

Again, the *position* parameter is interpreted such that the first item in the ComboBox List is at position 1. Deleting position zero deletes the last ComboBox List item. An error message is displayed by the Motif toolkit if no item exists at the specified *position*.

Selecting ComboBox Items

You can programmatically set the current choice in the ComboBox through the routine XmComboBoxSelectItem(). This has the following signature:

```
void XmComboBoxSelectItem (Widget combo, XmString item)
```

The routine selects the first occurrence of *item* in the ComboBox internal List. This also resets the contents of the built-in TextField. There is a side effect such that if *item* is not amongst the ComboBox List choices, an error message is displayed by the Motif toolkit.

The routine XmComboBoxSetItem() is very similar to XmComboBoxSelectItem() in that it sets the current selection. It also makes the selected choice the first visible item in the ComboBox List. The routine has the following format:

```
void XmComboBoxSetItem (Widget combo, XmString item)
```

Whether using XmComboBoxSetItem() or XmComboBoxSelectItem(), no callbacks are invoked by the ComboBox as a result of changing the current selection.

Updating the ComboBox

If you change the state of the ComboBox built-in components other than through the convenience routines, it is possible for the ComboBox to get out of step with what it believes is the current contents and state of its children. For example, you might fetch the built-in List through the XmNlist resource of the ComboBox, and set various resources directly on the child rather than going through the ComboBox mirror. This might be performed because the set of ComboBox convenience routines for manipulating the contents of the internal List is considerably smaller than the set provided for directly manipulating a normal List, and so by fetching the widget ID of the List child a much greater range of toolkit functionality becomes available for use. In these circumstances, the

ComboBox may need to be synchronized with the state of its children after you have finished manipulating them. The routine XmComboBoxUpdate() is provided for this purpose, and the ComboBox resets its state by refreshing its values read directly from the children. This routine is specified as follows:

```
void XmComboBoxUpdate (Widget combo)
```

ComboBox Callbacks

The only callback defined specifically by the ComboBox widget class is the XmNselectionCallback. This is called when the user changes the current selection, either by typing into the built-in TextField, or by selecting from the built-in Scrolled List. Each callback of this type is associated with the structure XmComboBoxCallbackStruct, which is defined as follows:

```
typedef struct
{
    int          reason;
    XEvent       *event;
    XmString     item_or_text;
    int          item_position;
} XmComboBoxCallbackStruct;
```

The reason element of an XmNselectionCallback will have the value XmCR_SELECT. The item_or_text element is a compound string which represents the current selection. This is temporarily allocated only for the duration of the callback, and so if you need to cache the current choice, you need to copy the element using XmStringCopy() or similar.

The interpretation of the item_position element depends upon the value of the XmNpositionMode resource. If the mode is XmONE_BASED, an item_position of 1 refers to the first item in the ComboBox List, an item_position of 2 is the second List item, and so forth. An item_position of zero indicates that there was no List selection - in other words, the item_or_text element value must have resulted from the user directly typing into the ComboBox TextField. If the XmNpositionMode resource is XmZERO_BASED, an item_position of zero could either have resulted from direct text entry, or the first List item was selected. An item_position of 1 indicates selection of the second List item, and so forth. In this mode, it can be very difficult to distinguish between the cases where the user has selected the first item in the List, and when the user has directly typed, simply by inspection of the callback data, particularly when the user navigates the built-in List using the keyboard. In both cases, the item_position element is zero, and the event element will report a KeyEvent of some description. If you need to know the difference between List and TextField selection, you should consider switching to a position mode of XmONE_BASED.

Note that the XmNselectionCallback is called every time the selection changes, even if the user has not finished with the current action. For example, if the user browses up and

down the List of choices using the arrow keys, the XmNselectionCallback will be invoked each time the choice is moved, even though the user has not made the final selection. Fortunately it is possible to distinguish between browsing around the List and actual selection because the event element of the callback structure is NULL when the user browses the List, and points to a proper KeyPress or ButtonRelease event otherwise.

Example 14-2 is a simple program which prints out the current selection, using an XmNselectionCallback for the purpose.*

Example 14-2 The combo_cb.c program

```
/* combo_cb.c -- demonstrate the combobox widget selection callback */

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

/* the list of colors */
char *colors[] = { "red", "green", "blue", "orange", "purple",
                   "pink", "white", "black", "yellow" };

/* selection_callback: simply prints out the current ComboBox selection */
void selection_callback (Widget w, XtPointer client_data, XtPointer call_data)
{
    XmComboBoxCallbackStruct *cb = (XmComboBoxCallbackStruct *) call_data;

    char *choice = (char *) XmStringUnparse (cb->item_or_text,
                                             XmFONTLIST_DEFAULT_TAG,
                                             XmCHARSET_TEXT, XmCHARSET_TEXT,
                                             NULL, 0, XmOUTPUT_ALL);

    if (cb->event == NULL) {
        printf ("Browsing: potential choice is %s\n", choice);
    }
    else {
        printf ("New current choice is %s\n", choice);
    }

    XtFree (choice);
}

main (int argc, char *argv[])
{
    Widget           toplevel, combo;
    XtAppContext     app;
    Arg              args[4];
    int              count = XtNumber (colors);
    int              i, n;
    XmStringTable    str_list;
```

* XtVaOpenApplication() and the SessionShell widget class are only available in X11R6. XmCreateDropDownComboBox() and XmStringUnparse() are only available from Motif 2.0 onwards.

```
    /* initialize the toolkit */
    XtSetLanguageProc (NULL, NULL, NULL);
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv, NULL,
                                    sessionShellWidgetClass, NULL);

    /* create the List items */
    str_list = (XmStringTable) XtMalloc ((unsigned) count *
                                                 sizeof (XmString *));

    for (i = 0; i < count; i++)
        str_list[i] = XmStringCreateLocalized (colors[i]);

    /* create the combobox */
    n = 0;
    XtSetArg (args[n], XmNitems, str_list);  n++;
    XtSetArg (args[n], XmNitemCount, count); n++;
    combo = XmCreateDropDownComboBox (toplevel, "combo", args, n);

    for (i = 0; i < n; i++)
        XmStringFree (str_list[i]);
    XtFree ((XtPointer) str_list);

    XtAddCallback (combo, XmNselectionCallback, selection_callback, NULL);

    XtManageChild (combo);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}
```

Summary

The beauty of the ComboBox is the fact that it provides the convenience of List selection while only occupying the screen space of a TextField when inactive. The user normally sees just the current selection, and only needs to inspect the full set of values when the choice is to be changed. Changing the choice is simply a matter of displaying and selecting from the List, or typing the new value. It is the conjunction of economy of space with ease of use which makes this one of the single most important additions to the Motif 2.x widget set.

Exercises

1. Create a program which displays a date using a combination of three ComboBoxes for each of the day, month, and year elements. The year ComboBox should offer a simple range of choices for the years 1990 through to 2010.
2. Modify your program such that the current date is displayed in the ComboBoxes when the program first starts.

In this chapter:

- *Creating a SimpleSpinBox*
- *Creating a SpinBox*
- *SpinBox and SimpleSpinBox Resources*
- *SpinBox and SimpleSpinBox Callbacks*
- *Summary*
- *Exercises*

15

The SpinBox and SimpleSpinBox Widgets

This chapter describes two components which allow the user to chose from a group of values that occur in a predefined set: the SpinBox, and the SimpleSpinBox widgets*.

The SimpleSpinBox consists of a TextField, and two arrows. The programmer associates a set of values with the SimpleSpinBox; the TextField displays the current selection from the set, and the user can chose the next or previous value in the set by pressing on an arrow. Think of the widget as a kind of ComboBox which, when an arrow is pressed, automatically selects the next value for the user, rather than the user selecting from a popdown List of choices. Figure 15-1 displays a typical SimpleSpinBox. In this case, the widget is associated with the names of the months, and the user simply chooses the previous or next month by pressing the increment or decrement arrow.

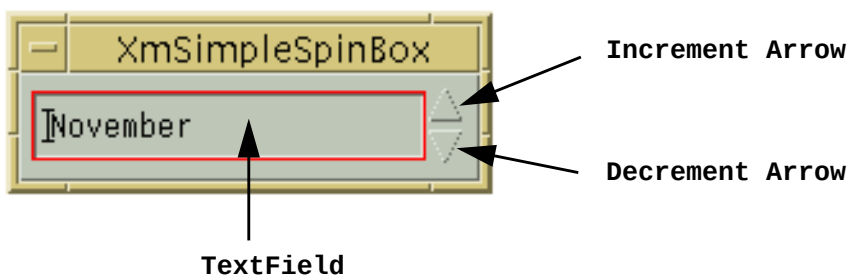


Figure 15-1 A SimpleSpinBox widget

The SpinBox is similar to the SimpleSpinBox: it provides two arrows for rotating the current selection. The difference is that the SpinBox is a general purpose manager: the programmer can add multiple TextFields to the widget, and the arrows simply manipulate the set of values associated with the current TextField child which has the focus. Figure 15-2 display a SpinBox which contains three TextField children, used in this instance for displaying the day, month, and year components of a date.

* The SpinBox is only available from Motif 2.0 onwards. The SimpleSpinBox is only available from Motif 2.1.

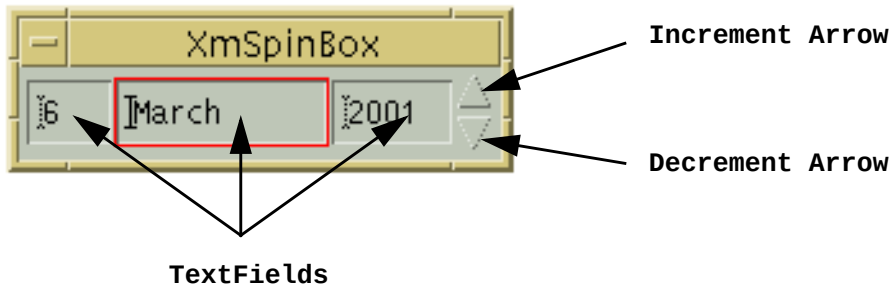


Figure 15-2 A SpinBox widget with three TextField children

SpinBoxes, and SimpleSpinBoxes, assume that their TextField children come in two different flavors: numeric, and string. A numeric SpinBox TextField allows the user to select from a range: the programmer specifies the lower and upper bounds of the range, as well as the interval between selections. Clearly, a TextField in a SpinBox displaying days of the month would have a lower bound of 1, and an upper bound variously of 28, 29, 30, or 31, The interval would be 1, because you would expect that pressing the increment or decrement arrow should change the value to an adjacent day. A string SpinBox TextField, on the other hand, simply has an ordered array of strings from which the user can sequentially select.

In either flavor, when the last value in the set of choices is reached, and the user presses the increment arrow, the new current value is the first value in the range: the choice wraps around to the start of the set. This wrapping behavior is also true when going in the other direction: pressing the decrement arrow when the first value is current causes the last value to become the current choice. If the programmer does not want this kind of wrapping behavior, it can be configured through resources.

The differences between the SpinBox and SimpleSpinBox widgets as far as the user is concerned are minimal: each presents the current choice to the user in the same way through a TextField or TextFields, and each allows the user to modify the choice through increment and decrement arrows automatically provided for the purpose.

As far as programming the two widgets is concerned, there are subtle differences.

Firstly, the SimpleSpinBox comes with a built-in TextField ready-prepared to display the current choice: the programmer only needs to specify the valid set of values through appropriate resources. The general purpose SpinBox manager, on the other hand, comes unprepared: the programmer must add all the necessary TextField children herself, as well as specifying the value ranges. Although derived from SpinBox, the SimpleSpinBox is not meant to be used as a general purpose manager, and you should not put additional children into the widget: use the SpinBox instead if you need to modify the structure in this way.

The second and most important difference is the fact that the SpinBox widget is a *constraint* manager pure and simple: the set of values associated with the various TextField children is specified by setting constraints on each. By contrast, the SimpleSpinBox provides *mirrored* resources for the built-in TextField child; you do not have to specify the set of values by setting constraints on the TextField: instead you set the same constraint resources defined by the SpinBox directly onto the SimpleSpinBox widget itself. The SimpleSpinBox thereafter arranges to apply the values to the built-in child on your behalf. In other words, the set of resources you can use to program a SpinBox and a SimpleSpinBox is the same in each case; what is different is where you apply them.

The notion of a set is important when considering using a SpinBox or SimpleSpinBox. If it is not obvious what the “next” or “previous” value means in a given context, then the SpinBox is probably not the correct widget to use. For example, if the set consists of days of the week, it is fairly intuitive what “next” means if the current value is Monday. On the other hand, if the set consists of arbitrary data such as a list of color names, it is not necessarily obvious what “next” means if the current value is Orange. Unless there is an imposed order on the set, for example, an alphabetical listing, then this usage of the SpinBox could well be entirely inappropriate and confusing to the user, who would not know whether to increment or decrement to find the required new current value. A simple List or ComboBox presentation should therefore be preferred if the data is unordered, or if the next or previous choice is not obvious given an arbitrary current selection.

In passing, note that the increment and decrement arrows are *drawn* by the SpinBoxes; they are not real widgets, and thus there is little which can be done to manipulate their appearance other than through resources which are directly provided by the SpinBoxes themselves.

Creating a SimpleSpinBox

Creating a SimpleSpinBox is performed in precisely the same kind of way that other Motif widgets are instantiated. Applications must include the header file associated with the widget class, which in this case is <*Xm/SSpinB.h*>. A SimpleSpinBox can be created in one of the following ways, either using a Motif convenience routine, or the general purpose Xt methods:

```
Widget sspin_b = XmCreateSimpleSpinBox (parent, “name”,
                                        resource-value-array, resource-value-count);
Widget sspin_b = XtCreateWidget ( "name", xmSimpleSpinBoxWidgetClass,
                                  parent, resource-value-list, NULL);
```

The `parent` can be a Shell or any manager widget. Since the geometry management involved in creating a SimpleSpinBox is minimal, involving as it does just a single TextField, the widget could be created managed using `XtCreateManagedWidget`() or similar without incurring a significant performance overhead. See Chapter 8, *Manager Widgets*, for a discussion of when manager widgets should be created in the managed or

unmanaged state. The resource-value parameters control the behavior and visual effects of the SimpleSpinBox, as well as its built-in TextField child. The most important resource is XmNspinBoxChildType, which specifies whether the widget is numerical or string-based in behavior. The value of the resource is either XmNUMERIC, or XmSTRING.

The Numerical SimpleSpinBox

Numerical SimpleSpinBoxes are created by specifying the XmNspinBoxChildType resource as XmNUMERIC*. Once we have made our SimpleSpinBox numerical, we need to specify the range of values from which the user can chose. We do this by supplying a lower bound, an upper bound, and an increment value. Example 15-1 shows how to create a numerical SimpleSpinBox which lets the user select an even number which falls between zero and twenty.

Example 15-1 The numeric_simplespin.c program

```
/* numeric_simplespin.c -- demonstrate the simple spin box widget */

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

main (int argc, char *argv[])
{
    Widget         toplevel, simple_b;
    XtAppContext   app;
    int            n;
    Arg            args[8];

    XtSetLanguageProc (NULL, NULL, NULL);

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

    n = 0;
    XtSetArg (args[n], XmNspinBoxChildType, XmNUMERIC); n++;
    XtSetArg (args[n], XmNminimumValue, 0);             n++;
    XtSetArg (args[n], XmNmaximumValue, 20);            n++;
    XtSetArg (args[n], XmNincrementValue, 2);           n++;
    XtSetArg (args[n], XmNeditable, TRUE);              n++;
    XtSetArg (args[n], XmNcolumns, 10);                 n++;
    XtSetArg (args[n], XmNwrap, FALSE);                 n++;
    simple_b = XmCreateSimpleSpinBox (toplevel, "simple", args, n);

    XtManageChild (simple_b);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}
```

* Note that this is really a SpinBox constraint resource, and we are in reality making the SimpleSpinBox built-in TextField numerical, rather than making the SimpleSpinBox numerical. We can apply it to the SimpleSpinBox because of the way it applies constraints on our behalf.

We specify the minimum bounds to the range of selectable values through the XmNminimumValue resource. The upper bound is given by the XmNmaximumValue resource, and the gap between selectable values is specified using the XmNincrementValue resource. For the sake of example, we also explicitly allow the user to directly type into the built-in TextField by setting the XmNeditable resource to True. We also explicitly specify that the SimpleSpinBox is not to wrap around to the start (or end) of the range of values when the user increments at the end of the range (or decrements at the beginning) by forcing the XmNwrap resource to False. The user will either have to type the value in directly, or cycle right through the set of values in the opposite direction. This may or may not be anti-social programming in any given circumstance: you should think carefully before turning either or both XmNeditable and XmNwrap behavior off. In the example, it makes some kind of sense to turn XmNwrap off so that the user is more aware of the upper bound; if XmNwrap is False, the bell is automatically rung if the user attempts to exceed the bounds of the data set.

The numeric SimpleSpinBox is not confined to situations where the data is integral in character: the XmNdecimalPoints resource can be specified so that the widget appears to handle real numbers. Supposing that we wanted to allow the user to chose from within the range 2.75 through to 3.45, incrementing by 0.05 on each click of the arrow. The following code fragment shows the way to do this: since there are two decimal places, we need to multiply all our bound resources by 10 to the power of two. This is consistent with the way that the XmNdecimalPoints resource behaves in the Scale widget.

```
n = 0;
XtSetArg (args[n], XmNspinBoxChildType, XmNUMERIC);    n++;
XtSetArg (args[n], XmNminimumValue, 275);              n++;
XtSetArg (args[n], XmNmaximumValue, 345);              n++;
XtSetArg (args[n], XmNdecimalPoints, 2);               n++;
XtSetArg (args[n], XmNincrementValue, 5);              n++;
XtSetArg (args[n], XmNpositionType, XmPOSITION_VALUE); n++;
XtSetArg (args[n], XmNposition, 275);                  n++;
...
simple_b = XmCreateSimpleSpinBox (toplevel, "simple", args, n);
...
```

The XmNpositionType and XmNposition resources are used to initialize the current value of the SimpleSpinBox. XmNposition specifies the current value of the SimpleSpinBox; XmNpositionType specifies whether the XmNposition resource is interpreted as an absolute value, or as an index into the set of values. They will be further discussed later in this chapter.

The String SimpleSpinBox

A string-based SimpleSpinBox is created by specifying the XmNspinBoxChildType resource as XmSTRING*. The set of data associated with the string SimpleSpinBox is defined by the XmNvalues and XmNnumValues resources. The XmNvalues resource is

internally represented by an array of compound strings; you are referred to Chapter 25, *Compound Strings*, if you require more details on the creation and manipulation of these objects.

Example 15-2 creates a string-based SimpleSpinBox, which allows the user to select from the names of the months.

Example 15-2 The string_simplespin.c program

```
/* string_simplespin.c -- demonstrate the simple spin box widget */

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

char *months[] = {   "January", "February", "March", "April", "May", "June",
                     "July", "August", "September", "October", "November",
                     "December" };

main (int argc, char *argv[])
{
    Widget           toplevel, simple_b;
    XtAppContext     app;
    int              i, n = XtNumber (months);
    Arg              args[8];
    XmStringTable    str_list;

    XtSetLanguageProc (NULL, NULL, NULL);

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

    str_list = (XmStringTable) XtMalloc (n * sizeof (XmString *));

    for (i = 0; i < n; i++)
        str_list[i] = XmStringCreateLocalized (months[i]);

    i = 0;
    XtSetArg (args[n], XmNspinBoxChildType, XmSTRING); i++;
    XtSetArg (args[i], XmNeditable, FALSE);            i++;
    XtSetArg (args[i], XmNnumValues, n);               i++;
    XtSetArg (args[i], XmNvalues, str_list);           i++;
    XtSetArg (args[i], XmNwrap, TRUE);                 i++;

    simple_b = XmCreateSimpleSpinBox (toplevel, "simple", args, i);

    for (i = 0; i < n; i++)
        XmStringFree (str_list[i]);
    XtFree ((XtPointer) str_list);

    XtManageChild (simple_b);
```

* Again, note that this is really a SpinBox constraint resource, and we are in reality making the SimpleSpinBox built-in TextField string-based, rather than making the SimpleSpinBox string-based.

```
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}
```

The output from the program is the same as that of Figure 15-1. We allocate an array of compound strings for use with the XmNvalues resource. We also specify XmNwrap to True, because the notion of a "next month" after December does make sense. We also set the XmNeditable resource False: allowing the user to directly type in an abbreviated or other style could give us some minor parsing problems, and we probably have better things to do with our time than writing verification callbacks for this.

The SimpleSpinBox Value

Now that we have created our SimpleSpinBox, we need to know how to explicitly set and fetch the current value. We could fetch the value using a simplistic algorithm: find the TextField which is built-in to the SimpleSpinBox, then retrieve the value using XmTextFieldGetString(). The name of the TextField is given by *spin*_TF, where *spin* is the name of the SimpleSpinBox parent. The following code fragment demonstrates this scheme:

```
char *FetchSimpleSpinValue (Widget simple_spin) {
    char   *pname = XtName (simple_spin);
    char   *tname = XtMalloc ((unsigned) strlen (pname) + 4);
    Widget textf;

    (void) sprintf (tname, "%s_TF", pname);

    textf = XtNameToWidget (simple_spin, tname);
    XtFree (tname);
    return XmTextFieldGetString (textf); /* Caller frees */
}
```

Whilst this would work, the method is somewhat heavy handed. Added to which, we would probably have to convert the fetched value into another form, particularly if the SimpleSpinBox is numeric in behavior. We most certainly could *not* use this kind of method for setting the value. The SimpleSpinBox internally keeps track of the selection through an index into the set of possible values, and directly writing the value into the TextField is not going to update the index correctly.

The correct way to fetch and store the current selection is to manipulate the XmNposition and XmNpositionType resources. XmNpositionType defines the way in which XmNposition is interpreted. If XmNpositionType is XmPOSITION_VALUE, the XmNposition resource represents an absolute value, bounded by the XmNmaximumValue and XmNminimumValue resource settings. If the position type is XmPOSITION_INDEX, then the XmNposition resource represents an index into the set of possible values. Clearly how we set the position type depends on the type of the SimpleSpinBox itself: if the widget is numeric, we probably want to fetch or set the current choice using an absolute value, but

if the widget is string-based, we would prefer an index into the array of compound strings stored behind the XmNvalue resource. XmNpositionType is a create only-resource; by default, it is XmPOSITION_VALUE. This needs not concern us if we are using a string SimpleSpinBox: since the default value of XmNposition is zero, we set and fetch the resource as though the value of XmNposition is indeed an index into the array of compound strings. Example 15-3 creates a pair of SimpleSpinBoxes, one string based, the other numeric, and a pair of PushButtons. Pressing a PushButton prints out the current value of a SimpleSpinBox, and increments the value as a side effect.

Example 15-3 The simplespin_value.c program

```
/* simplespin_value.c -- demonstrate the simple spin box widget */

#include <Xm/Xm.h>
#include <Xm/RowColumn.h>
#include <Xm/PushB.h>
#include <Xm/SSpinB.h>

char *months[] = { "Jan", "Feb", "Mar", "Apr", "May", "Jun",
                   "Jul", "Aug", "Sep", "Oct", "Nov", "Dec" };

/*
** Callback which prints out, then increments,
** the SimpleSpinBox value
*/
void increment_spinbox (Widget w, XtPointer client_data,
                        XtPointer call_data)
{
    Widget           spin = (Widget) client_data;
    unsigned char    type;
    int              position;

    XtVaGetValues (spin, XmNspinBoxChildType, &type,
                            XmNposition, &position, NULL);

    printf ("type: %s current position: %d\n",
                        (type == XmNUMERIC ? "numeric" : "string"),
                        position);

    if (type == XmNUMERIC) {
        int max;
        int min;

        XtVaGetValues (spin, XmNmaximumValue, &max,
                                        XmNminimumValue, &min, NULL);

        if (position == max) /* Wrap */
            position = min;
        else
            position++;

        XtVaSetValues (spin, XmNposition, position, NULL);
```

```
    }
    else {
        XmStringTable  values;
        int            count;

        XtVaGetValues (spin, XmNvalues, &values,
                                        XmNnumValues, &count, NULL);

        if (position == count - 1) /* Wrap */
            position = 0;
        else
            position++;

        XtVaSetValues (spin, XmNposition, position, NULL);
    }
}

main (int argc, char *argv[])
{
    Widget           toplevel, simple_b, push_b, rowcol, rc;
    XtAppContext     app;
    int              i, n = XtNumber (months);
    Arg              args[8];
    XmStringTable    str_list;

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

    rowcol = XmCreateRowColumn (toplevel, "rowcol", NULL, 0);

    i = 0;
    XtSetArg (args[i], XmNorientation, XmHORIZONTAL); i++;
    rc = XmCreateRowColumn (rowcol, "rowcol", args, i);

    /* Create a numeric SimpleSpinBox */
    i = 0;
    XtSetArg (args[i], XmNmaximumValue, 31);               i++;
    XtSetArg (args[i], XmNminimumValue, 1);                i++;
    XtSetArg (args[i], XmNincrementValue, 1);              i++;
    XtSetArg (args[i], XmNpositionType, XmPOSITION_VALUE); i++;
    XtSetArg (args[i], XmNposition, 1);                    i++;
    XtSetArg (args[i], XmNspinBoxChildType, XmNUMERIC);    i++;
    simple_b = XmCreateSimpleSpinBox (rc, "simple", args, i);
    XtManageChild (simple_b);

    push_b = XmCreatePushButton (rc, "Push me", NULL, 0);
    XtAddCallback (push_b, XmNactivateCallback, increment_spinbox,
                           (XtPointer) simple_b);
    XtManageChild (push_b);
    XtManageChild (rc);

    i = 0;
    XtSetArg (args[i], XmNorientation, XmHORIZONTAL); i++;
```

```
        rc = XmCreateRowColumn (rowcol, "rowcol", args, i);

        /* Create a string SimpleSpinBox */
        str_list = (XmStringTable) XtMalloc ((unsigned) n * sizeof (XmString *));

        for (i = 0; i < n; i++)
            str_list[i] = XmStringCreateLocalized (months[i]);

        i = 0;
        XtSetArg (args[i], XmNcolumns, 3);                      i++;
        XtSetArg (args[i], XmNeditable, FALSE);                 i++;
        XtSetArg (args[i], XmNnumValues, n);                    i++;
        XtSetArg (args[i], XmNvalues, str_list);                i++;
        XtSetArg (args[i], XmNwrap, TRUE);                      i++;
        XtSetArg (args[i], XmNpositionType, XmPOSITION_INDEX); i++;
        XtSetArg (args[i], XmNspinBoxChildType, XmSTRING);      i++;
        simple_b = XmCreateSimpleSpinBox (rc, "simple", args, i);
        XtManageChild (simple_b);

        for (i = 0; i < n; i++)
            XmStringFree (str_list[i]);
        XtFree((char *) str_list);

        push_b = XmCreatePushButton (rc, "Push me", NULL, 0);
        XtAddCallback (push_b, XmNactivateCallback, increment_spinbox,
                       (XtPointer) simple_b);
        XtManageChild (push_b);

        XtManageChild (rc);

        XtManageChild (rowcol);
        XtRealizeWidget (toplevel);
        XtAppMainLoop (app);
    }
```

The output of the program is given in Figure 15-3.

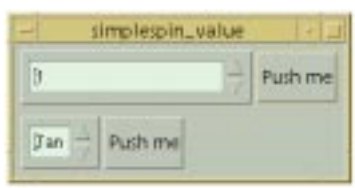


Figure 15-3 Output of the simplespin_values program

Creating a SpinBox

Applications must include the header file associated with the SpinBox widget class, which is <*Xm/SpinB.h*>. A SpinBox can be created in one of the following ways, either using a Motif convenience routine, or the general purpose Xt methods:

```
Widget sspin_b = XmCreateSpinBox ( parent, "name", resource-value-array,
                                   resource-value-count);
...
Widget sspin_b = XtCreateWidget ( "name", xmSpinBoxWidgetClass, parent,
                                  resource-value-list, NULL);
```

The parent can be a Shell or any manager widget. The widget could probably be created managed using XtCreateManagedWidget() or similar without incurring a significant performance overhead since the layout which the widget performs for its children is very simplistic: it simply lays them out in a line. See Chapter 8, *Manager Widgets*, for a discussion of when manager widgets should be created in the managed or unmanaged state.

The SpinBox widget is a *constraint* manager. Unlike the SimpleSpinBox which provides a mirror for resources which are applied to the built-in TextField child, the SpinBox is programmed by directly applying resources to each textual child, which must be individually added by the programmer. The SpinBox behavior will work equally well with either Text or TextField children; in the examples which follow, we use a TextField throughout, but this should not be taken as in any way necessary. Example 15-4 creates a SpinBox which displays the time.

Example 15-4 The date_spinbox.c program

```
/* date_spinbox.c -- demonstrate the spin box widget */
#include <Xm/Xm.h>
#include <Xm/RowColumn.h>
#include <Xm/SpinB.h>
#include <Xm/TextF.h>
#include <Xm/Label.h>
#include <sys/types.h>
#include <sys/time.h>

Widget hours_t, mins_t, ampm_t;

main (int argc, char *argv[])
{
    Widget           toplevel, spin, child;
    XtAppContext     app;
    XmStringTable    ampm;
    Arg              args[8];
    int              n;
    /* Initialize the spinbox to the current time */
    long tick = time ((long *) 0);
    struct tm *tm = localtime (&tick);
    /* 12 hour clock */
    int hours = ((tm->tm_hour > 12) ? (tm->tm_hour - 12) : tm->tm_hour);

    XtSetLanguageProc (NULL, NULL, NULL);

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

    /* Create the SpinBox */
```

```
    spin = XmCreateSpinBox (toplevel, "spin", NULL, 0);

    /* Create the Hours field */
    n = 0;
    XtSetArg (args[n], XmNspinBoxChildType, XmNUMERIC); n++;
    XtSetArg (args[n], XmNcolumns, 2);                  n++;
    XtSetArg (args[n], XmNeditable, FALSE);             n++;
    XtSetArg (args[n], XmNminimumValue, 1);             n++;
    XtSetArg (args[n], XmNmaximumValue, 12);          n++; /* 12 hour clock */
    XtSetArg (args[n], XmNposition, hours);             n++;
    XtSetArg (args[n], XmNwrap, TRUE);                  n++;
    hours_t = XmCreateTextField (spin, "hourText", args, n);
    XtManageChild (hours_t);

    /* Hours/Minutes separator */
    child = XmCreateLabel (spin, ":", NULL, 0);
    XtManageChild (child);

    /* Create the Minutes field */
    n = 0;
    XtSetArg (args[n], XmNspinBoxChildType, XmNUMERIC); n++;
    XtSetArg (args[n], XmNcolumns, 2);                  n++;
    XtSetArg (args[n], XmNeditable, FALSE);             n++;
    XtSetArg (args[n], XmNminimumValue, 0);             n++;
    XtSetArg (args[n], XmNmaximumValue, 59);            n++;
    XtSetArg (args[n], XmNwrap, TRUE);                  n++;
    XtSetArg (args[n], XmNposition, tm->tm_min);        n++;
    mins_t = XmCreateTextField (spin, "minuteText", args, n);
    XtManageChild (mins_t);

    /* Create the am/pm indicator field */
    ampm = (XmStringTable) XtMalloc(2 * sizeof (XmString *));
    ampm[0] = XmStringCreateLocalized ("am");
    ampm[1] = XmStringCreateLocalized ("pm");

    n = 0;
    XtSetArg (args[n], XmNspinBoxChildType, XmSTRING); n++;
    XtSetArg (args[n], XmNcolumns, 2);                 n++;
    XtSetArg (args[n], XmNeditable, FALSE);            n++;
    XtSetArg (args[n], XmNnumValues, 2);               n++;
    XtSetArg (args[n], XmNvalues, ampm);               n++;
    XtSetArg (args[n], XmNwrap, TRUE);                 n++;
    XtSetArg (args[n], XmNposition, (tm->tm_hour > 12) ? 1 : 0); n++;
    ampm_t = XmCreateTextField (spin, "ampmText", args, n);
    XtManageChild (ampm_t);
    XmStringFree (ampm[0]);
    XmStringFree (ampm[1]);
    XtFree ((char *) ampm);

    XtManageChild (spin);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}
```

The output of the date_spinbox.c program is given in Figure 15-4.

Figure 15-4 Output of the date_spinbox program

In the example, we create a SpinBox, and add four children: three TextFields to display the hours, minutes, and time-of-day, and a Label used to separate the hours and minutes fields. The SpinBox simply lays the children out horizontally in the order of creation. The widget is not sensitive to the XmNlayoutDirection resource in this respect: it is not possible to lay the children out vertically. The SpinBox itself is created very simply:

```
spin = XmCreateSpinBox (toplevel, "spin", NULL, 0);
```

The TextField to display the hours is programmed using SpinBox constraints: the XmNspinBoxChildType is set to XmNUMERIC, and the XmNmaximumValue and XmNminimumValue resources are set appropriately for a 12 hour clock. As a gloss, we initialize the value of the TextField from the current time, by using the XmNposition resource. The current time is fetched through the standard UNIX system calls time() and localtime(). You should refer to your system documentation if you are unfamiliar with these routines.

```
/* Initialize the spinbox to the current time */
long tick = time ((long *) 0);
struct tm *tm = localtime (&tick);
/* 12 hour clock */
int hours = ((tm->tm_hour > 12) ? (tm->tm_hour - 12) : tm->tm_hour);
...
n = 0;
XtSetArg (args[n], XmNspinBoxChildType, XmNUMERIC); n++;
XtSetArg (args[n], XmNcolumns, 2);                  n++;
XtSetArg (args[n], XmNeditable, FALSE);             n++;
XtSetArg (args[n], XmNminimumValue, 1);             n++;
XtSetArg (args[n], XmNmaximumValue, 12);            n++; /* 12 hour clock */
XtSetArg (args[n], XmNposition, hours);            n++; /* The current time */
XtSetArg (args[n], XmNwrap, TRUE);                  n++;
hours_t = XmCreateTextField (spin, "hourText", args, n);
XtManageChild (hours_t);
```

Although the SpinBox works by rotating the values of the TextField which currently has the input focus, the SpinBox will layout any widget class which is added as a child. In the example, we add a Label after the first TextField to act as a logical separator between the hours and minutes field.

```
/* Hours/Minutes separator */
child = XmCreateLabel (spin, ":", NULL, 0);
XtManageChild (child);
```

The TextField to display the minutes field is created in a similar fashion to the hours field; the XmNspinBoxChildType resource is set to XmNUMERIC, the XmNmaximumValue and XmNminimumValue constraint resources are set appropriately for the range of values to display, and the XmNposition resource is initialized to the current time.

The last TextField, used to display the morning or afternoon indicator, is created differently. In this case, the XmNspinBoxChildType constraint is set to XmSTRING, and the XmNvalues and XmNnumValues resources are used to specify an array of compound strings which contain the am/pm labels. Again, we initialize the TextField to display the current time using the XmNposition resource. Since the XmNspinBoxChildType is XmSTRING, the XmNposition resource in this case represents an index into the array of compound strings, rather than an absolute value.

```
/* Create the am/pm indicator field */
ampm = (XmStringTable) XtMalloc(2 * sizeof (XmString *));
ampm[0] = XmStringCreateLocalized ("am");
ampm[1] = XmStringCreateLocalized ("pm");

n = 0;
XtSetArg (args[n], XmNspinBoxChildType, XmSTRING); n++;
XtSetArg (args[n], XmNcolumns, 2);                 n++;
XtSetArg (args[n], XmNeditable, FALSE);            n++;
XtSetArg (args[n], XmNnumValues, 2);               n++;
XtSetArg (args[n], XmNvalues, ampm);               n++;
XtSetArg (args[n], XmNwrap, TRUE);                 n++;
/* Current time of day */
XtSetArg (args[n], XmNposition, (tm->tm_hour > 12) ? 1 : 0); n++;
ampm_t = XmCreateTextField (spin, “ampmText”, args, n);
XtManageChild (ampm_t);
```

Clearly, this arrangement of widgets could trivially be used to create a simple clock. We would only need to add a looping timeout handler, and adjust the XmNposition resources of each TextField appropriately as the timer expires. A program to create a clock using the SpinBox is listed amongst the Exercises section at the end of the chapter.

SpinBox and SimpleSpinBox Resources

The SpinBox, and SimpleSpinBox, resources for configuring the range of values associated with TextField children have already been mentioned in previous sections and examples of this chapter. In the paragraphs which follow, SpinBox is to include SimpleSpinBox unless otherwise stated. For a SimpleSpinBox, resources which are specified as constraints should be applied to the SimpleSpinBox itself, rather than on the SimpleSpinBox built-in TextField.

To recap, TextField children of SpinBoxes are considered to come in two flavors: numeric, and string.

A numeric TextField child of the SpinBox is configured by specifying the constraint resource XmNspinBoxChildType as XmNUMERIC. The range of possible values is programmed by specifying the XmNmaximumValue and XmNminimumValue constraints, and the current value is specified by the XmNposition constraint.

A string SpinBox TextField is configured by specifying the constraint resource XmNspinBoxChildType as XmSTRING. The values associated with the SpinBox is controlled through the XmNvalues and XmNnumValues constraints, which specify an array of compound strings. The current value is given by the XmNposition constraint, which represents an index into the array of compound strings.

In addition to specifying the range of values, it is also possible to control the way in which the user can activate the arrows for spinning through these values. The constraint resource XmNarrowSensitivity controls which of the arrows is currently sensitive to user input. The value XmARROWS_DECREMENT_SENSITIVE makes it possible for the user to only decrement the current value of the SpinBox. Similarly, XmARROWS_INCREMENT_SENSITIVE only allows an increase in SpinBox value. If neither action is currently allowed, the value XmARROWS_INSENSITIVE can be used to turn off spinning altogether. The default value, XmARROWS_SENSITIVE, allows both increase and decrease actions. In addition, a SpinBox, but not a SimpleSpinBox, supports the resource XmNdefaultArrowSensitivity, which specifies a default layout for TextField children which do not have the XmNarrowSensitivity constraint explicitly set. XmNdefaultArrowSensitivity can hold the same values as the XmNarrowLayout resource, and as a normal non-constraint resource should be applied to the SpinBox and not the TextField children.

The SpinBox supports the notion of automatic spin. That is, if the user presses an arrow, and holds the mouse button down, the SpinBox will automatically rotate through the values as though the user clicked multiple times. Automatic spin is configured through the XmNinitialDelay and XmNrepeatDelay resources. XmNinitialDelay specifies a time interval which is to elapse since the mouse button was first held down before automatic spinning becomes operative. XmNrepeatDelay specifies the time interval between successive spins of the value. If XmNrepeatDelay is zero, automatic spinning is disabled.

The location of the arrows relative to the SpinBox TextFields can be configured through the XmNarrowLayout resource. The arrows can be placed either horizontally or vertically aligned, both before and after the TextField children. Possible values for the XmNarrowLayout resource are:

```
XmARROWS_BEGINNING          XmARROWS_END
XmARROWS_FLAT_BEGINNING     XmARROWS_FLAT_END       XmARROWS_SPLIT
```

Figure 15-5 shows the effect of the various XmNarrowLayout values.

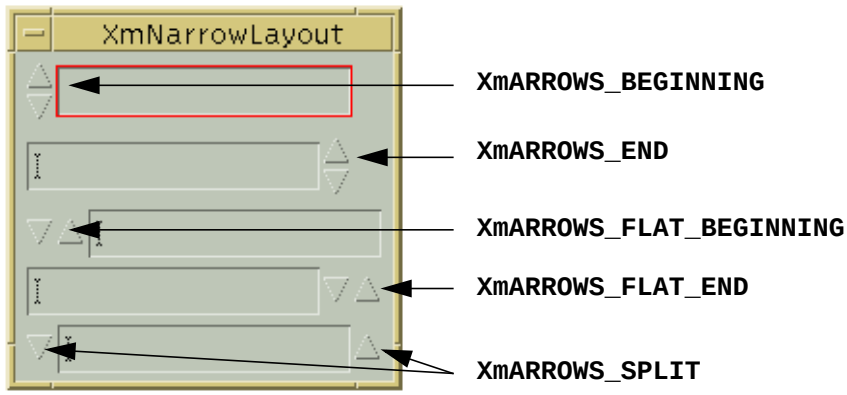


Figure 15-5 The various settings of the XmNarrowLayout resource

SpinBox and SimpleSpinBox Callbacks

The callback routines associated with the SpinBox and SimpleSpinBox widgets give the programmer control both over the value to display, and whether to allow spinning of the value, in any given instant. The callback model for the SpinBox widgets is analogous to those for the Text and TextField widgets, which is not surprising since the SpinBoxes simply manage the contents of child Text components.

The `XmNmodifyVerifyCallback` of the SpinBox can be used to verify the contents of a SpinBox TextField child after the user has requested a new selection, but before the actual contents are changed in the TextField itself. In the callback, the programmer can accept, reject, or otherwise alter the new proposed choice in any way which she feels fit by modifying suitable elements in the structure passed to the callback.

The `XmNvalueChangedCallback` is called after any `XmNmodifyVerifyCallback` has completed, provided that the `XmNmodifyVerifyCallback` did not reject the new SpinBox TextField value.

These two callbacks are typically used in conjunction; the `XmNmodifyVerifyCallback` polices the validity of the user selection, and the `XmNvalueChangedCallback` blindly processes the new value, safe in the knowledge that the value has been validated.

Both `XmNmodifyVerifyCallback` and `XmNvalueChangedCallback` functions are passed a structure in the following form:

```
typedef struct
{
    int         reason;
    XEvent      *event;
    Widget      widget;
```

```
    Boolean      doit;
    int          position;
    XmString     value;
    Boolean      crossed_boundary;
} XmSpinBoxCallbackStruct;
```

The `reason` field specifies the action performed by the user. Table 15-1 lists the callback name, reason, and associated user action for all values of the `reason` field:

Table 15-1 Callback resources for the SpinBox Widget

Resource Name	Reason	Action
`XmNmodifyVerifyCallback`, `XmNvalueChangedCallback`	`XmCR_SPIN_FIRST`	The SpinBox is at the lower end of the range of values
`XmNmodifyVerifyCallback`, `XmNvalueChangedCallback`	`XmCR_SPIN_LAST`	The SpinBox is at the upper end of the range of values
`XmNmodifyVerifyCallback`, `XmNvalueChangedCallback`	`XmCR_SPIN_PRIOR`	The user has armed the decrement arrow
`XmNmodifyVerifyCallback`, `XmNvalueChangedCallback`	`XmCR_SPIN_NEXT`	The user has armed the increment arrow
`XmNvalueChangedCallback`	`XmCR_OK`	The user has changed the value

The `widget` element is set to whichever Text or TextField child of the SpinBox currently has the input focus.

The `doit` element is the key to performing validation of the user's action. If the programmer sets this field to `False`, the SpinBox action is cancelled, and no change takes place to the current selection.

The `position` element is the callback equivalent of the `XmNposition` resource. The SpinBox sets the element to what it believes should be the newly displayed value before calling any `XmNmodifyVerifyCallback`: the programmer can subsequently alter the element inside the callback if an alternative position is required.

The `value` field contains a compound string representation of the newly selected value. If the SpinBox TextField which is being modified is numeric in nature, the `value` field contains the position of the SpinBox converted into a temporary compound string allocated only for the duration of the callback. The conversion process takes into account the `XmNdecimalPoints` resource of the changed TextField. On the other hand, if the SpinBox TextField is string-based, then the `value` field is not allocated - it merely points directly into the array of compound strings associated with the `XmNvalue` resource. In either case, if the programmer wishes to store the `value` field for use outside the callback, she must allocate a separate copy. The `value` field should not be freed by the programmer under any circumstances.

The `crossed_boundary` field indicates whether the user action wraps around the set of values associated with the current TextField. For example, if the user in a numeric TextField selects the decrement action and the currently displayed position is equal to

XmNminimum, or if in a string-based TextField the user selects the increment action and the currently displayed choice is the last item in the XmNvalues array.

Example 15-5 is a program which utilizes both the XmNmodifyVerifyCallback and this XmNvalueChangedCallback resources. It creates a SpinBox with three TextFields for day, month, year fields. The XmNmodifyVerifyCallback ensures that the date displayed is valid. For example, if February is the current month then the day field cannot exceed 28 or 29, depending on whether the year field represents a leap year. The XmNvalueChangedCallback simply prints out the new current date as it changes.

Example 15-5 The date_spinbox_cb.c program

```
/* date_spinbox_cb.c -- demonstrate the spin box widget callbacks */

#include <Xm/Xm.h>
#include <Xm/RowColumn.h>
#include <Xm/SpinB.h>
#include <Xm/TextF.h>
#include <Xm/Label.h>

Widget day_t, month_t, year_t;

char *months[] = {   "January", "February", "March", "April", "May", "June",
                     "July", "August", "September", "October", "November",
                     "December" };

int month_days[2][12] =  {
                    { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 },
                    { 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
                        };

/* check_days: the XmNmodifyVerifyCallback for the SpinBox */
void check_days (Widget w, XtPointer client_data, XtPointer call_data)
{
    XmSpinBoxCallbackStruct *sb = (XmSpinBoxCallbackStruct *) call_data;
    int year;
    int month;
    int day;
    int leap;

    if (sb->widget == day_t) {
        /* Make sure that the new day never exceeds the maximum
        ** for the current month
        */
    XtVaGetValues (year_t, XmNposition, &year, 0);

    leap = (((year % 4) == 0) ? ((year % 400) == 0 ? 1 : 0) : 0);

    XtVaGetValues (month_t, XmNposition, &month, 0);

    if (sb->position > month_days[leap][month]) {
        if (sb->crossed_boundary)
            /* Going backwards */
```

```
            sb->position = month_days[leap][month];
        else
            /* Going forwards */
            sb->position = 1;
        }
    }
    else {
        /* The month or year has changed.
        ** Recheck the day field to ensure it does not exceed the
        ** maximum for the new month or year.
        */
        if (sb->widget == month_t) {
            month = sb->position;

            XtVaGetValues (year_t, XmNposition, &year, 0);
        }
        else {
            year = sb->position;

            XtVaGetValues (month_t, XmNposition, &month, 0);
        }

        leap = (((year % 4) == 0) ? ((year % 400) == 0 ? 1 : 0) : 0);

        XtVaGetValues (day_t, XmNposition, &day, 0);

        if (day > month_days[leap][month]) {
            XtVaSetValues (day_t, XmNposition, month_days[leap][month], NULL);
        }
    }
}

/* print_date: the XmNvalueChangedCallback for the SpinBox */
void print_date (Widget w, XtPointer client_data, XtPointer call_data)
{
    XmSpinBoxCallbackStruct *sb = (XmSpinBoxCallbackStruct *) call_data;
    int day;
    int month;
    int year;

    if (sb->reason == XmCR_OK) {
        XtVaGetValues (day_t, XmNposition, &day, 0);
        XtVaGetValues (month_t, XmNposition, &month, 0);
        XtVaGetValues (year_t, XmNposition, &year, 0);

        printf ("New date: %d/%s/%d\n", day, months[month], year);
    }
}

main (int argc, char *argv[])
{
    Widget          toplevel, spin;
    XtAppContext    app;
    XmStringTable   ampm;
```

```
Arg             args[8];
int             i, n;
XmStringTable   str_list;

XtSetLanguageProc (NULL, NULL, NULL);

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

/* Create the SpinBox */
spin = XmCreateSpinBox (toplevel, "spin", NULL, 0);

/* Create the Day field */
n = 0;
XtSetArg (args[n], XmNspinBoxChildType, XmNUMERIC); n++;
XtSetArg (args[n], XmNcolumns, 2);                  n++;
XtSetArg (args[n], XmNeditable, FALSE);             n++;
XtSetArg (args[n], XmNminimumValue, 1);             n++;
XtSetArg (args[n], XmNmaximumValue, 31);            n++;
XtSetArg (args[n], XmNposition, 1);                 n++;
XtSetArg (args[n], XmNwrap, TRUE);                  n++;
day_t = XmCreateTextField (spin, "dayText", args, n);
XtManageChild (day_t);

/* Create the Month field */
n = XtNumber (months);
str_list = (XmStringTable) XtMalloc (n * sizeof (XmString *));

for (i = 0; i < n; i++)
    str_list[i] = XmStringCreateLocalized (months[i]);

n = 0;
XtSetArg (args[n], XmNspinBoxChildType, XmSTRING);   n++;
XtSetArg (args[n], XmNeditable, FALSE);              n++;
XtSetArg (args[n], XmNcolumns, 10);                  n++;
XtSetArg (args[n], XmNwrap, TRUE);                   n++;
XtSetArg (args[n], XmNvalues, str_list);             n++;
XtSetArg (args[n], XmNnumValues, XtNumber (months)); n++;
month_t = XmCreateTextField (spin, "monthText", args, n);
XtManageChild (month_t);

for (i = 0; i < XtNumber (months); i++)
    XmStringFree (str_list[i]);
XtFree((XtPointer) str_list);

/* Create the Year field */

n = 0;
XtSetArg (args[n], XmNspinBoxChildType, XmNUMERIC); n++;
XtSetArg (args[n], XmNcolumns, 4);                  n++;
XtSetArg (args[n], XmNeditable, FALSE);             n++;
XtSetArg (args[n], XmNminimumValue, 1990);          n++;
XtSetArg (args[n], XmNmaximumValue, 2010);          n++;
XtSetArg (args[n], XmNposition, 2000);              n++;
```

```
        XtSetArg (args[n], XmNwrap, TRUE);                  n++;
        year_t = XmCreateTextField (spin, "yearText", args, n);
        XtManageChild (year_t);

        XtManageChild (spin);
        XtAddCallback (spin, XmNmodifyVerifyCallback, check_days, NULL);
        XtAddCallback (spin, XmNvalueChangedCallback, print_date, NULL);

        XtRealizeWidget (toplevel);
        XtAppMainLoop (app);
    }
```

The output of the program is similar to Figure 15-2. The `XmNmodifyVerifyCallback check_days` is straightforward: if a field changes, we variously fetch the day, month and year field positions to ensure that the day number does not exceed the maximum allowed for the given date. If it is the day field that has changed, we only need to reset the `position` element of the callback structure to the new valid day, and we know if a day is invalid simply by comparing the callback `position` against the `month_days` array. Otherwise, if the month or year has changed, we must perform an explicit `XtVaSetValues`() call on the day_t field if we need to change the value. The `XmNvalueChangedCallback` is also straight forwards: format and print out the value of all three fields, but only if this callback is being called for the right reason and we know that any prior verification process has succeeded. This is the case if the reason field of the callback structure is set to `XmCR_OK`.

Summary

The SpinBox and SimpleSpinBox widgets can be used whenever there is a choice from a well-defined set of items. They are the natural choice of interface component if the data is in an ordered sequence. The next or previous item in the sequence of choices should be intuitive and obvious to the user at all times. Whenever the data meets these requirements, the SpinBox widgets provide a simple, compact, and natural method of selection. Remember that the user only sees the current value: if the next or previous selection is not entirely obvious, then the SpinBox or SimpleSpinBox is probably the wrong choice of interface component to present the choices to the user in the first place.

When to use a SpinBox, and when a SimpleSpinBox, can be a tricky choice. We could just as easily present the day/month/year components of a date through three SimpleSpinBoxes rather than using a SpinBox with three TextFields. Each component of the date would have individually associated increment and decrement arrows, which the user may, or may not, find more convenient than making sure the right SpinBox TextField child has the focus when using the arrows. On the other hand, the SpinBox manager method is slightly more compact, using a single set of arrows for all of the TextField children. It may well be that in any given situation, the choice of a SpinBox or multiple SimpleSpinBoxes makes little difference in terms of ease of use.

We recommend the SpinBox should be used in preference when you need to emphasize the fact that the components of the value are to be considered as parts of a single entity. This may well be the deciding factor: whether the data is, when considered logically, a single unit. In the example of a date, this is likely to be manipulated and stored as one, rather than each component being separated out and used in differing calculations, and thus here the SpinBox method is probably preferable to multiple SimpleSpinBoxes. Probably is an important word: we would not like to put hard and fast rules on when to use a SpinBox, and when to use multiple SimpleSpinBoxes; this is a judgement call the interface designer is going to have to make for herself, involving as it does all kinds of application-specific and human-computer interaction considerations.

Exercises

1. Create a simple working clock using the SpinBox widget. The clock should be in 12-hour format, and should display hours, minutes, seconds, and the time-of-day indicator appropriately as the time changes. Hint: you will need to use the function `XtAppAddTimeOut`() to effect the ticking of the clock.
2. Modify the `date_spinbox_cb` program so that if the user wraps round the day field, the month changes automatically, and similarly if the months field wraps round, the year changes. For example, if the current date is 31 December 1999, incrementing the day field should display 1 January 2000, and decrementing the day thereafter should revert to the original 31 December 1999 date.

In this chapter:

- *Creating a Scale Widget*
- *Scale Values*
- *Scale Orientation and Movement*
- *Scale Resources*
- *Scale Callbacks*
- *Scale Tick Marks*
- *Summary*

16

The Scale Widget

This chapter describes how to use the Scale widget to represent a range of values. The widget can be manipulated to change the value.

The Scale widget displays a numeric value that falls within upper and lower bounds. The widget allows the user to change that value interactively using a *slider* mechanism similar to that of a ScrollBar. This style of interface is useful when it is inconvenient or inappropriate to have the user change a value using the keyboard. The widget is also extremely intuitive to use; inexperienced users often understand how a Scale works when they first see one. Figure 16-1 shows how Scale widgets can be used with other widgets in an application.

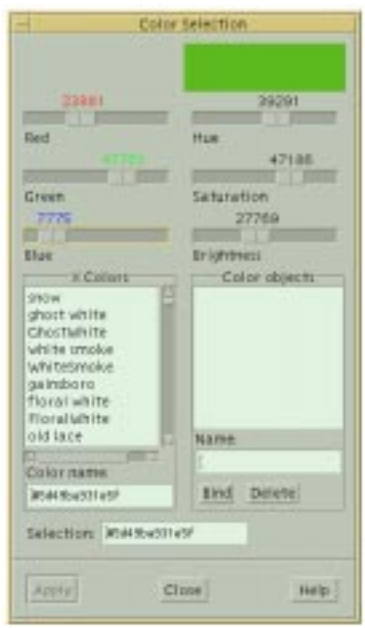


Figure 16-1 Scale widgets in an application dialog

A Scale can be oriented either horizontally or vertically. The values given to a Scale are stored as integers, but decimal representation of values is possible through the use of a resource that allows you to place a decimal point in the value. A Scale can be put in output-only mode, in which it is sometimes called a *gauge*. When a Scale is read-only, it implies that the value is controlled by another widget or that it is being used to report status information specific to the application. In Motif 1.2, the standard way to create a read-only Scale is to specify that it is insensitive. Unfortunately, this technique has the side-effect of greying out the widget. In Motif 2.0 and later, the widget supports the `XmNeditable` resource: setting this to `False` effects the required gauge behavior without the described side-effect.

Creating a Scale Widget

Applications that use the Scale widget must include the header file <*Xm/Scale.h*>. You can then create a Scale widget as follows:

```
Widget scale = XtVaCreateWidget ( "name", xmScaleWidgetClass, parent,
                                  resource-value-list, NULL);
Widget scale = XmCreateScale ( parent, "name", resource-value-array,
                               resource-value-count);
```

Even though the Scale widget functions as a primitive widget, it is actually subclassed from the Manager widget. All the parts of a Scale are really other primitive widgets, but these subwidgets are not directly accessible through the Motif toolkit. The fact that the Scale is a Manager widget means that you can create widgets that are children of a Scale. The children are arranged so that they are evenly distributed along the vertical or horizontal axis parallel to the slider, depending on the orientation of the Scale. In Motif 1.2, this technique is used primarily to provide "tick marks" for the Scale. In Motif 2.0 and later, tick marks can be added automatically through the function `XmScaleSetTicks`(), which will be described later in the chapter. In all other respects, a Scale can be treated just like other primitive widgets. Example 16-1 shows a program that creates some Scale widgets.*

Example 16-1 The simple_scale.c program

```
/* simple_scale.c -- demonstrate a few scale widgets. */

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

Widget create_scale (Widget parent, char *title,
int max, int min, int value)
{
    Arg       args[8];
    int       n = 0;
    XmString  xms = XmStringCreateLocalized (title);
```

* `XtVaAppInitialize()` is considered deprecated in X11R6.

```
    void      new_value(); /* callback for Scale widgets */
    Widget    scale;

    XtSetArg (args[n], XmNtitleString, xms); n++;
    XtSetArg (args[n], XmNmaximum, max);     n++;
    XtSetArg (args[n], XmNminimum, min);     n++;
    XtSetArg (args[n], XmNvalue, value);     n++;
    XtSetArg (args[n], XmNshowValue, True);  n++;

    scale = XmCreateScale (parent, title, args, n);
    XtAddCallback (scale, XmNvalueChangedCallback, new_value, NULL);
    XtManageChild (scale);

    return scale;
}

main (int argc, char *argv[])
{
    Widget         toplevel, rowcol, scale;
    Arg            args[2];
    XtAppContext   app;

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

    XtSetArg (args[0], XmNorientation, XmHORIZONTAL);
    rowcol = XmCreateRowColumn (toplevel, "rowcol", args, 1);

    scale = create_scale (rowcol, "Days", 7, 1, 1);
    scale = create_scale (rowcol, "Weeks", 52, 1, 1);
    scale = create_scale (rowcol, "Months", 12, 1, 1);
    scale = create_scale (rowcol, "Years", 20, 1, 1);

    XtManageChild (rowcol);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

void new_value (Widget scale_w, XtPointer client_data, XtPointer call_data)
{
    XmScaleCallbackStruct *cbs = (XmScaleCallbackStruct *) call_data;

    printf("%s: %d\n", XtName (scale_w), cbs->value);
}
```

The output of this program is shown in Figure 16-2.

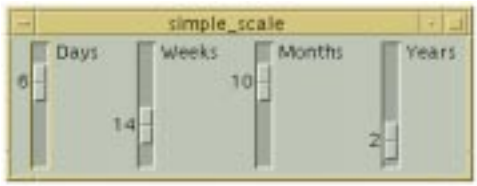


Figure 16-2 Output of the simple_scale program

The four Scales represent the number of days, weeks, months, and years, respectively. Each Scale displays a title that is specified by the `XmNtitleString` resource. Just as with other Motif widgets that display strings, the `XmNtitleString` must be set as a compound string, not a normal C string. Conversion between C strings and compound strings is described in detail in Chapter 25, *Compound Strings*.

A Scale cannot have a pixmap as its label. Since real estate for the label is limited in a Scale widget, you should take care to use small strings. If you need to use a longer string, you should include a separator so that the text is printed on two lines. If the string is too long, the label may be too wide and look awkward as a result. For a horizontal Scale, the label is displayed beneath the slider, while for a vertical Scale it is shown to the side of the slider.

The maximum and minimum values are set with the `XmNmaximum` and `XmNminimum` resources, respectively. The minimum values are set to `1` for the user's benefit; the minimum value of a Scale defaults to `0`. Note that if you set a minimum value other than `0`, you must also provide a default value for `XmNvalue` that is at least as large as the value of `XmNminimum`, as we have done in our example. Each Scale displays its current value because the `XmNshowValue` resource is set to `True`.

Scale Values

The value of a Scale can only be stored as an integer. This restriction is largely based on the fact that variables of type `float` and double cannot trivially be passed through `XtVaSetValues()`, `XtVaGetValues()`, or any of the widget creation functions.* If you need to represent fractional values, you must use the `XmNdecimalPoints` resource. This resource specifies the number of places to move the decimal point to the left in the displayed value, which gives the user the impression that the value displayed is fractional.

For example, a Scale widget used to display the value of a barometer might range from 29 to 31, with a granularity of 1-100th. The necessary widget could be created as shown in the following code fragment:

* While the Xt functions mentioned do allow the passing of the address of a variable of type `float` or `double`, the Scale widget does not support this type of value representation.

```
Widget   scale;
Arg      prgs[...];
int      n;
...
XtSetArg (args[n], XmNmaximum, 3100);    n++;
XtSetArg (args[n], XmNminimum, 2900);    n++;
XtSetArg (args[n], XmNdecimalPoints, 2); n++;
XtSetArg (args[n], XmNvalue, 3000);      n++;
XtSetArg (args[n], XmNshowValue, True);  n++;
scale = XmCreateScale (parent, "barometer",args, n);
```

The value for XmNdecimalPoints is 2, so that the value displayed is 30.00, rather than 3000. If you are using a Scale to represent fractional values, it is probably a good idea to set XmNshowValue to True since fine tuning is probably necessary.

There is no limit to the values that can be specified for the XmNmaximum, XmNvalue, and XmNminimum resources, provided they can be represented by the int type, which includes negative numbers. In the previous example, the initial value of the Scale (XmNvalue) is set arbitrarily; the value must be set within the minimum and maximum values. If the value of the Scale is retrieved using XtVaGetValues() or through a callback routine, the integer value is returned. To get the appropriate decimal value, you need to divide the value by 10 to the power of the value of XmNdecimalPoints. For example, since XmNdecimalPoints is 2, the value needs to be divided by 10 to the power of 2, or 100.

The value of a Scale can be set and retrieved using XtVaSetValues() and XtVaGetValues() on the XmNvalue resource. Motif also provides the functions XmScaleSetValue() and XmScaleGetValue() to serve the same purpose. These functions take the following form:

```
void XmScaleSetValue (Widget scale_w, int value)
void XmScaleGetValue (Widget scale_w, int *value)
```

The advantage of using the Motif convenience routines, rather than the Xt routines, is that the Motif routines manipulate data in the widget directly, rather than using the set and get methods of the Scale. As a result, there is less overhead involved, although the added overhead of the Xt methods are negligible.

Scale Orientation and Movement

A Scale can be either vertical or horizontal and the maximum and minimum values can be on either end of the Scale. By default, as shown in the examples so far, the Scale is oriented vertically with the maximum on the top and the minimum on the bottom. The XmNorientation resource can be set to XmHORIZONTAL to produce a horizontal Scale. The XmNprocessingDirection resource controls the location of the maximum and minimum values. The possible values for the resource are:

```
XmMAX_ON_TOP                XmMAX_ON_BOTTOM
XmMAX_ON_LEFT               XmMAX_ON_RIGHT
```

Unfortunately, you cannot set the processing direction unless you know the orientation of the Scale, so if you hard-code one resource, you should set both of them. If the Scale is oriented vertically, the default value is XmMAX_ON_TOP, but if it is horizontal, the default depends on the value of XmNlayoutDirection*. If you use a font that is read from right to left, then the maximum value is displayed on the left rather than on the right.

As the user drags the slider, the value of the Scale changes incrementally in the direction of the movement. If the user clicks the middle mouse button inside the Scale widget, but not on the slider itself, the slider moves to the location of the click. Unfortunately, in a small Scale widget, the slider takes up a lot of space, so this method provides very poor control for moving the slider close to its current location.

If the user clicks the left mouse button inside the slider area, but not on the slider itself, the slider moves in increments determined by the value of XmNscaleMultiple. The value of this resource defaults to the difference between the maximum and minimum values divided by 10.†For example, a Scale widget whose maximum value is 250 has a scale increment of 25. If the user presses the left mouse button over the area above or below the slider, the Scale's value increases or decreases by 25. If the button is held down, the movement continues until the button is released, even if the slider moves past the location of the pointer.

Scale Resources

The Scale widget is internally implemented using a ScrollBar: many of the ScrollBar resources listed in Chapter 10, *ScrolledWindows and ScrollBars*, are explicitly implemented to support Scale visuals: the ScrollBar resources XmNslidingMode, XmNsliderMark, XmNsliderVisual are *mirrored* in the Scale widget class‡. For example, setting the XmNslidierVisual resource on the Scale internally sets the resource for the constituent ScrollBar.

Probably the most interesting of the mirrored resources is the XmNslidingMode resource. The default value, XmSLIDER, gives the familiar Scale behavior, whereby the slider moves freely between the maximum and minimum points at the Scale ends. The Scale can, however, behave like a classic thermometer, with the slider anchored at one end. XmTHERMOMETER is the required value of XmNslidingMode for this effect. Figure 16-3 shows the difference between the two settings.

* As of Motif 2.0, XmNstringDirection is obsolete, and is replaced by the XmNlayoutDirection resource.

† You should set XmNscaleMultiple explicitly if the difference between XmNmaximum and XmNminimum is less than 10. Otherwise, incremental scaling does not work.

‡ XmNsliderMark, XmNsliderVisual, XmNslidingMode are available only from Motif 2.0 onwards.

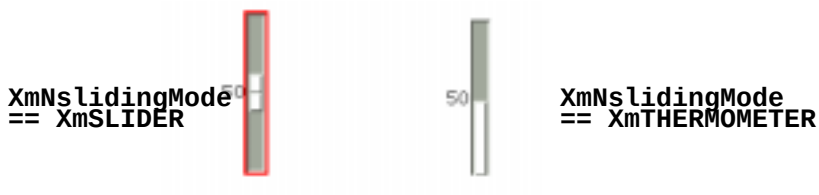


Figure 16-3 The XmNslidingMode resource and its effects

Each of the two Scales have identical XmNmaximum, XmNminimum, XmNvalue resources: they differ only in the value of XmNslidingMode. In the right-hand thermometer Scale, the slider is anchored at one end. Note that when the Scale is configured as a thermometer, directly setting the XmNsliderSize resource has no effect.

Scale Callbacks

The Scale widget provides two callbacks that can be used to monitor the value of the Scale. The XmNdragCallback callback routines are invoked whenever the user drags the slider. This action does not mean that the value of the Scale has actually changed or that it will change; it just indicates that the slider is being moved.

The XmNvalueChangedCallback is invoked when the user releases the slider, which results in an actual change of the Scale's value. It is possible for the XmNvalueChangedCallback to be called without the XmNdragCallback having been called. For example, when the user adjusts the Scale using the keyboard or moves the slider incrementally by clicking in the slider area, but not on the slider itself, only the XmNvalueChangedCallback is invoked.

These callback routines take the form of an XtCallbackProc, just like any other callback. As with all Motif callback routines, Motif defines a callback structure for the Scale widget callbacks.The XmScaleCallbackStruct is defined as follows:

```
typedef struct {
    int      reason;
    XEvent   *event;
    int      value;
} XmScaleCallbackStruct;
```

The reason field of this structure is set to XmCR_DRAG or XmCR_VALUE_CHANGED, depending on the action that invoked the callback. The value field represents the current value of the Scale widget.

Example 16-2 shows another example of how the Scale widget can be used. In this case, we create a color previewer that uses Scales to control the red, green, and blue values of the color that is being edited. This example demonstrates how the XmNdragCallback can be used to automatically adjust colors as the slider is being dragged. The

XmNvalueChangedCallback is also used to handle the cases where the user adjusts the Scale without dragging the slider. For a discussion of the Xlib color setting routines used in this program, see Volume 1, *Xlib Programming Manual*.*

Example 16-2 The color_slide.c program

```
/* color_slide.c -- Use scale widgets to display the different
** colors of a colormap.
*/

#include <Xm/LabelG.h>
#include <Xm/Scale.h>
#include <Xm/RowColumn.h>
#include <Xm/DrawingA.h>

Widget colorwindow; /* the window the displays a solid color */
XColor color; /* the color in the colorwindow */

Widget create_scale (Widget parent, char *name, int mask)
{
    Arg        args[8];
    int        n = 0;
    Widget     scale;
    XrmValue   from;
    XrmValue   to;
    XmString   xms = XmStringCreateLocalized (name);
    void       new_value();

    to.addr = NULL;
    from.addr = name;
    from.size = strlen (name) + 1;
    XtConvertAndStore (parent, XmRString, &from, XmRPixel, &to);

    XtSetArg (args[n], XmNforeground, (*(Pixel *) to.addr)); n++;
    XtSetArg (args[n], XmNtitleString, xms); n++;
    XtSetArg (args[n], XmNshowValue, True); n++;
    XtSetArg (args[n], XmNmaximum, 255); n++;
    XtSetArg (args[n], XmNscaleMultiple, 5); n++;
    scale = XmCreateScale (parent, name, args, n);
    XmStringFree (xms);

    XtAddCallback (scale, XmNdragCallback, new_value, (XtPointer) mask);
    XtAddCallback (scale, XmNvalueChangedCallback, new_value,
                                                    (XtPointer) mask);
    XtManageChild (scale);

    return scale;
}

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

* XtVaAppInitialize() is considered deprecated in X11R6.

```
    Widget         toplevel, rowcol, rc, scale;
    XtAppContext   app;
    Arg            args[8];
    int            n;

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

    if (DefaultDepthOfScreen (XtScreen (toplevel)) < 2) {
        puts ("You must be using a color screen.");
        exit (1);
    }

    color.flags = DoRed | DoGreen | DoBlue;

    /* initialize first color */
    XAllocColor (XtDisplay (toplevel),
                 DefaultColormapOfScreen (XtScreen (toplevel)), &color);

    rowcol = XmCreateRowColumn (toplevel, "rowcol", NULL, 0);

    /* create a canvas for the color display */
    n = 0;
    XtSetArg (args[n], XmNheight, 100);             n++;
    XtSetArg (args[n], XmNbackground, color.pixel); n++;
    colorwindow = XmCreateDrawingArea (rowcol, "colorwindow", args, n);
    XtManageChild (colorwindow);

    /* create another RowColumn under the 1st */
    n = 0;
    XtSetArg (args[n], XmNorientation, XmHORIZONTAL); n++;
    rc = XmCreateRowColumn (rowcol, "rc", args, n);

    /* create the scale widgets */
    scale = create_scale (rc, "Red", DoRed);
    scale = create_scale (rc, "Green", DoGreen);
    scale = create_scale (rc, "Blue", DoBlue);

    XtManageChild (rc);
    XtManageChild (rowcol);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

void new_value (Widget scale_w, XtPointer client_data, XtPointer call_data)
{
    int rgb = (int) client_data;
    XmScaleCallbackStruct *cbs = (XmScaleCallbackStruct *) call_data;
    Colormap cmap = DefaultColormapOfScreen (XtScreen (scale_w));

    switch (rgb) {
        case DoRed   : color.red = (cbs->value << 8); break;
        case DoGreen : color.green = (cbs->value << 8); break;
```

```
            case DoBlue  : color.blue = (cbs->value << 8);
        }
        /* reuse the same color again and again */
        XFreeColors (XtDisplay (scale_w), cmap, &color.pixel, 1, 0);

        if (!XAllocColor (XtDisplay (scale_w), cmap, &color)) {
            puts ("Couldn't XAllocColor!"); exit(1);
        }

        XtVaSetValues (colorwindow, XmNbackground, color.pixel, NULL);
    }
```

The output of this program is shown in Figure 16-4. Obviously, a black and white book makes it difficult to show how this application really looks. However, when you run the program, you should get a feel for using Scale widgets.

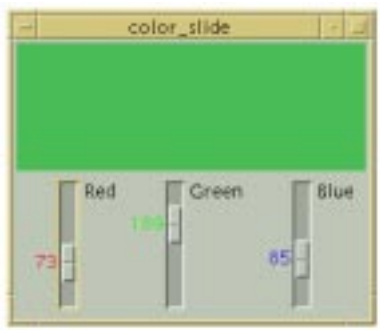


Figure 16-4 Output of the color_slide program

One interesting aspect of the *color_slide.c* program is the use of `XtConvertAndStore()`. We use this function to convert from a string representation of a color name to a Pixel value, so that the toolkit handles the type conversion. For a discussion on type conversion and the use of `XtConvertAndStore()`, see Volume 4, *X Toolkit Intrinsics Programming Manual*.

Scale Tick Marks

The *Motif Style Guide* suggests that a Scale widget can have "tick marks" that represent the incremental positions of the Scale. The Scale widget does not provide these marks by default, but you can add them yourself, either by creating Labels or Separators as children of a Scale widget, or through the routine `XmScaleSetTicks()`*. This convenience function considers ticks to be of three kinds: small, medium, and big. Medium ticks are evenly spaced between the big ticks, and small ticks are evenly spaced between the medium ticks. The function takes the following form:

* `XmScaleSetTicks()` is only available from Motif 2.0 onwards.

```
void XmScaleSetTicks ( Widget         scale,
                       int            big_every,
                       Cardinal       num_medium,
                       Cardinal       num_small,
                       Dimension      size_big,
                       Dimension      size_medium,
                       Dimension      size_small)
```

The function works by creating SeparatorGadget children along the edges of the Scale. The parameter *big_every* specifies the number of scale values between big ticks. The number of medium ticks between big ticks is given by *num_medium*, and the number of small ticks between the medium ticks is given by *num_small*. The *size_** parameters refer to the size of each tick in Pixels. Example 16-3 creates a Scale with tick marks: the code places big ticks every 100 units of scale value, medium ticks at every 10 units in between, and small ticks in the intervening 5 unit mark*. This means that there are 9 (not ten) medium ticks between the big ticks, and only one small tick between the medium ticks.

Example 16-3 The tick_marks.c program

```
/* tick_marks.c -- demonstrate a scale widget with tick marks. */
#include <Xm/Scale.h>

main (int argc, char *argv[])
{
    Widget        toplevel, scale;
    XtAppContext  app;
    int           n;
    Arg           args[8];
    XmString      xms;

    XtSetLanguageProc (NULL, NULL, NULL);

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

    xms = XmStringCreateLocalized ("Process Load");
    n = 0;
    XtSetArg (args[n], XmNtitleString, xms); n++;
    XtSetArg (args[n], XmNminimum, 0); n++;
    XtSetArg (args[n], XmNmaximum, 200); n++;
    XtSetArg (args[n], XmNvalue, 100); n++;
    XtSetArg (args[n], XmNshowValue, True); n++;
    scale = XmCreateScale (toplevel, "load", args, n);

    XmScaleSetTicks(scale, 100, 9, 1, 16, 8, 4);

    XtManageChild (scale) ;
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}
```

* XtVaAppInitialize() is considered deprecated in X11R6. XmScaleSetTicks() is only available from Motif 2.0 onwards.

The output of this program is shown in Figure 16-5.

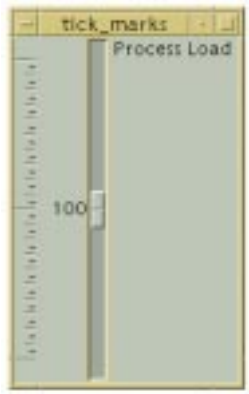


Figure 16-5 Output of the tick_marks program

The Scale can have any kind of widget as a child. All of the children are evenly distributed along the axis of the slider. As you can see in Figure 16-5, the tick marks are placed all the way to the left of the Scale widget to leave space for the value indicator. If you wanted to add ticks using `XmScaleSetTicks`() as well as adding other children, it could be rather difficult to achieve any kind of sensible layout without some particularly non-trivial coding.

Summary

The Scale widget is a simple widget, both in concept and in practical use. In this chapter, we have showed a few possible uses of the Scale to represent a range of values. The range of a Scale, as well as its orientation, are customizable. The widget also provides callbacks that allow an application to keep track of the value of the Scale as the user changes it. These features make the Scale quite versatile.

In this chapter:
- *Creating a Notebook*
- *Notebook Resources*
- *Notebook Constraints*
- *Notebook Callbacks*
- *Notebook Functions*
- *Summary*

17

The Notebook Widget

This chapter describes the Notebook widget, introduced in Motif 2.0.

The Notebook is a Constraint Manager which organises its children as though they are pages in a book. At any given time, only one child is visible. The Notebook has visuals so that it appears as though the children are stacked on top of each other like real pages; a spiral binding and overlapping back pages complete the appearance.

Continuing the analogy, the pages of a Notebook can be divided into sections and sub-sections by creating tabs which are inserted along the edges of the Notebook pages, just as real notebooks have tab inserts. Tabs can be associated with pages, so that activating a tab causes a specific child (page) to be displayed. Each section involves the creation of a Major Tab - this is simply a button with a particular constraint resource set, and each sub-section is added by creating a Minor Tab, which again is simply a button with an appropriate constraint. Minor Tabs appear on a different edge to the Major Tabs along the Notebook sides, which is possibly the one aspect of behavior which is not the same as a real notebook. The Notebook automatically creates a Tab Scroller, consisting of a pair of ArrowButtonGadgets, for scrolling along the set of Tabs when there are more Tabs than can be displayed along the Notebook edge.

The Notebook can also contain a Page Scroller, so that logical pages can be displayed by rotating through the values of a SpinBox, rather than having to manually select each tab individually.

Any given page can also be assigned a Status Area, which is a region of the Notebook display which can be used for information such as the current page number, date, and so forth.

Typically, pages are added to a Notebook simply by adding a Manager child to the widget: the various elements of the page are added to this Manager in turn. The Notebook has a default algorithm which takes a note of each child as it is added: it assumes that any added Manager is supposed to represent a page, a button will form a Major Tab, a textual widget will form a Status Area, and a SpinBox will form a Page Scroller. These are usually valid assumptions, and so it is not normally necessary to assign roles to each of the children as they are added to the Notebook by assigning the relevant constraint resource. For typical

usage, only the role of Minor Tab need to be explicitly coded by the programmer. As for associating Tabs with pages, again the Notebook has a default algorithm which assigns ascending page numbers to objects in the order in which they are added. The first Page child is assigned the logical page number 1 by default. The programmer is free to explicitly assign logical pages to each of the various Tab and Page children: there can even be gaps in the numbering scheme, so that a Tab can refer to a number for which no corresponding Page child exists. In this case, the Notebook simply displays a blank background when the Tab is activated. It is also possible to assign logical page numbers to Tabs and Pages out of order to child widget creation: logical page numbers can be assigned in any order, although in this case the programmer should of course make sure that his own code does not implicitly rely on the order of child creation in any way. The Notebook will automatically sort the children into ascending page order.

This all seems to indicate that a Notebook needs to be fully loaded with all pages before displaying the widget to the user. This is not the case: it is possible to create pages dynamically by associating a callback with the Notebook which is called whenever a Tab is activated: the relevant page associated with the Tab can then be created on demand.

Figure 17-1 shows a Notebook with all the various elements indicated.

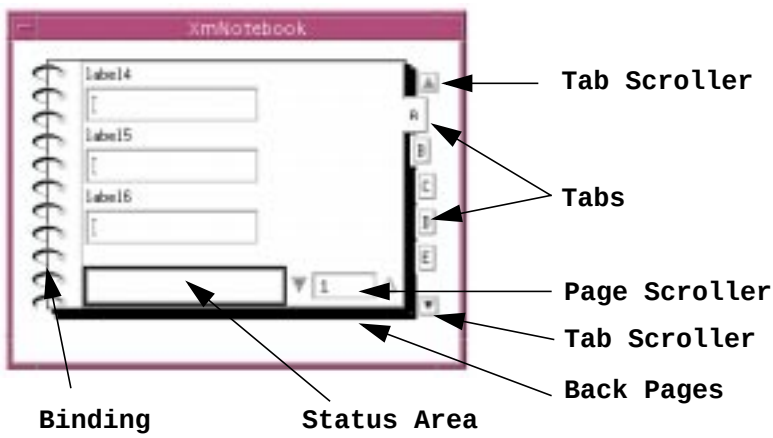


Figure 17-1 The Notebook widget and its constituent parts

The Notebook is highly configurable. Each of the various components of the Notebook are fully under programmer control, so that it is possible to convert the Notebook into a traditional Tab Manager, simply through setting appropriate resources to disable some of the more florid aspects of the illusion and to rotate the Tabs onto the top of the pages. The Spiral Binding can be configured for various styles of presentation, including a programmer-supplied pixmap; it can also be removed, as can the overlapping page illusion. The Page Scroller, Status Area, and Tabs are entirely optional, although it is difficult to imagine the usefulness of a Notebook which contained neither Page Scroller nor Tabs:

selecting between pages would then become problematic as far as the user is concerned, although this arrangement would make some sense if page display is fully under program control.

Figure 17-2 is an example of this kind of traditional Tab Manager configuration., where the spiral binding and overlapping page illusions have been removed,the Tabs rotated to the top of the Notebook, and the Page Scroller disabled.

Figure 17-2 A Notebook configured to behave like a traditional Tab Manager

Creating a Notebook

Incorporating the Notebook widget into your code is straightforward. An application which uses the Notebook widget must include the header file *<Xm/Notebook.h>*. The header file declares the types of the public Notebook functions and the widget class name `xmNotebookWidgetClass`. A Notebook can be created in either of the ways as shown in the following code fragment:

```
Widget notebook = XmCreateNotebook ( parent, name, resource-value-array,
                                     resource-value-count)
Widget notebook = XtCreateWidget ( "name", xmNotebookWidgetClass, parent,
                                   resource-value-list, NULL);
```

The Notebook can potentially manage a very large number of children, and so it is probably best not to create the widget in a managed state (`XtCreateManagedWidget`()) otherwise performance may suffer. See Chapter 8, *Manager Widgets*, for a discussion of when widgets should be created in the managed or unmanaged state.

The *`parent`* of the Notebook can be any Shell or Manager widget. Once the Notebook has been created, the next step is to add logical Pages to the Notebook, and thereafter add Tab inserts when and if required. The Notebook is a *constraint* widget: as children are added, the role which the child is to take is specified using a constraint resource, `XmNnotebookChildType`. The possible values of the resource are as follows:

```
XmMAJOR_TAB             XmMINOR_TAB
XmPAGE_SCROLLER         XmSTATUS_AREA
XmPAGE
```

The Notebook has a default algorithm for assigning roles to children. Unless specified otherwise, any Manager child is assigned the XmNnotebookChildType constraint value XmPAGE, any PushButton is assigned the value XmMAJOR_TAB, any Textual component is given the value XmSTATUS_AREA, and any SpinBox or derivative is given the role XmPAGE_SCROLLER. If no page scroller is added to the Notebook, the widget creates one for its own use automatically.

Creating Notebook Pages

As stated above, any Manager child which is added to the Notebook is assumed to be a logical page. Logical page numbers are assigned by the Notebook in ascending order as required. Adding a Page to the Notebook is therefore simple: create a Form or other Manager widget as child of the Notebook, and then add the page contents by way of adding extra children to the Form. The following code in Example 17-1 creates a Notebook with four pages.

Example 17-1 The simple_notebook.c program

```
/* simple_notebook - create a Notebook with four pages
*/

#include <Xm/Xm.h>
#include <Xm/Notebook.h>
#include <Xm/RowColumn.h>
#include <Xm/Label.h>

/* Arbitrary data for the first “page” */
char *page1_labels[] =
{
    “Introduction”,
    “”,
    “The Motif Programming Model”,
    “Overview of the Motif Toolkit”,
    “The Main Window”,
    “Introduction to Dialogs”,
    (char *) 0
};
/* Arbitrary data for the second “page” */
char *pane2_;labels[] =
{
    “Selection Dialogs”,
    “Custom Dialogs”,
    “Manager Widgets”,
    “The Container and IconGadget Widgets”,
    “Scrolled Windows and ScrollBars”,
    (char *) 0
};
/* Arbitrary data for the third “page” */
```

```
char *page3_labels[] =
{
    “The DrawingArea”,
    “Labels and Buttons”,
    “The List Widget”,
    “The ComboBox Widget”,
    “The SpinBox and SimpleSpinBox Widgets”,
    (char *) 0
};
/* Arbitrary data for the fourth “page” */
char *page4_labels[] =
{
    “The Scale Widget”,
    “The Notebook Widget”,
    “Text Widgets”,
    “Menus”,
    (char *) 0
};

/* A pointer to all the “page” data */
char **pages[] =
{
    page1_labels,
    page2_labels,
    page3_labels,
    page4_labels,
    (char **) 0
};

void main (int argc, char **argv)
{
    Widget          toplevel, notebook, page, label;
    XtAppContext    app;
    Arg             args[4];
    XmString        xms;
    int             i, j, n;

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

    /* Create the Notebook */
    notebook = XmCreateNotebook (toplevel, “notebook”, NULL, 0);

    /* Create the “pages” */
    for (i = 0; pages[i] != (char **) 0; i++) {
        page = XmCreateRowColumn (notebook, “page”, NULL, 0);

    for (j = 0; pages[i][j] != (char *) 0; j++) {
        xms = XmStringGenerate (pages[i][j], XmFONTLIST_DEFAULT_TAG,
                                XmCHARSET_TEXT, NULL);

    n = 0;
    XtSetArg (args[n], XmNlabelString, xms); n++;
```

```
            label = XmCreateLabel (page, "label", args, n);
            XtManageChild (label);
            XmStringFree (xms);
        }
        XtManageChild (page);
    }

    XtManageChild (notebook);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}
```

The output of the program is given in Figure 17-3.

Figure 17-3 Output of the simple_notebook program

Creating Notebook Tabs

Just as the Notebook assumes that Manager children are pages, so it assumes that PushButton children are to form Tab inserts. Tabs are considered to be of two kinds: Major Tabs, which presumably separate the important sections of the Notebook children, and Minor Tabs, which mark less important boundaries between the Major Tabs. The Notebook by default assumes that any Tab child is a Major Tab, and so explicit action must always be taken by the programmer when providing the Minor Tab inserts by specifying the appropriate constraint resources. A Major Tab has the XmNnotebookChildType constraint set to XmMAJOR_TAB, and a Minor Tab has the value XmMINOR_TAB. The following additional code adds a Major and a Minor Tab to Example 17-1 (Figure 17-3).

```
Widget       tab;
char         buffer[32];
...
/* Create the "pages" */
for (i = 0; pages[i] != (char **) 0; i++) {
    page = XmCreateRowColumn (notebook, "page", NULL, 0);

    for (j = 0; pages[i][j] != (char *) 0; j++) {
        ...
    }

    /* An even page is a Major Tab */
    /* And an odd page is a Minor Tab */
    n = 0;
```

```
        if ((i % 2) == 0) {
            XtSetArg (args[n], XmNnotebookChildType, XmMAJOR_TAB); n++;
        }
        else {
            XtSetArg (args[n], XmNnotebookChildType, XmMINOR_TAB); n++;
        }
        (void) sprintf (buffer, "%s %d", ((i % 2) == 0) ? "Major" : "Minor", i);
        tab = XmCreatePushButton (notebook, buffer, args, n);
        XtManageChild (tab);
        XtManageChild (page);
    }
```

The output of the modifications results in the dialog shown in Figure 17-4.

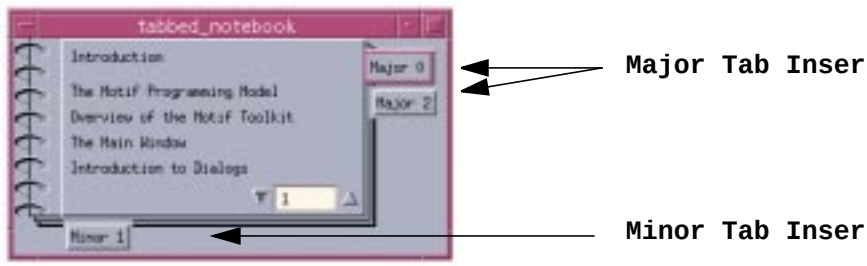


Figure 17-4 The Notebook with added Tabs

Clearly there is a difference in the way that the Notebook displays Major and Minor Tabs. All the Major Tabs are displayed along one edge of the Notebook. However, only the Minor Tabs which logically are associated with a page number between the current active Major Tab and the next Major Tab are displayed. If in the Figure 17-4 Major Tab "Page 2" is activated, only the Minor Tab "Page 3" is displayed along the bottom.

Manipulating the Page Scroller

The Notebook creates a SpinBox with a single TextField child to act as a Page Scroller automatically, unless the programmer takes steps to provide her own Page Scroller when the Notebook is created. This means that in principle trying to create Notebook pages which are only accessible through the Tab mechanisms looks a little tricky. In practice, the default Page Scroller can be removed simply enough by accessing the built-in SpinBox widget, and then unmanaging it. The Notebook creates the SpinBox using the constant name "PageScroller", and so the following code fragment will effectively hide the Page Scroller from view:

```
extern Widget notebook;
Widget scroller;

scroller = XtNameToWidget (notebook, "PageScroller");
XtUnmanageChild (scroller);
```

If you wanted to add your own Page Scroller, you would need to create the widget as a child of the Notebook, setting the XmNnotebookChildType constraint to XmPAGE_SCROLLER.

Notebook Resources

The resources associated with the Notebook fall into three basic sets: those which configure the current pages on view, those which controls the visuals associated with the binding and page illusions, and those which configure the Tab placement.

Page Resources

The current logical page displayed by the Notebook is specified using the resource XmNcurrentPageNumber. This can be used to programmatically display a given page. If the programmer wishes to set lower and upper bounds upon the pages in view, the resources XmNfirstPageNumber and XmNlastPageNumber resources can be specified. The Notebook will not display any Tabs or Pages whose logical page assignments fall outside the range falling between the two values. How logical page numbers are assigned to children is covered in the following *Notebook Constraints* Section. Any Page which has an assigned page number outside the range can only be displayed if the programmer either modifies the range, or re-assigns the logical number associated with the Page so that it falls within the first/last page bounds. By default, the XmNlastPageNumber value is maintained by the Notebook itself as page children are added to the widget. However, if the programmer sets this XmNlastPageNumber value herself, the Notebook no longer maintains the value, and it becomes the programmers responsibility to maintain the value from then on. The automatic page numbering scheme uses the XmNfirstPageNumber resource to seed the counting algorithm. By default, the first page number is 1.

Visual Resources

The page illusion can be configured through a variety of resources, both to configure the general layout, and to specify the appearance.

The color of the back page is controlled using XmNbackPageForeground and XmNbackPageBackground, which are Pixel-valued resources. The number of overlapping pages which constitute the back page (or rather, the number of lines drawn to give the appearance of overlapping pages) is controlled through the XmNbackPageNumber resource, the thickness of the lines drawn being specified using the XmNbackPageSize resource.

The size of the binding drawn along the Notebook is bounded by the value of the XmNbindingWidth resource, the binding style itself is specified through XmNbindingType. This has the following possible values:

```
XmSPIRAL      XmNONE      XmPIXMAP      XmPIXMAP_OVERLAP_ONLY      XmSOLID
```

The default is XmSPIRAL, which displays a spiral binding down the edge of the Notebook. XmSOLID draws a solid binding using the foreground color of the widget. The binding can be removed using the value XmNONE: this would be done if you wanted to turn the Notebook into something more along the lines of a traditional Tab Manager. You can also supply your own binding in the form of a Pixmap using the XmNbindingPixmap resource. If you do this, you need to also specify that the XmNbindingType is either XmPIXMAP or XmPIXMAP_OVERLAP_ONLY. The difference is that if the type is XmPIXMAP, your image will not be clipped by any XmNbindingWidth resource - the binding will grow if your Pixmap is wider than the current XmNbindingWidth. On the other hand, if the binding type is XmPIXMAP_OVERLAP_ONLY, the size of the binding drawn is bounded by the XmNbindingWidth value.

Figure 17-5 shows the effect of setting the various binding appearance resources.

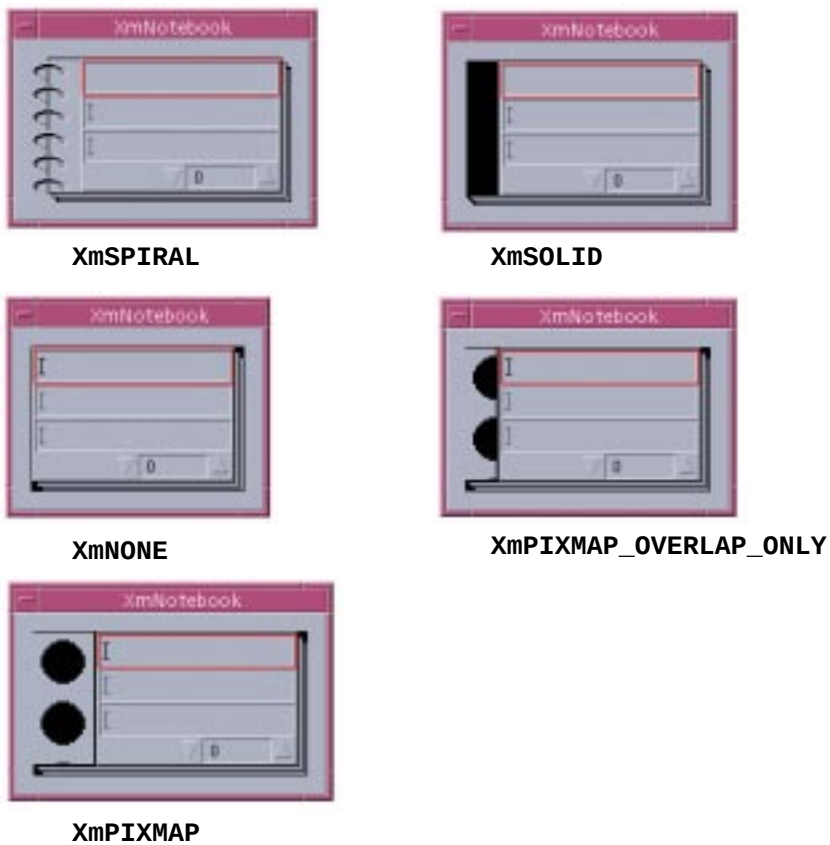


Figure 17-5 The Notebook binding styles

The general layout of the Notebook is configured using the XmNbackPagePlacement resource. This is probably misnamed, in that although the resource does indeed configure

the placement of the back page illusion, it also configures the location of the Tabs and the binding as a side effect. The resource has the following possible values:

XmBOTTOM_RIGHT	XmBOTTOM_LEFT
XmTOP_RIGHT	XmTOP_LEFT

The default value is sensitive to the XmNlayoutDirection and XmNorientation resources. Figure 17-6 shows the effects of setting the various values:

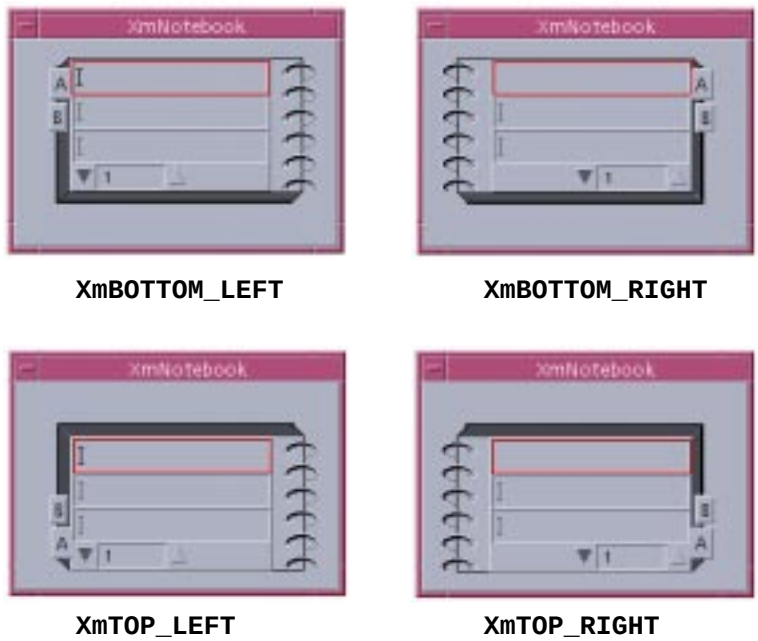


Figure 17-6 The Notebook binding placements

Clearly the XmNbackPagePlacement resource also affects the binding and Tabs, because the binding is always opposite the overlapping back page illusion, and the Major Tabs are placed along the back page opposite the binding.

The orientation of the Notebook can be configured through the XmNorientation resource, which has the possible values XmHORIZONTAL and XmVERTICAL. Figure 17-7 shows a vertically-oriented Notebook. The spiral binding has also been removed so that the Notebook looks more like a traditional Tab Manager. The Tabs, however, appear along the top as opposed to the sides, which happens with the horizontal arrangement. If you wanted the Major Tabs to appear on the bottom of the Notebook, you would need to set the XmNorientation to XmVERTICAL, and the XmNbackPagePlacement to XmBOTTOM_LEFT or XmBOTTOM_RIGHT to choice.

.

Figure 17-7 The Notebook with a vertical orientation

Tab Resources

Tab placement has already been discussed in the previous paragraphs - the XmNbackPagePlacement resource configures the Tab placement as a side effect. All that remains which can potentially be configured is the spacing between the Tabs and the Notebook pages. The distance between Major Tabs and the page border is specified using the XmNmajorTabSpacing resource, and the distance between Minor Tabs and the page is specified through the XmNminorTabSpacing resource. Setting these values to zero, as well as specifying the XmNbackPageNumber resource as zero, gives an appearance which is more consistent with a normal Tab Manager.

Notebook Constraints

The roles which each child can take in the various Notebook operations is controlled through the XmNnotebookChildType resource. This can take the following values:

```
XmPAGE
XmMAJOR_TAB          XmMINOR_TAB
XmSTATUS_AREA        XmPAGE_SCROLLER
```

The XmNnotebookChildType resource is create-only, which means that you must specify the resource when you add the child if the required role differs from the default which would be assigned by the Notebook. Formally, the default algorithm assumes that if the child supports the XmQTactivatable Trait, then the child is assigned the constraint value XmMAJOR_TAB. The PushButton, DrawnButton, ArrowButton and derived classes support this Trait. If the child widget supports the XmQTnavigator Trait, then the child by default is assigned the constraint value XmPAGE_SCROLLER. The ScrollBar, SpinBox, and derived widget classes support this Trait. If the child widget supports the XmQTaccessTextual Trait, then the role XmSTATUS_AREA is the default. The Label, LabelGadget, Text, TextField, and derived classes support the XmQTaccessTextual Trait. Everything else is assigned the default role XmPAGE. It follows that all Minor Tabs must be explicitly set by the programmer since the default algorithm does not assign this role. The internals of the Trait mechanisms are beyond the scope of this book.

The association between Tabs and Pages is specified using the XmNpageNumber constraint. A Tab will display a given Page when activated if the Tab and Page child share the same XmNpageNumber constraint value.

The last constraint defined by the Notebook is the XmNresizable resource, which simply specifies whether the Notebook will process resize requests from the given child.

Notebook Callbacks

The Notebook defines a single callback, the XmNpageChangedCallback. This is called whenever there is a request to change the current Logical Page, whether through the activation of a Tab or through selection using a Page Scroller. Each callback when invoked is passed a pointer to an XmNotebookCallbackStruct structure, which is defined as follows:

```
typedef struct
{
    int          reason;
    XEvent       *event;
    int          page_number;
    Widget       page_widget;
    int          prev_page_number;
    Widget       prev_page_widget;
} XmNotebookCallbackStruct;
```

The reason element specifies the cause of callback invocation. It may have any of the following possible values:

```
XmCR_NONE           XmCR_MAJOR_TAB      XmCR_MINOR_TAB
XmCR_PAGE_SCROLLER_INCREMENT            XmCRPAGE_SCROLLER_DECREMENT
```

The value will be XmCR_NONE on Notebook invocation, as the widget chooses the first current page. It will also be XmCR_NONE if the XmNcurrentPageNumber resource is changed programmatically. Otherwise, the value will reflect a user action, depending upon whether a Tab of some kind has been activated, or the Page Scroller selected.

The page_number field represents the new logical page number. This may, or may not, be associated with a real Page. If the page_number does refer to a child with a matching XmNpageNumber constraint value, then the page_widget element will be set to this child. Otherwise, the page_widget element will be NULL, and the Notebook will display a blank page.

The prev_page_number and prev_page_widget elements specify the old current page. At Notebook initialisation, prev_page_number is the value XmUNSPECIFIED_PAGE_NUMBER, and prev_page_widget is NULL. The prev_page_widget element will also be NULL if the Notebook is currently displaying a blank Page - no child widget has the constraint XmNpageNumber which matches the prev_page_number value.

The fields of the callback structure are not used by the Notebook routines when the XmNpageChangedCallback terminates. This means that in effect the elements are all read-only and purely notificatory in effect: modifying the page_number or page_widget element in the hope of programmatically setting the next page will have no effect. The resource XmNcurrentPageNumber should be set directly if this is the required behavior.

The following code fragment simply prints out the state information as various changes take place to the Notebook current page:

```
void notebook_changed_callback ( Widget         w,
                                 XtPointer      client_data,
                                 XtPointer      call_data)
{
    XmNotebookCallbackStruct *nptr;
    nptr = (XmNotebookCallbackStruct *) call_data;

    if (nptr->reason == XmCR_NONE) {
        if (nptr->prev_page_number == XmUNSPECIFIED_PAGE_NUMBER) {
            printf ("Notebook initialisation: first page %d\n",
            nptr->page_number);
        }
        else {
            printf ("Program request: new page %d\n",
            nptr->page_number);
        }
    }
    else {
        printf ("New Page: %d %s Old page: %d %s\n",
                nptr->page_number,
                (nptr->page_widget ?
                    XtName(nptr->page_widget) :
                        "(blank)"),
                nptr->prev_page_number,
                (nptr->prev_page_widget ?
                    XtName(nptr->prev_page_widget) :
                        "(blank)"));
    }
}
```

Notebook Functions

A programmer can request information about a logical page of the Notebook using the routine XmNotebookGetPageInfo(), which has the following signature:

```
XmNotebookPageStatus XmNotebookGetPageInfo ( Widget              notebook,
                                             int                 page_number,
                                             XmNotebookPageInfo  *page_info)
```

The return value is an XmNotebookPageStatus, which is an enumerated type. Possible values are:

```
XmPAGE_FOUND                XmPAGE_EMPTY
XmPAGE_INVALID              XmPAGE_DUPLICATED
```

If the requested *page_number* falls outside the bounds between the *notebook* XmNfirstPageNumber and XmNlastPageNumber resources, the function returns XmPAGE_INVALID. Otherwise, if exactly one child which has an XmNpageNumber constraint which matches *page_number* is found, the return value is XmPAGE_FOUND. If more than one child shares the given page number, the return value is XmPAGE_DUPLICATED. Otherwise, the return value is XmPAGE_EMPTY.

The *page_info* parameter specifies an address where data about the requested *page_number* is returned. The data is filled in by the XmNotebookGetPageInfo() routine except if the return value is XmPAGE_INVALID. Where there are duplicate pages with the requested *page_number*, the information will refer to the last child found. The XmNotebookPageInfo data type is defined as follows:

```
typedef struct
{
    int        page_number;
    Widget     page_widget;
    Widget     status_area_widget;
    Widget     major_tab_widget;
    Widget     minor_tab_widget;
} XmNotebookPageInfo;
```

If the matching child found is a Status Area, the status_area_widget element will be filled in with the widget ID of the child. Similarly, a matching Major Tab child is placed into the major_tab_widget element, a matching Minor Tab is stored in the minor_tab_widget element, and a matching Page child is stored in the page_widget field.

In addition, Major Tabs and Minor Tabs are stored into the major_tab_widget and minor_tab_widget fields as they are encountered regardless of whether the page number matches. Since children of the Notebook are stored internally in sorted ascending XmNpageNumber order, and since the search terminates if a child is found with a page number exceeding the request, it means that the major_tab_widget and minor_tab_widget elements contain the Tab with a page number nearest to (but not in excess of) the requested page number in addition to any data returned in the page_widget field.

Summary

The Notebook is a most useful, if some what over-decorous, addition to the Motif 2.0 widget set. It performs the services of a traditional Tab Manager by displaying only one given page child at a time. Indeed the visuals of the Notebook can be configured to give the usual appearance of a Tab Manager, and it is in this form that it is most likely to be used in

typical application programming. It works by assigning roles to the various children as they are added: the default algorithm for assigning page numbers and roles is more than sufficient for most purposes, but the programmer has sufficient control should the need to override the defaults arise.

In this chapter:

- *Interacting With Text Widgets*
- *Text Widget Basics*
- *Text Clipboard Functions*
- *A Text Editor*
- *Text Callbacks*
- *Text Widget Internationalization*
- *Summary*
- *Exercises*

18

Text Widgets

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

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

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

The text widgets do have their limitations. For example, they do not support multiple colors or fonts, so a single widget can only use one color and one font*. There is no support for text formatting such as paragraph specifications, automatic line numbering, or indentation, so you cannot create WYSIWYG documents.†The Text widget is not a terminal emulator;

* The abortive compound string Text widget, CSText, was introduced in Motif 2.0. This did support multi-font capability, but it had serious performance problems; it was excised from Motif 2.1.

it cannot be used to run interactive programs. The widgets cannot display multi-media objects either, which means that it is not possible to insert graphics into the text stream.

There are some cases where a text widget is not the most appropriate user-interface element, even though you are displaying text. For example, a Text widget should not be used to display a list whose items can be individually selected; that is the job of the List widget. Text that cannot be edited, or selected should be displayed in a Label widget. Chapter 12, *Labels and Buttons*, and Chapter 13, *The List Widget*, describe the appropriate uses of these components.

If you have not used the Motif Text widget, you should familiarize yourself with one before getting too involved in this chapter. Running some of our introductory examples should provide an adequate platform for experimentation. Figure 18-1 shows an application that uses several Text widgets. Two widgets are used for single-line data entry. The widget with the ScrollBars attached to it is used for editing multiple lines.

Figure 18-1 An editor application with both Text and TextField widgets

The Text widget supports both single-line and multi-line editing. In single-line mode, which is the default mode, newlines are ignored. However, single-line text entry is usually done with the TextField widget class. This widget class is a completely separate class, not a subclass, of Text that is lighter-weight because it only supports single-line text editing. Although they are two separate widget classes, the Text and TextField widgets provide many of the same resources and convenience routines. We will point out the differences as

† WYSIWYG stands for *What You See Is What You Get*. This term is used to describe page formatting programs that can produce camera-ready documents that match what is displayed on the screen.

we go, but keep in mind that there are two widget classes so you don't get confused as we discuss them throughout this chapter.

Since the TextField widget cannot handle multiline editing, you must use the Text widget for this purpose. When multiple lines are used for editing, the number of lines typically grows and shrinks dynamically as the user edits the text. The Text widget is often used in a scrollable window, so that the user can view different portions of the underlying text. The combination of a Text widget and a ScrolledWindow widget is called a ScrolledText object. This object is not a widget class, although there is a convenience routine, `XmCreateScrolledText()`, that allows you to create both widgets at once.

Interacting With Text Widgets

The Text and TextField widgets are highly configurable in terms of appearance and behavior. Given the level of sophistication for both the programmer and the user, the widgets should not be taken lightly or underestimated. The ease of configurability should not tempt you to enforce your personal ideas about how a text editor should work. The best thing to do with text widgets is configure them as minimally as possible to suit the needs of your program. You should let the user have control over as many details of their display and operation as possible. This laissez-faire approach ensures that your application is more compatible with other Motif programs.

Inserting Text

The user interface for the text widgets follows the point-and-click model of interaction. The insertion cursor indicates the position where the next character that is typed will be inserted. The insertion position is marked by an I-beam cursor. Using the left mouse button, the user can click on a new location in the widget to move the insertion cursor there, so text may be inserted at any location in the widget.

The text widgets have predefined action routines that allow the user to perform simple editing operations such as moving one character to the right or deleting backwards to the beginning of the line. The user can specify translations in a resource file that modify the input behavior of the widgets. The widgets are by default always in text-insertion mode. There is an action that puts the Text widget in overstrike mode if this is required.

The user can use the action routines provided by the widgets to set up the translation table to mimic an editor such as *emacs*. The Text widget does not insert non printable characters, so users typically bind control-character sequences to editing action routines. An editor like *vi* cannot be emulated because there is no distinction between command mode and text-entry mode, and in this sense, the Motif Text widgets are completely modeless.

Selecting Text

Users have become accustomed to the ability to cut and paste text between windows in GUI-based applications. Cut and paste is more difficult for the programmer to implement with the X Window System than a system where a single vendor controls all of the variables, because the nature of X requires a more general solution. For example, applications running on the same display may actually be executing on different systems; these systems may have different byte orders or other differences in the underlying data format.* In order to insulate cut and paste operations from dependencies like these, all communication between applications is implemented via the X server. Data that is cut is stored in a *property* on the X server. A property is simply a named piece of data associated with a window and stored on the server.

The *Interclient Communications Conventions Manual*† (ICCCM) defines a set of standard property names to be used for operations such as cut and paste and lays out rules for how applications should interact with these properties. According to the ICCCM, text that is selected is typically stored in the PRIMARY property. The SECONDARY property is defined as an alternate storage area for use by applications that wish to support more than one simultaneous selection operation or that wish to support operations requiring two selections, such as switching the contents of the two selections. The CLIPBOARD property is defined as a longer-term holding area for data that is actually cut (rather than simply copied) from the application's window. When we refer to the primary, secondary, or clipboard selection, we mean the property of the same name.

The most common implementation of the selection mechanism is provided by the X Toolkit Intrinsics. The low-level routines that are used to implement selections are described in detail in Volume 4, *X Toolkit Intrinsics Programming Manual*. In general, applications such as *xterm* and widgets such as the Motif Text widget encapsulate this functionality in action routines that are invoked by the user with mouse button or key combinations.

The user can select text in a Motif Text widget by pressing the left mouse button and dragging the pointer across the text. The selected text is displayed in reverse video. When the button is released, the text widget has ownership of the selection, but no text is copied. The selection can be extended either by pressing the SHIFT key and then dragging the pointer with the left mouse button down, or by pressing any of the arrow keys while holding down the SHIFT key. In addition to the click-and-drag technique for text selection, the Text widget also supports multiple-clicking techniques: double-clicking selects a word, triple-clicking selects the current line, and quadruple-clicking selects all of the text in the widget. An important constraint imposed by the ICCCM is that only one window may own a

* Currently, only text selections are implemented, which makes byte order irrelevant. However, the mechanism is designed to allow transparent transfer of any kind of data.

† Reprinted as *Appendix L in Volume Zero, X Protocol Reference Manual*.

selection property at one time, which means that once the user makes another primary selection, the original selection is lost.

The user can copy text directly from the primary selection into the Text widget by clicking the middle mouse button at the location where the text is to be inserted. This action is sometimes called stuffing the selection into the widget. The user can stuff text at any location in the text stream, as long as the location is not inside the current selection. The text is copied only when the middle mouse button is clicked, which is defined as a quick succession of press and release actions. The operation does not take place simply because the middle mouse button is pressed, as this action is used for drag and drop operations.

In Motif, the Text and TextField widgets support the drag-and-drop model of transferring textual data. Once text has been selected in a widget, the selection can be dragged by pressing the middle mouse button over the selection and dragging the pointer. The text is transferred when the user releases the middle mouse button with the pointer over another location in the same widget or over another text widget. By default, the text is moved, which means that the original text is deleted once the transfer is complete. The user can force a copy operation by holding down the CONTROL key while dragging the pointer and releasing the mouse button. For more information on drag and drop, see Chapter 22, *Drag and Drop*, and Chapter 23, *The Uniform Transfer Model*.

The secondary selection is used by the Motif text widgets to copy text directly within a widget. The user performs this type of operation by first selecting the location where the copied text is to be placed; clicking the left mouse button places the insertion point. Then the text that is to be copied is selected by pressing and dragging the middle mouse button while the ALT key is pressed. The selected text is underlined rather than highlighted in reverse video. When the button is released, the selected text is immediately stuffed at the location of the insertion cursor. Unlike the primary selection, which may be retrieved many times, the secondary selection is immediate and can only be stuffed once.

The third location for holding text is the clipboard selection. The clipboard selection is designed to be used as a longer-term storage area for data. For example, MIT provides a client called *xclipboard* that asserts ownership of the CLIPBOARD property and provides a user interface to it. *xclipboard* not only allows a selection to survive the termination of the window where the data was originally selected, but it also allows for the storage of multiple selections. The user can view all of the selections before deciding which one to paste.

OSF's implementation of the clipboard is incompatible with *xclipboard*. If *xclipboard* is running, any Motif routines that attempt to store data on the clipboard will not succeed. The Motif routines temporarily try to lock the clipboard, and *xclipboard* will not give up its own lock. Motif treats the clipboard as a two-item cache. Only Motif applications that use the clipboard routines described in Chapter 21, *The Clipboard*, can inter-operate using this selection. The advantage of the Motif implementation is that it provides functionality far

beyond that provided by the standard MIT clients. With *xterm* and the Athena widgets, selections can really only be used for copy-and-paste operations; the selected text is unchanged. The Motif Text widget, by contrast, allows you to cut, copy, clear, or type over a selection. While there is a translation and action-based interface defined for these operations, it is typically not implemented.

As described in Chapter 2, *The Motif Programming Model*, Motif defines translations in terms of virtual key bindings. By default, the virtual keys *osfCut*, *osfCopy*, *osfPaste*, *et. al.* , are not bound to any actual keys. If a user wants to use these keys, he must specify the bindings in a *.motifbind* file in his home directory. The interface for these features is usually provided by menu items associated with the Text widget, as we will demonstrate in this chapter.

When text is selected in a Text widget, it is automatically stored in the primary selection. When one of the Text widget functions, such as `XmTextCut()`, is used, the text is also stored in the clipboard selection. Most users will be completely unaware that there are separate holding areas for selected text. If your application gets heavily into cutting and pasting, you may find that the fusion of the primary and clipboard selections in the convenience routines is confusing. You should be careful to implement the selection operations so that the different properties are transparent to the user.

In Motif 2.0 and later, the Uniform Transfer Model as described in Chapter 23 hides some of the internal details of differences between the various methods of data transference from the programmer; note that the UTM does not, however, hide the differences between selection, the clipboard, or drag-and-drop from the user.

The reference pages for the Text and TextField widgets (in Volume 6B, *Motif Reference Manual;* Section 2, *Motif and Xt Widget Classes*) lists the default translations for the widgets. See Volume 4, *X Toolkit Intrinsics Programming Manual*, for a description of how to programmatically alter translation tables; see Volume 3, *X Window System User's Guide*, for a description of how a user can customize widget translations. See Chapter 21, *The Clipboard*, for a discussion of the lower-level Motif clipboard functions.

Text Widget Basics

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

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

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

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

The Textual Data

The `XmNvalue` resource of the Text and TextField widgets provides the most basic means of access to the internal text storage for the widgets. Unlike the other widgets in the Motif toolkit that use text, the text widgets do not use compound strings for their values. Instead, the value is specified as a regular C string, as shown in Example 18-1.*

Example 18-1 The simple_text.c program

```
/* simple_text.c -- Create a minimally configured Text widget */

#include <Xm/Text.h>

main (int argc, char *argv[])
{
    Widget             toplevel;
    XtAppContext       app;
    Widget             text_w;
    Arg                args[2];

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

    XtSetArg (args[0], XmNvalue, "Now is the time...");
    text_w = XmCreateText (toplevel, "text", args, 1);
    XtManageChild (text_w);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}
```

This short program simply creates a Text widget with the initial value as shown in Figure 18-2.

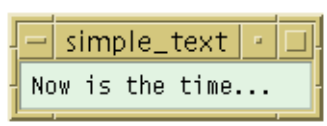


Figure 18-2 Output of the simple_text program

* `XtVaAppInitialize`() is considered deprecated in X11R6.

Both widgets also provide the XmNvalueWcs resource for storing a wide-character representation of the text value. For more information on using the text widgets in an internationalized application, see the *Text Widget Internationalization* Section.

Specifying the Text

The initial value of the XmNvalue resource may be set either when the widget is created or by using XtVaSetValues() after the widget has been created. The value of the resource always represents the entire text of the widget. You can also use a Motif convenience routine, XmTextSetString(), to set the text value. This routine takes the following form:

```
void XmTextSetString (Widget text_w, char *value)
```

This routine works for both Text and TextField widgets. The TextField widget has a corresponding routine, XmTextFieldSetString(), but it only works for TextField widgets. If you are using both types of text widgets in an application, we recommend using the Text widget routines to manipulate all of the widgets. Since these routines work with both types of widgets, you don't need to keep track of the widget types.

Although the convenience routine and XtVaSetValues() produce the same results, the convenience routine may be more efficient since it accesses the internals of the widget directly, while the XtVaSetValues() method involves going through Xt. On the other hand, if you are setting a number of resources at the same time, the XtVaSetValues() method is better because all of the resources can be set in a single function call. Whichever function you use, the text value is copied into the internals of the widget, and the displayed value is changed accordingly.

If, for whatever reason, you are making multiple changes in a short period of time to the text in a Text widget, you may have problems with visual flashing in the widget. You can solve this problem by calling XmTextDisableRedisplay() to turn off visual updating in the widget. After the call, the appearance of the widget remains unchanged until XmTextEnableRedisplay() is called.

Retrieving the Text

You can access the textual data in a Text widget using XtVaGetValues() or XmTextGetString(). The function XmTextGetString() allocates enough space (using XtMalloc()) for all of the text in the widget and returns a pointer to the allocated data. You can modify the returned data any way you like, and then you must free it using XtFree() when you are done. The code fragment below demonstrates the use of XmTextGetString():

```
char *text;
if (text = XmTextGetString (text_w)) {
    /* manipulate text in whatever way is necessary */
    ...
    /* free text or there will be a memory leak */
```

```
        XtFree (text);
    }
```

XmTextGetString() works with both Text and TextField widgets, while the corresponding TextField routine, XmTextFieldGetString(), only works with TextField widgets. You can also use XmTextGetSubstring() to get a copy of a portion of the text in a Text widget.

The alternative to XmTextGetString() is the Xt function XtVaGetValues(). The Text widget responds to XtVaGetValues() by allocating memory and returning a copy of the text. As a result, this data must be freed after use. This use of the GetValues() method is different from most other resources. For most resources, XtVaGetValues() returns a pointer to internal data that should be treated as read-only data. In order to avoid memory leaks, you need to be sure to free the memory that is allocated by XtVaGetValues() for the XmNvalue resource, as shown in the following code fragment:

```
char *text;
XtVaGetValues (text_w, XmNvalue, &text, NULL);
/* manipulate text in whatever way is necessary */
...
/* free text or there will be a memory leak */
XtFree (text);
```

Getting the value of a Text widget can be an expensive operation if the widget contains a large amount of text. In all situations, whenever text is retrieved from the Text widget with any function, the length of time the data is valid is only guaranteed until the next Xt call into the same Text widget; what any particular call might do to the internal text stream is undefined, and that information will not be reflected in the current character pointer handle you may have.

A Text widget may contain an arbitrarily large amount of text, assuming that there is enough memory on the computer running the client application. The text for a widget is not stored on the X server; only the client computer stores widget-specific information. The server displays a bitmap rendition of what the Text widget chooses to show. The XmNmaxLength resource specifies the upper limit on the number of characters the user can type into a Text widget. The default value of this resource is the largest integer for the particular system, so it is likely that the user's computer will run out of memory before the Text widget's maximum capacity is reached. You can lower the value of the resource to limit the number of characters that the user can input to a particular Text widget.

The Text widget does not use a temporary file to store its data. All of the data resides in memory on the machine, so you cannot use a Text widget to browse or edit a file directly. Instead, you load the contents of a file into a Text widget and allow the user to edit the internal buffer. The application controls when to rewrite files with updated data. An application can also provide an interface that allows the user to control this action. Applications that use Text widgets to edit vital information should make provisions for data

recovery if the system fails or the application terminates unexpectedly. The Text widget does not support this type of recovery.

Single and Multiple Lines

In Example 18-1, the Text widget provides a single-line text entry area that is 20 columns wide; it is shown in Figure 18-2. Both the single-line editing style and the width are default values. The width of each column is based on the font that is used for the text. Since the widget uses the single-line editing style, nothing happens when the user presses RETURN in the widget. If the user types more text than the widget can display, the text scrolls to the left. Since newlines are not interpreted when they are typed by the user, textual data is always a single line.* The user can resize the widget to make it appear large enough to display multiple lines, but this action does not affect the operation of the widget or the way it handles input.

Multiline editing allows the user to enter newlines into a Text widget and provides the capability to edit a large amount of text. The switch from single-line to multiline causes a number of changes in the behavior of the widget. For example, now widget geometry must be considered in order to determine the amount of text that is visible at one time. The Text widget may need to be placed in a ScrolledWindow, so that the user can view all of the text.

Single or multiline editing is controlled through the `XmNeditMode` resource. The value of the resource can be either `XmSINGLE_LINE_EDIT` or `XmMULTI_LINE_EDIT`. While the two editing modes are quite different in concept, it should be quite intuitive when to use the different modes. Single-line text entry areas are commonly used to prompt for file and directory names, short phrases, or single words. They are also useful for command-line entry in applications that were originally based on a tty-style interface. Multiline editing is used for editing files or other large quantities of text.

Scrollable Text

The layout of a multiline Text widget can be difficult to manage, especially if the text is editable by the user. An application needs to decide how many lines of text are displayed, how to handle the layout when the user adds new text, and how to deal with resizing the Text widget. The easiest way to manage an editable multiline Text widget is to create it as part of a ScrolledText compound object. The ScrolledText object is not a widget class in and of itself, but rather a compound object that is composed of a Text widget and a ScrolledWindow widget.

When you create a ScrolledText object, the ScrolledWindow automatically handles scrolling the text in the Text widget. Basically, the two widget classes have hooks and

* It is possible to set `XmNvalue` to a string that contains newline characters in a single-line Text widget, but the interaction with the user is undefined, and the widget produces confusing behavior.

procedures that allow them to cooperate intelligently with each other. As of Motif 1.2, the performance of the ScrolledText object has improved considerably.* In previous releases, scrolling operations could be quite slow when the Text widget contained a large amount of text.

You can create a ScrolledText object using the Motif convenience routine `XmCreateScrolledText()`, which takes the following form:

```
Widget XmCreateScrolledText ( Widget parent, char *name, ArgList arglist,
                              Cardinal argcount)
```

This routine is not a variable-argument list function; it uses the argument-list style of setting resources with the `XtSetArg()` macro.

`XmCreateScrolledText()` creates a ScrolledWindow widget and a Text widget as its child. The routine returns a handle to the Text widget; you can get a handle to the ScrolledWindow using the function `XtParent()`. When you are laying out an application that uses ScrolledText objects, you should be sure to use `XtParent()` to get the ScrolledWindow widget, since that is the widget that you need to position.

For purposes of specifying resources, the ScrolledWindow takes the name of the Text widget with the suffix `SW`. For example, if the name of the Text widget is `name`, its ScrolledWindow parent widget has the name `nameSW`.

If you specify an argument list in a call to `XmCreateScrolledText()`, the resources are set for the Text widget or the ScrolledWindow as appropriate. The routine also sets some resources for the ScrolledWindow so that scrolling is handled automatically. You should be sure to set the `XmNeditMode` resource to `XmMULTI_LINE_EDIT`, since it doesn't make sense to have a single-line Text widget in a ScrolledWindow. If you don't set the resource, the Text widget defaults to single-line editing mode. The behavior of a single-line Text widget (or a TextField widget) in a ScrolledWindow is undefined.

`XmCreateScrolledText()` is adequate for most situations, but you can also create the two widgets separately, as shown in the following code fragment:

```
Widget    scrolled_w, text_w;
Arg       args[6];
int       n = 0;

XtSetArg (args[n], XmNscrollingPolicy, XmAPPLICATION_DEFINED); n++;
XtSetArg (args[n], XmNvisualPolicy, XmVARIABLE);               n++;
XtSetArg (args[n], XmNscrollBarDisplayPolicy, XmSTATIC);       n++;
XtSetArg (args[n], XmNshadowThickness, 0);                     n++;
scrolled_w = XmCreateScrolledWindow (parent, "scrolled_w", args, n);

n = 0;
```

* One unfortunate side-effect of the performance improvement is that subclasses of the Text widget may not work under Motif 1.2, due to the addition of a new data structure.

```
XtSetArg (args[n], XmNeditMode, XmMULTI_LINE_EDIT); n++;
...
text_w = XmCreateText (scrolled_w, "text", args, n);
...
XtManageChild (text);
XtManageChild (scrolled_w);
```

We create the ScrolledWindow widget with the same resource setting that the Motif function uses. Since we are creating the ScrolledWindow ourselves, we can give it our own name. The Text widget itself is created as a child of the ScrolledWindow. In this situation, it is clear that the parent of the ScrolledWindow controls the position of both of the widgets.

This creation method makes the programmer responsible for managing both of the widgets, as shown at the bottom of the fragment. You may also need to handle the case in which the widgets are destroyed. When you call `XmCreateScrolledText()`, the routine installs an `XmNdestroyCallback` on the Text widget that destroys the ScrolledWindow parent. When you create the widgets yourself, you also need to be sure that they are destroyed together, either by destroying them explicitly or installing a callback routine on the Text widget. Unless you are creating and destroying ScrolledText objects dynamically, this issue should not be a concern.

Example 18-2 shows a simple file browser that displays the contents of a file using a ScrolledText object. The user can specify a file by typing a filename in the TextField widget below the *Filename*: prompt. The user can also select a file from the FileSelectionDialog that is popped up by the *Open* entry on the *File* menu. The specified file is displayed immediately in the Text widget.*

Example 18-2 The file_browser.c program

```
/* file_browser.c -- use a ScrolledText object to view the
** contents of arbitrary files chosen by the user from a
** FileSelectionDialog or from a single-line text widget.
*/

#include <X11/Xos.h>
#include <Xm/Text.h>
#include <Xm/TextF.h>
#include <Xm/FileSB.h>
#include <Xm/MainW.h>
#include <Xm/RowColumn.h>
#include <Xm/LabelG.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>

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

* `XtVaAppInitialize()` is considered deprecated in X11R6. `XmStringGetLtoR`() and `XmMainWindowSetAreas`() are considered deprecated in Motif 2.0 and later.

```
Widget         top, main_w, menubar, menu, rc, text_w, file_w, label_w;
XtAppContext   app;
XmString       file, open, exit;
void           read_file(Widget, XtPointer, XtPointer);
void           file_cb(Widget, XtPointer, XtPointer);
Arg            args[10];
int            n;

XtSetLanguageProc (NULL, NULL, NULL);

/* initialize toolkit and create toplevel shell */
top = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv, NULL,
                           sessionShellWidgetClass, NULL);

/* MainWindow for the application -- contains menubar
** and ScrolledText/Prompt/TextField as WorkWindow.
*/
main_w = XmCreateMainWindow (top, "main_w", NULL, 0);

/* Create a simple MenuBar that contains one menu */
file = XmStringCreateLocalized ("File");
menubar = XmVaCreateSimpleMenuBar (main_w, "menubar", XmVaCASCADEBUTTON,
                                   file, 'F', NULL);
XmStringFree (file);

/* Menu is "File" -- callback is file_cb() */
open = XmStringCreateLocalized ("Open...");
exit = XmStringCreateLocalized ("Exit");
menu = XmVaCreateSimplePulldownMenu (menubar, "file_menu", 0, file_cb,
                        XmVaPUSHBUTTON, open, 'O', NULL, NULL,
                        XmVaSEPARATOR,
                        XmVaPUSHBUTTON, exit, 'x', NULL, NULL,
                        NULL);
XmStringFree (open);
XmStringFree (exit);

/* Menubar is done -- manage it */
XtManageChild (menubar);

rc = XmCreateRowColumn (main_w, "work_area", NULL, 0);
n = 0;
XtSetArg (args[n], XmNalignment, XmALIGNMENT_BEGINNING); n++;
label_w = XmCreateLabelGadget (rc, "Filename:", args, n);
XtManageChild (label_w);

file_w = XmCreateTextField (rc, "text_field", NULL, 0);
XtManageChild (file_w);

/* Create ScrolledText -- this is work area for the MainWindow */
n = 0;
XtSetArg (args[n], XmNrows, 12);                    n++;
XtSetArg (args[n], XmNcolumns, 70);                 n++;
XtSetArg (args[n], XmNeditable, False);             n++;
XtSetArg (args[n], XmNeditMode, XmMULTI_LINE_EDIT); n++;
XtSetArg (args[n], XmNcursorPositionVisible, False); n++;
```

```
    text_w = XmCreateScrolledText (rc, "text_w", args, n);
    XtManageChild (text_w);

    /* store text_w as user data in "File" menu for file_cb() callback */
    XtVaSetValues (menu, XmNuserData, text_w, NULL);

    /* add callback for TextField widget passing "text_w" as client data */
    XtAddCallback (file_w, XmNactivateCallback, read_file,
                        (XtPointer) text_w);
    XtManageChild (rc);

    /* Store the filename text widget to ScrolledText object */
    XtVaSetValues (text_w, XmNuserData, file_w, NULL);

    /* Configure the Main Window layout */
    XtVaSetValues (main_w, XmNmenuBar, menubar, XmNworkWindow, rc, NULL);
    XtManageChild (main_w);

    XtRealizeWidget (top);
    XtAppMainLoop (app);
}

void popdown_fsb (Widget fsb, XtPointer client_data, XtPointer call_data)
{
    /* This calls the ChangeManaged() routine of the parent DialogShell
    ** which then internally calls XtPopdown
    */

    XtUnmanageChild (fsb);
}

/* file_cb() -- "File" menu item was selected so popup a
** FileSelectionDialog.
*/
void file_cb (Widget widget, XtPointer client_data, XtPointer call_data)
{
    static Widget  dialog;
    Widget         text_w;
    void           read_file(Widget, XtPointer, XtPointer);
    int            item_no = (int) client_data;

    if (item_no == 1)
        exit (0); /* user chose Exit */

    if (!dialog) {
        Widget menu = XtParent (widget);
        dialog = XmCreateFileSelectionDialog (menu, "file_sb", NULL, 0);

        /* Get the text widget handle stored as "user data" in File menu */
        XtVaGetValues (menu, XmNuserData, &text_w, NULL);
        XtAddCallback (dialog, XmNokCallback, read_file, (XtPointer) text_w);
        XtAddCallback (dialog, XmNcancelCallback, popdown_fsb, NULL);
    }
```

```
    /* The DialogShell parent ChangeManage() calls XtPopup() internally */
    XtManageChild (dialog);
    XMapRaised (XtDisplay (dialog), XtWindow (XtParent (dialog)));
}

/* read_file() -- callback routine when the user selects OK in the
** FileSelection Dialog or presses Return in the single-line text widget.
** The specified file must be a regular file and readable.
** If so, it's contents are displayed in the text_w provided as the
** client_data to this function.
*/
void read_file ( Widget widget, /* file selection or text field widget */
                 XtPointer client_data, XtPointer call_data)
{
    char           *filename, *text;
    struct stat    statb;
    FILE           *fp;
    Widget         file_w;
    Widget         text_w = (Widget) client_data;
    XmFileSelectionBoxCallbackStruct *cbs;

    cbs = (XmFileSelectionBoxCallbackStruct *) call_data;

    if (XtIsSubclass (widget, xmTextFieldWidgetClass)) {
        filename = XmTextFieldGetString (widget);
        file_w = widget; /* this *is* the file_w */
    } else {
        /* file was selected from FileSelectionDialog */
        filename = XmStringUnparse (cbs->value, XmFONTLIST_DEFAULT_TAG,
                                    XmCHARSET_TEXT, XmCHARSET_TEXT, NULL,
                                    0, XmOUTPUT_ALL);
        /* the user data stored the file_w widget in the text_w */
        XtVaGetValues (text_w, XmNuserData, &file_w, NULL);
    }
    if (!filename || !*filename) {/* nothing typed? */
        if (filename)
            XtFree (filename);
        return;
    }
    /* make sure the file is a regular text file and open it */
    if (stat (filename, &statb) == -1 || (statb.st_mode & S_IFMT) != S_IFREG
                || !(fp = fopen (filename, "r"))) {
        if ((statb.st_mode & S_IFMT) == S_IFREG)
            perror (filename); /* send to stderr why we can't read it */
        else
            fprintf (stderr, "%s: not a regular file\n", filename);
        XtFree (filename);
        return;
    }
    /* put the contents of the file in the Text widget by allocating
    ** enough space for the entire file, reading the file into the
    ** allocated space, and using XmTextFieldSetString() to show the file.
    */
    if (!(text = XtMalloc ((unsigned)(statb.st_size + 1)))) {
```

```
            fprintf (stderr, "Can't alloc enough space for %s", filename);
            XtFree (filename);
            (void) fclose (fp);
            return;
        }
        if (!fread (text, sizeof (char), statb.st_size + 1, fp))
            fprintf (stderr, "Warning: may not have read entire file!\n");

        text[statb.st_size] = 0; /* be sure to NULL-terminate */
        /* insert file contents in Text widget */
        XmTextSetString (text_w, text);
        /* make sure text field is up to date */
        if (file_w != widget) {
            /* only necessary if activated from FileSelectionDialog */
            XmTextFieldSetString (file_w, filename);
            XmTextFieldSetCursorPosition (file_w, strlen (filename));
        }
        /* free all allocated space and */
        XtFree (text);
        XtFree (filename);
        (void) fclose (fp);
    }
```

The output of the program is shown in Figure 18-3.

Figure 18-3 Output of the file_browse program

We use the convenience routine `XmCreateScrolledText()` to create a ScrolledText area. We specify that the Text widget displays 12 lines by 70 columns of text by setting the `XmNrows` and `XmNcolumns` resources. These settings are used only at initialization. Once the application is up and running, the user can resize the window and effectively change those dimensions.

The `XmNeditable` resource is set to `False` to prevent the user from editing the contents of the Text widget. Since we do not provide a way to write changes back to the file, we don't want to mislead the user into thinking that the file is editable. Since a non editable

Text widget should not display an insertion cursor, we remove it by setting XmNcursorPositionVisible to False.

The FileSelectionDialog is created and managed when the user selects the *Open* button from the *File* menu. The user can exit the program by selecting the *Exit* button from this menu. The read_file() routine is activated when the user presses the *OK* button in the FileSelectionDialog or enters RETURN in the TextField widget. This function gets the specified file and checks its type. If the file chosen is not a regular file (e.g., if it is a directory, device, tty, etc.) or if it cannot be opened, an error is reported and the function simply returns.

Assuming that the file checks out, its contents are placed in the Text widget. Rather than loading the file by reading each line using a function like fgets(), we allocate enough space to contain the entire file and read it all in with one call to fread(). The text is then loaded into the Text widget using XmTextSetString(). The ScrollBars are updated automatically and the text is positioned so that the beginning of the file is displayed.

Line Wrapping and ScrollBar Placement

In *file_browser.c*, the ScrolledText object has two ScrollBars that are installed automatically. The vertical ScrollBar is needed in case the text exceeds 12 lines; the horizontal ScrollBar is needed in case any of those lines are wider than 70 columns. Most users are accustomed to having Text windows be a fixed width (typically 80 columns), especially if they have ever used an ASCII terminal. However, it can be annoying to have text that is scrollable in the horizontal direction, since you need to see the entire line to read smoothly through a page of text.

The XmNscrollHorizontal resource controls whether or not a horizontal ScrollBar is displayed. If the resource is set to False, the ScrollBar is not displayed, but that does not stop text from being displayed beyond the visible area. In order to have text wrap appropriately, the XmNwordWrap resource must be set to True. When this resource is set, the Text widget breaks lines at spaces, tabs, and newlines. While line breaking is fine for previewing files and other output-only Text widgets, you should not enforce such a policy for Text widgets that are used for text editing, as the user may want to edit wide files.

The XmNscrollVertical resource controls whether or not a vertical ScrollBar is displayed. This resource defaults to True when a Text widget is created as a child of a ScrolledWindow. The XmNscrollLeftSide and XmNscrollTopSide resources take Boolean values that control the location of the ScrollBars within the ScrolledWindow. By default, XmNscrollTopSide is set to False, which causes the ScrollBar to be placed below the ScrolledWindow. The default value of XmNscrollLeftSide depends upon the value of XmNstringDirection. These two resources should not be set by the application, but left to users to specify themselves.

Automatic Resizing

The XmNresizeWidth and XmNresizeHeight resources control whether or not a Text widget should resize itself vertically or horizontally in order to display the entire text stream. Both of the resources default to False. If XmNresizeWidth is set to True and new text is added such that the number of columns needs to grow, the width of the widget grows to contain the new text. Similarly, if XmNresizeHeight is set to True and the number of lines increases, the height of the widget increases so that it can display all of the lines. These resources have no effect in a ScrolledText object, since the ScrollBars are managing the widget's size. Also, if line breaking is active, XmNresizeWidth has no effect.

In most cases, it is not appropriate to set these resources, as it is regarded as poor user-interface design to have a Text widget that dynamically resizes as the text is being edited. It is also impolite for a window to resize itself except as the result of an explicit user action. One example of an acceptable use of these resources involves using a Text widget to display text for a help dialog. In this situation, the Text widget can resize itself silently before it is mapped to the screen, so that by the time it is visible, its size is constant.

Text Positions

A *position* in a Text widget specifies the number of characters from the beginning of the text in the widget, where the first character position is defined as zero (0). All whitespace and newline characters are considered part of the text and are counted as single characters. For example, in Figure 18-3, the insertion cursor in the TextField widget is at position 14. When the user types in a Text widget, the new text is always added at the position of the insertion cursor and the insertion cursor is advanced. If the user does not move the cursor, it is always positioned at the end of the text in the widget.

You can set the position of the insertion cursor explicitly using XmTextSetInsertionPosition(), which takes the following form:

```
void XmTextSetInsertionPosition (Widget text_w, XmTextPosition position)
```

This function is identical to XmTextSetCursorPosition(). The XmTextPosition type is a long value, so it can represent all of the positions in a Text widget.You can get the current cursor position using XmTextGetInsertionPosition() or XmTextGetCursorPosition(). As with most of the Text widget functions, there are corresponding TextField functions for setting and getting the position of the insertion cursor. The TextField routines only work with TextField widgets, while the Text routines work with both Text and TextField widgets.

Example 18-3 shows an application that uses these routines as part of a search operation. The program searches the Text widget for a specified pattern and then positions the insertion cursor so that the pattern is displayed.*

Example 18-3 The search_text.c program

```
/* search_text.c -- demonstrate how to position a cursor at a
** particular location. The position is determined by a pattern
** match search.
*/

#include <Xm/Text.h>
#include <Xm/TextF.h>
#include <Xm/LabelG.h>
#include <Xm/RowColumn.h>
#include <X11/Xos.h>

/* for the index() function */
Widget text_w, search_w, text_output;

main (int argc, char *argv[])
{
    Widget         toplevel, rowcol_v, rowcol_h, label_w;
    XtAppContext   app;
    int            i, n;
    void           search_text(Widget, XtPointer, XtPointer);
    Arg            args[10];

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

    rowcol_v = XmCreateRowColumn (toplevel, "rowcol_v", NULL, 0);
    XtSetArg (args[0], XmNorientation, XmHORIZONTAL);
    rowcol_h = XmCreateRowColumn (rowcol_v, "rowcol_h", args, 1);
    label_w = XmCreateLabelGadget (rowcol_h, "Search Pattern:", NULL, 0);
    XtManageChild (label_w);
    search_w = XmCreateTextField (rowcol_h, "search_text", NULL, 0);
    XtManageChild (search_w);
    XtManageChild (rowcol_h);

    n = 0;
    XtSetArg (args[n], XmNeditable, False);              n++;
    XtSetArg (args[n], XmNcursorPositionVisible, False); n++;
    XtSetArg (args[n], XmNshadowThickness, 0);           n++;
    XtSetArg (args[n], XmNhighlightThickness, 0);        n++;
    text_output = XmCreateText (rowcol_v, "text_output", args, n);
    XtManageChild (text_output);

    n = 0;
    XtSetArg (args[n], XmNrows, 10);                    n++;
    XtSetArg (args[n], XmNcolumns, 80);                 n++;
    XtSetArg (args[n], XmNeditMode, XmMULTI_LINE_EDIT); n++;
    XtSetArg (args[n], XmNscrollHorizontal, False);     n++;
    XtSetArg (args[n], XmNwordWrap, True);              n++;
```

* XtVaAppInitialize() is considered deprecated in X11R6.

```
    text_w = XmCreateScrolledText (rowcol_v, "text_w", args, n);
    XtManageChild (text_w);

    XtAddCallback (search_w, XmNactivateCallback, search_text, NULL);

    XtManageChild (rowcol_v);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

/* search_text() -- called when the user activates the TextField. */
void search_text (Widget widget, XtPointer client_data, XtPointer call_data)
{
    char             *search_pat, *p, *string, buf[32];
    XmTextPosition   pos;
    int              len;
    Boolean          found = False;

    /* get the text that is about to be searched */
    if (!(string = XmTextGetString (text_w)) || !*string) {
        XmTextSetString (text_output, "No text to search.");
        XtFree (string); /* may have been ""; free it */
        return;
    }
    /* get the pattern we're going to search for in the text. */
    if (!(search_pat = XmTextGetString (search_w)) || !*search_pat) {
        XmTextSetString (text_output, "Specify a search pattern.");
        XtFree (string); /* this we know is a string; free it */
        XtFree (search_pat); /* this may be "", XtFree() checks.. */
        return;
    }
    len = strlen (search_pat);
    /* start searching at current cursor position + 1 to find
    ** the -next- occurrence of string. we may be sitting on it.
    */
    pos = XmTextGetCursorPosition (text_w);

    for (p = &string[pos+1]; p = index (p, *search_pat); p++)
        if (!strncmp (p, search_pat, len)) {
            found = True;
            break;
        }

    if (!found) { /* didn't find pattern? */
        /* search from beginning till we've passed "pos" */
        for (p = string;
            (p = index (p, *search_pat)) && p - string <= pos;
            p++)
            if (!strncmp (p, search_pat, len)) {
                found = True;
                break;
            }
    }
```

```
        if (!found)
            XmTextSetString (text_output, "Pattern not found.");
        else {
            pos = (XmTextPosition) (p - string);
            sprintf (buf, "Pattern found at position %ld.", pos);
            XmTextSetString (text_output, buf);
            XmTextSetInsertionPosition (text_w, pos);
        }

        XtFree (string);
        XtFree (search_pat);
    }
```

In this example, the user can search for strings in a ScrolledText, as shown in Figure 18-4.

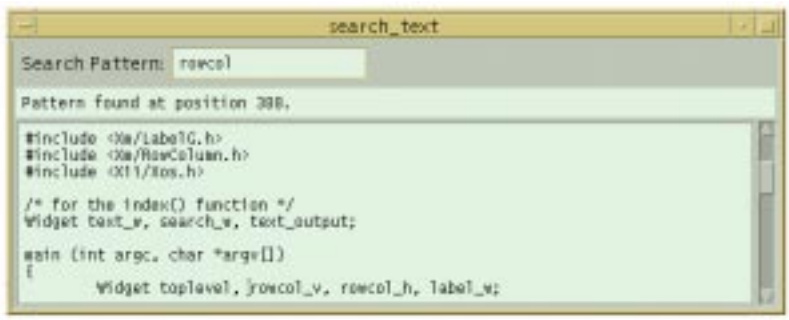


Figure 18-4 Output of the search_text program

This program doesn't provide a way to load a file, so if you want to experiment, you need to type or paste some text into the widget. Once there is some text in the widget, type a string pattern in the *Search Pattern* TextField widget and press RETURN to activate the search. The text is searched starting at the position immediately following the current cursor position. If the search routine reaches the end of the text before it finds the pattern, it resumes searching from the beginning of the text and continues until it finds the pattern or reaches the cursor position. If the routine finds the pattern, it moves the insertion point to that location using `XmTextSetInsertionPosition()`. Otherwise, the routine prints an error message and does not move the cursor.

The `search_text()` routine shown in Example 18-3 searches the text using various string routines. However, in Motif, there is a Text routine that provides the same functionality. `XmTextFindString()` searches a Text widget for a specified string. This routine takes the following form:

```
Boolean XmTextFindString ( Widget            text_w,
                           XmTextPosition    start,
                           char              *string,
                           XmTextDirection   direction,
                           XmTextPosition    *position)
```

The *start* argument specifies the starting position for the search, while *direction* indicates whether the routine searches forward or backward in the text. This parameter can have the value XmTEXT_FORWARD or XmTEXT_BACKWARD. The routine returns True if it finds the string, and in this case, the *position* parameter returns the position where the string starts in the text. If the string is not found, the routine returns False, and the value of *position* is undefined. It is easy to rewrite search_text() to take advantage of XmTextFindString(). Later in this chapter we implement a full text editor and use XmTextFindString() to handle the various search operations.

The text_output widget in *search_text.c* is also a Text widget, even though it looks more like a Label widget. By setting XmNshadowThickness to 0 and XmNeditable to False, we create the Text widget that doesn't look like a normal Text widget, and the user cannot edit the text. We demonstrate this technique not to advocate such usage, but to point out the versatility of this widget class.

If you paste a large amount of text into the main Text widget and search repeatedly for a common pattern, you should notice that the Text widget may scroll automatically to make the specified text visible. This action is controlled by the XmNautoShowCursorPosition resource. This resource has a default value of True, which means that the Text widget adjusts the visible text to make sure that the cursor is always visible. When the resource is set to False, the widget does not scroll to compensate for the cursor's invisibility. This resource also works in single-line Text widgets and TextField widgets; these widgets may scroll their displays horizontally to display the insertion cursor.

It is easy to scroll a Text widget to a particular position in the text stream by setting the cursor position and then calling XmTextShowPosition(). This routine takes the following form:

```
void XmTextShowPosition (Widget text_w, XmTextPosition position)
```

To scroll to the end of the text, you need to scroll to the last position, which can be retrieved using XmTextGetLastPosition(). It is also possible to perform relative scrolling using the function XmTextScroll(), which takes the following form:

```
void XmTextScroll (Widget text_w, int lines)
```

A positive value for lines causes a Text widget to scroll upward by that many lines, while a negative value causes downward scrolling. The Text widget does not have to be a child of ScrolledWindow for this routine to work; the widget simply adjusts the viewable text.

Now that we have a routine that searches for text, the next logical step is to implement a function that performs a search-and-replace operation. Motif makes this task fairly easy by providing the XmTextReplace() routine, which takes the following form:

```
void XmTextReplace ( Widget            text_w,
                     XmTextPosition    from_pos,
                     XmTextPosition    to_pos,
                     char              *value)
```

This function identifies the text to be replaced in the Text widget starting at the position *from_pos* and ending at, but not including, the position *to_pos*. This text is replaced by the text in *value*. If *value* is NULL or an empty string, the text between the two positions is simply deleted. If you want to remove all of the text from the widget, call XmTextSetString() with a NULL string as the text value.

To add search-and-replace functionality to the program in Example 18-3, we need to add a new TextField widget that prompts for the replacement text and provide a callback routine for the widget. Example 18-4 shows the additional code that is necessary.

Example 18-4 The search_and_replace() function

```
Widget text_w, search_w, replace_w, text_output;

main (int argc, char *argv[])
{
    ...
    replace_w = XmCreateTextField (rowcol_h, "replace_text", NULL, 0);
    XtManageChild (replace_w);
    XtAddCallback (replace_w, XmNactivateCallback,
    search_and_replace, NULL);
    ...
}

void search_and_replace (Widget widget, XtPointer client_data,
                         XtPointer call_data)
{
    char             *search_pat, *p, *string, *new_pat, buf[32];
    XmTextPosition   pos;
    int              search_len, pattern_len;
    int              nfound = 0;

    string = XmTextGetString (text_w);
    if (!*string) {
        XmTextSetString (text_output, "No text to search.");
        XtFree (string);
        return;
    }
    search_pat = XmTextGetString (search_w);
    if (!*search_pat) {
        XmTextSetString (text_output, "Specify a search pattern.");
        XtFree (string);
        XtFree (search_pat);
        return;
    }
    new_pat = XmTextGetString (replace_w);
    search_len = strlen (search_pat);
    pattern_len = strlen (new_pat);

    /* start at beginning and search entire Text widget */
    for (p = string; p = index (p, *search_pat); p++)
        if (!strncmp (p, search_pat, search_len)) {
            nfound++;
```

```
                /* get the position where pattern was found */
                pos = (XmTextPosition) (p-string);
                /* replace the text from our position + strlen (new_pat) */
                XmTextReplace (text_w, pos, pos + search_len, new_pat);
                /* "string" has changed -- we must get the new version */
                XtFree (string); /* free the one we had first... */
                string = XmTextGetString (text_w);
                /* continue search for next pattern -after- replacement */
                p = &string[pos + pattern_len];
            }

        if (!nfound)
            strcpy (buf, "Pattern not found.");
        else
            sprintf (buf, "Made %d replacements.", nfound);
        XmTextSetString (text_output, buf);
        XtFree (string);
        XtFree (search_pat);
        XtFree (new_pat);
    }
```

In this routine, the pattern search starts at the beginning of the text and searches all of the text in the widget. We are not interested in the cursor position and do not attempt to move it. The main loop of the function only needs to find the specified pattern and replace each occurrence with the new text. After each call to `XmTextReplace()`, we reread the text, since the old value is no longer valid. As with the `search_text()` routine, we could easily use `XmTextFindString()` to search for the pattern, as we do in the text editor later in this chapter.

Output-only Text

The Text and TextField widgets can be used in an output-only mode by setting the `XmNeditable` resource to `False`. If the user tries to edit the text in a read-only widget, the widget beeps and does not allow the modification. We used an output-only Text widget in our file browsing application.

Our next example addresses a common need for many developers: a method for displaying text messages while an application is running. These messages may include status messages about application actions, as well as error messages from Xlib, Xt, and functions internal to the application. The message area is an important part of the main window of many applications, as discussed in Chapter 4, *The Main Window*. While a message area can be implemented using a Label widget, an output-only ScrolledText object is better suited for use as a message area because the user can scroll back to previous messages.

Example 18-5 shows the `wprint()` function that we wrote to handle displaying messages. The function acts like `printf()` in that it takes variable arguments and understands the standard string formatting characters. The output goes to a ScrolledText widget so the user

can review previous messages. All new text is appended to the end of the output, so it is immediately visible and the user does not have to manually scroll to the end of the display.

Example 18-5 The wprint() function

```
#include <stdio.h>
#include <stdargs.h> /* or <varargs.h> */

/* global variable */
Widget text_output;

main (int argc, char *argv[])
{
    Arg  args[10];
    int  n;
    ....
    /* Create output_text as a ScrolledText window */
    n = 0;
    XtSetArg (args[n], XmNrows, 6);                      n++;
    XtSetArg (args[n], XmNcolumns,  80);                 n++;
    XtSetArg (args[n], XmNeditable, False);              n++;
    XtSetArg (args[n], XmNeditMode, XmMULTI_LINE_EDIT);  n++;
    XtSetArg (args[n], XmNwordWrap, True);               n++;
    XtSetArg (args[n], XmNscrollHorizontal, False);      n++;
    XtSetArg (args[n], XmNcursorPositionVisible, False); n++;
    text_output = XmCreateScrolledText (rowcol, "text_output", args, n);
    XtManageChild (text_output);
    ....
}

/*PRINTFLIKE1*/
void wprint (const char *fmt, ...)
{
    char                    msgbuf[256];
    static XmTextPosition   wpr_position;
    va_list                 data;

    va_start (data, fmt);
    (void) vsprintf (msgbuf, fmt, data);
    va_end (args);

    XmTextInsert (text_output, wpr_position, msgbuf);
    wpr_position = wpr_position + strlen (msgbuf);
    XtVaSetValues (text_output, XmNcursorPosition, wpr_position, NULL);
    XmTextShowPosition (text_output, wpr_position);
}
```

Since the `wprint()` function acts like `printf()`, it takes a variable-length argument list, which requires the inclusion of either <*varargs.h*> or <*stdarg.h*>. `vsprintf()` is a varargs version of `sprintf()` that exists on most modern UNIX machines.* If your machine does

* System V has `vsprintf()`, as does SunOS, but older Ultrix and BSD machines may use `_doprnt()`.

not have vsprintf(), you can use _doprnt(): consult your system documentation for details if you do not have the standard C varargs package available. Whether using vsprintf() or _doprnt(), both of these functions consume all of the arguments in the list and leave the result in msgbuf.

Now that we have the complete string in msgbuf, we can append it to the existing text in the Text widget. We keep track of the end of text_output with wpr_position. Each time msgbuf is concatenated to the end of the text, the value of wpr_position is incremented appropriately. The new text is added using the convenience routine XmTextInsert(), which takes the following form:

```
void XmTextInsert (Widget text_w, XmTextPosition position, char *string)
```

The function simply inserts the given text at the specified position. Finally, we call XmTextShowPosition() to make the end position visible within the Text widget. This routine may cause the Text widget to adjust its text so that the new text is visible, as a convenience to the user so that he does not have to scroll the window to view new messages.

The routines in Example 18-6 show how wprint() can be used to reset the error handling functions for Xlib and Xt so that the messages are printed in a Text widget rather than to stderr.

Example 18-6 The x_error() and xt_error() routines

```
extern void wprint(const char *fmt, ...);
static void x_error (Display *dpy, XErrorEvent *err_event)
{
    char buf[256];
    XGetErrorText (dpy, err_event->error_code, buf, (sizeof buf));
    wprint ("X Error: <%s>\n", buf);
}

static void xt_error (char *message)
{
    wprint ("Xt Error: %s\n", message);
}

main (int argc, char *argv[])
{
    XtAppContext app;
    ...
    /* catch Xt errors */
    XtAppSetErrorHandler (app, xt_error);
    XtAppSetWarningHandler (app, xt_error);
    /* and Xlib errors */
    XSetErrorHandler (x_error);
    ...
}
```

Using the functions `XtAppSetErrorHandler()`, `XtAppSetWarningHandler()`, and `XSetErrorHandler()`, we send all X-related error messages to a Text widget through `wprint()`. You can also use `wprint()` to send any application-specific messages to the ScrolledText area.

Text Clipboard Functions

Both the Text widget and the TextField widget have convenience routines that support communication with the clipboard. Using these functions, you can implement the standard cut, copy, and paste functionality, as well as support communication with other windows or applications on the desktop. If you are not familiar with the clipboard and how it works, see Chapter 21, *The Clipboard*. Briefly, the clipboard is one of three transient locations where arbitrary data such as text can be stored so that other windows or applications can copy the data. For the Text widget, we are only interested in copying textual data and providing visual feedback within the widget. The Text widget can send and receive data from all three of the locations, depending on the interface style that you are using.

As described earlier in this chapter, the user typically selects text by pressing the first mouse button and dragging the pointer across the text. When text is selected, it is rendered in reverse video and automatically copied into the primary selection. Now the user can paste text from the primary selection into any Text widget on the desktop by pressing the middle mouse button. The insertion cursor is moved to the location of the button press, and the data is automatically copied into the Text widget at this position. This functionality works by default within the Text widget. However, the actions operate on the primary selection, not the clipboard selection. Furthermore, the actions only allow you to copy data to and from the selection, not cut it or clear it.

To provide these features, most applications provide other user-interface controls, such as a PulldownMenu and appropriate menu items, that call Text widget clipboard routines. These routines store text on the clipboard. They also allow the user to move text between the clipboard and the primary selection, as well as between windows that are interested only in the clipboard selection. Typical menu entries include *Cut*, *Copy*, *Paste*, and *Clear*. Example 18-7 demonstrates these common editing actions. The application creates a MenuBar with an *Edit* PulldownMenu that contains actions that operate on the Text widget*

Example 18-7 The cut_paste.c program

```
/* cut_paste.c -- demonstrate the text functions that handle
** clipboard operations. These functions are convenience routines
** that relieve the programmer of the need to use clipboard functions.
** The functionality of these routines already exists in the Text
```

* `XtVaAppInitialize`() is considered deprecated in X11R6.

```
** widget, yet it is common to place such features in the interface
** via the MenuBar's "Edit" pulldown menu.
*/

#include <Xm/Text.h>
#include <Xm/LabelG.h>
#include <Xm/PushBG.h>
#include <Xm/RowColumn.h>
#include <Xm/MainW.h>

Widget text_w, text_output;

main (int argc, char *argv[])
{
    Widget         toplevel, main_w, menubar, rowcol_v;
    XtAppContext   app;
    void           cut_paste(Widget, XtPointer, XtPointer);
    XmString       label, cut, clear, copy, paste;
    Arg            args[10];
    int            n;

    XtSetLanguageProc (NULL, NULL, NULL);
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv, NULL,
                                    sessionShellWidgetClass, NULL);
    main_w = XmCreateMainWindow (toplevel, "main_w", NULL, 0);
    /* Create a simple MenuBar that contains a single menu */
    label = XmStringCreateLocalized ("Edit");
    menubar = XmVaCreateSimpleMenuBar (main_w, "menubar",
    XmVaCASCADEBUTTON, label, 'E', NULL);
    XmStringFree (label);
    cut = XmStringCreateLocalized ("Cut");
    /* create a simple    */
    copy = XmStringCreateLocalized ("Copy");
    /* pulldown menu that */
    clear = XmStringCreateLocalized ("Clear");
    /* has these menu     */
    paste = XmStringCreateLocalized ("Paste");
    /* items in it.       */
    XmVaCreateSimplePulldownMenu (menubar, "edit_menu", 0, cut_paste,
                                  XmVaPUSHBUTTON, cut, 't', NULL, NULL,
                                  XmVaPUSHBUTTON, copy, 'C', NULL, NULL,
                                  XmVaPUSHBUTTON, paste, 'P', NULL, NULL,
                                  XmVaSEPARATOR,
                                  XmVaPUSHBUTTON, clear, 'l', NULL, NULL,
                                  NULL);
    XmStringFree (cut);
    XmStringFree (clear);
    XmStringFree (copy);
    XmStringFree (paste);
    XtManageChild (menubar);

    /* create a standard vertical RowColumn... */
    rowcol_v = XmCreateRowColumn (main_w, "rowcol_v", NULL, 0);
```

```
    n = 0;
    XtSetArg (args[n], XmNeditable, False);              n++;
    XtSetArg (args[n], XmNcursorPositionVisible, False); n++;
    XtSetArg (args[n], XmNshadowThickness, False);       n++;
    XtSetArg (args[n], XmNhighlightThickness, 0);        n++;
    text_output = XmCreateText (rowcol_v, "text_output", args, n);
    XtManageChild (text_output);

    n = 0;
    XtSetArg (args[n], XmNrows,      10);               n++;
    XtSetArg (args[n], XmNcolumns,   80);               n++;
    XtSetArg (args[n], XmNeditMode, XmMULTI_LINE_EDIT); n++;
    XtSetArg (args[n], XmNscrollHorizontal, False);     n++;
    XtSetArg (args[n], XmNwordWrap, True);              n++;
    text_w = XmCreateScrolledText (rowcol_v, "text_w", args, n);
    XtManageChild (text_w);

    XtManageChild (rowcol_v);
    XtManageChild (main_w);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

/* cut_paste() -- the callback routine for the items in the edit menu */
void cut_paste (Widget widget, XtPointer client_data, XtPointer call_data)
{
    Boolean  result = True;
    int      reason = (int) client_data;
    XEvent   *event = ((XmPushButtonCallbackStruct *) call_data)->event;
    Time     when;

    XmTextSetString (text_output, NULL);   /* clear message area */
    if (event != NULL) {
        switch (event->type) {
            case ButtonRelease : when = event->xbutton.time; break;
            case KeyRelease    : when = event->xkey.time; break;
            default            : when = CurrentTime; break;
        }
    }
    switch (reason) {
        case 0 : result = XmTextCut (text_w, when); break;
        case 1 : result = XmTextCopy (text_w, when); break;
        case 2 : result = XmTextPaste (text_w); /* FALLTHROUGH */
        case 3 : XmTextClearSelection (text_w, when); break;
    }
    if (result == False)
        XmTextSetString (text_output, "There is no selection.");
    else
        XmTextSetString (text_output, NULL)
}
```

The application creates a MainWindow widget, so that it can contain the MenuBar. The MenuBar and the PulldownMenu are created using their respective convenience routines,

as described in Chapter 4, *The Main Window*, and Chapter 20, *Interacting With the Window Manager*. The output of the program is shown in Figure 18-5.

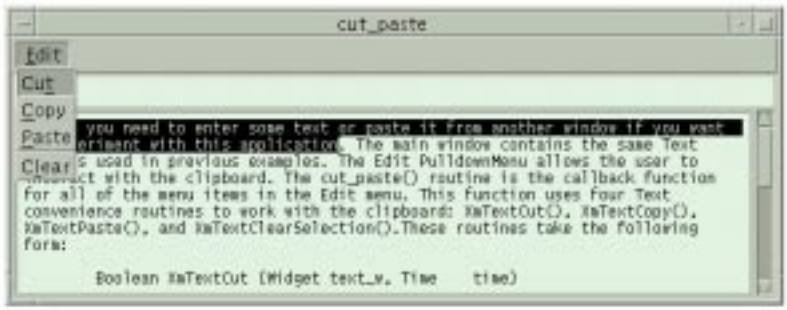


Figure 18-5 Output of the cut_paste program

Again, you need to enter some text or paste it from another window if you want to experiment with this application. The main window contains the same Text widgets used in previous examples. The *Edit* PulldownMenu allows the user to interact with the clipboard. The `cut_paste()` routine is the callback function for all of the menu items in the *Edit* menu. This function uses four Text routines to work with the clipboard: `XmTextCut()`, `XmTextCopy()`, `XmTextPaste()`, and `XmTextClearSelection()`. These routines take the following form:

```
Boolean XmTextCut (Widget text_w, Time time)
Boolean XmTextCopy (Widget text_w, Time time)
Boolean XmTextPaste (Widget text_w)
void XmTextClearSelection (Widget text_w, Time time)
```

`XmTextCopy()` copies the text that is selected in the Text widget and places it on the clipboard. `XmTextCut()` is similar to `XmTextCopy()`, except that the Text widget that owns the selection is instructed to delete the text once it has been copied to the clipboard.* The *time* parameters should not be set to `CurrentTime` to avoid race conditions with other clipboard operations that may be occurring at the same time. Since the clipboard routines are called by menu item callback routines, you can use the `time` field of the `XEvent` that is passed in the callback structure, as we do in Example 18-7. Both `XmTextCopy()` and `XmTextCut()` return `True` if the operation succeeds. `False` may be returned if there is no selected text or an error occurs in attempting to communicate with the clipboard.

`XmTextPaste()` gets the current selection from the clipboard and inserts it at the location of the insertion cursor. If there is some selected text in the Text widget, that text is replaced

* The deletion is handled by sending a DELETE protocol request to the window holding the selection. This protocol is not the same as the WM_DELETE protocol, which indicates that a window is being deleted. See Chapter 20, *Interacting With the Window Manager*, for more information on window manager protocols.

by the selection from the clipboard. XmTextPaste() returns True if there is a selection on the clipboard that can be retrieved.

XmTextClearSelection() deselects the text selection in the Text widget. If there is no selected text, nothing happens. The routine does not provide any feedback or return any value. Any text that is held on the clipboard or in a selection property remains.

One additional convenience routine that operates on the selection is XmTextRemove(). This function is like XmTextCut(), in that it removes the selected text from a Text widget, but it does not place the text on the clipboard.

Getting the Selection

You can get the selected text from a Text widget using XmTextGetSelection(), which takes the following form:

```
char *XmTextGetSelection (Widget text_w)
```

This routine returns allocated data that contains the selected text. This text must be freed using XtFree() when you are through using it. The routine returns NULL if there is no text selected in the Text widget.

XmTextGetSelectionPosition() provides information about the selected text in a Text widget. This routine takes the following form:

```
Boolean XmTextGetSelectionPosition ( Widget          text_w,
                                     XmTextPosition  *left,
                                     XmTextPosition  *right)
```

If XmTextGetSelectionPosition() returns True, the values for *left* and *right* specify the boundaries of the selected text. If the routine returns False, the widget does not contain any selected text, and the values for *left* and *right* are undefined.

Modifying the Selection Mechanisms

The Text widget supports multi-clicking techniques for selecting increasingly large chunks of text. The default multi-clicking actions in the Text widget are shown in Table 18-1.

Table 18-1 Default Selection Actions for Multiple Clicks

User Action	Text Widget Action
Single click	Resets insertion cursor to position
Double click	Selects a word (bounded by whitespace)
Triple click	Selects a line (bounded by newlines)
Quadruple click	Selects all of the text

These default actions can be modified using the XmNselectionArray and XmNselectionArrayCount resources. The XmNselectionArray resource specifies an

array of XmTextScanType values, where XmTextScanType is an enumerated type defined as follows:

```
typedef enum {
    XmSELECT_POSITION, XmSELECT_WHITESPACE,* XmSELECT_WORD, XmSELECT_LINE,
    XmSELECT_PARAGRAPH, XmSELECT_ALL
} XmTextScanType;
```

Each successive button click in a Text widget selects the text according to the corresponding item in the array. The default array is defined as follows:

```
static XmTextScanType sarray[] = {
    XmSELECT_POSITION, XmSELECT_WORD, XmSELECT_LINE, XmSELECT_ALL
};
```

You should keep the items in the array in ascending order, so as not to confuse the user. The following code fragment shows an acceptable change to the array:

```
static XmTextScanType sarray[] = {
    XmSELECT_POSITION, XmSELECT_WORD, XmSELECT_LINE,
    XmSELECT_PARAGRAPH, XmSELECT_ALL
};
...
XtVaSetValues (text_w, XmNselectionArray, selectionArray,
                            XmNselectionArrayCount, 5, NULL);
```

The maximum time interval between button clicks in a multi-click action is specified by the multiClickTime resource. This resource is maintained by the X server and set for all applications; it is not a Motif resource. The value of the resource can be retrieved using XtGetMultiClickTime() and changed with XtSetMultiClickTime(). For more discussion on this value, see Chapter 12, *Labels and Buttons*.

The XmNselectThreshold resource can be used to modify the behavior of click-and-drag actions. This resource specifies the number of pixels that the user must move the pointer before a character can be selected. The default value is 5, which means that the user must move the mouse at least 5 pixels before the Text widget decides whether or not to select a character. This threshold is used throughout a selection operation to determine when characters are added or deleted from the selection. If you are using an extremely large font, you may want to increase the value of this resource to cut down on the number of calculations that are necessary to determine if a character should be added or deleted from the selection.

A Text Editor

Before we describe the Text widget callback routines, we are going to present an example that combines all the information covered so far. The example is a full-featured text editor

* XmSELECT_WHITESPACE works in the same way as XmSELECT_WORD.

built from the examples presented so far in this chapter. You should recognize most of the code in the example; the code that you don't recognize should be understandable from the context in which it is used. The output of the program is shown in Figure 18-6; the code is shown in Example 18-8.*

Figure 18-6 Output of the editor program

Example 18-8 The editor.c program

```
/* editor.c -- create a full-blown Motif editor application complete
** with a menubar, facilities to read and write files, text search
** and replace, clipboard support and so forth.
*/

#include <Xm/Text.h>
#include <Xm/TextF.h>
#include <Xm/LabelG.h>
#include <Xm/PushBG.h>
#include <Xm/RowColumn.h>
#include <Xm/MainW.h>
#include <Xm/Form.h>
#include <Xm/FileSB.h>
#include <X11/Xos.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>

Widget text_edit, search_text, replace_text, text_output;

#define FILE_OPEN          0
#define FILE_SAVE          1
#define FILE_EXIT          2
#define EDIT_CUT           0
#define EDIT_COPY          1
```

* XtVaAppInitialize() is considered deprecated in X11R6. XmStringGetLtoR() is deprecated in Motif 2.0 and later. XmRepTypeInstallTearOffModelConverter() is deprecated in Motif 2.0: the converter is installed internally.

```
#define EDIT_PASTE         2
#define EDIT_CLEAR         3
#define SEARCH_FIND_NEXT   0
#define SEARCH_SHOW_ALL    1
#define SEARCH_REPLACE     2
#define SEARCH_CLEAR       3

main (int argc, char *argv[])
{
    XtAppContext    app_context;
    Widget          toplevel, main_window, menubar, form, search_panel,
                    label_w;
    void            file_cb(Widget, XtPointer, XtPointer);
    void            edit_cb(Widget, XtPointer, XtPointer);
    void            search_cb(Widget, XtPointer, XtPointer);
    Arg             args[10];
    int             n = 0;
    XmString        open, save, exit, exit_acc, file, edit, cut, clear,
                    copy, paste, search, next, find, replace;

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

    main_window = XmCreateMainWindow (toplevel, "main_window", NULL, 0);
    /* Create a simple MenuBar that contains three menus */
    file = XmStringCreateLocalized ("File");
    edit = XmStringCreateLocalized ("Edit");
    search = XmStringCreateLocalized ("Search");
    menubar = XmVaCreateSimpleMenuBar (main_window, "menubar",
                        XmVaCASCADEBUTTON, file, 'F',
                        XmVaCASCADEBUTTON, edit, 'E',
                        XmVaCASCADEBUTTON, search, 'S',
                        NULL);
    XmStringFree (file);
    XmStringFree (edit);
    XmStringFree (search);

    /* First menu is the File menu -- callback is file_cb() */
    open = XmStringCreateLocalized ("Open...");
    save = XmStringCreateLocalized ("Save...");
    exit = XmStringCreateLocalized ("Exit");
    exit_acc = XmStringCreateLocalized ("Ctrl+C");
    XmVaCreateSimplePulldownMenu (menubar, "file_menu", 0, file_cb,
                        XmVaPUSHBUTTON, open, 'O', NULL, NULL,
                        XmVaPUSHBUTTON, save, 'S', NULL, NULL,
                        XmVaSEPARATOR,
                        XmVaPUSHBUTTON, exit, 'x', "Ctrl<Key>c", exit_acc,
                        NULL);
    XmStringFree (open);
    XmStringFree (save);
    XmStringFree (exit);
    XmStringFree (exit_acc);
```

```
/*...create the "Edit" menu -- callback is edit_cb() */
cut = XmStringCreateLocalized ("Cut");
copy = XmStringCreateLocalized ("Copy");
clear = XmStringCreateLocalized ("Clear");
paste = XmStringCreateLocalized ("Paste");
XmVaCreateSimplePulldownMenu (menubar, "edit_menu", 1, edit_cb,
                    XmVaPUSHBUTTON, cut, 't', NULL, NULL,
                    XmVaPUSHBUTTON, copy, 'C', NULL, NULL,
                    XmVaPUSHBUTTON, paste, 'P', NULL, NULL,
                    XmVaSEPARATOR,
                    XmVaPUSHBUTTON, clear, 'l', NULL, NULL,
                    NULL);
XmStringFree (cut);
XmStringFree (copy);
XmStringFree (paste);

/* create the "Search" menu -- callback is search_cb() */
next = XmStringCreateLocalized ("Find Next");
find = XmStringCreateLocalized ("Show All");
replace = XmStringCreateLocalized ("Replace Text");
XmVaCreateSimplePulldownMenu (menubar, "search_menu", 2, search_cb,
                    XmVaPUSHBUTTON, next, 'N', NULL, NULL,
                    XmVaPUSHBUTTON, find, 'A', NULL, NULL,
                    XmVaPUSHBUTTON, replace, 'R', NULL, NULL,
                    XmVaSEPARATOR,
                    XmVaPUSHBUTTON, clear, 'C', NULL, NULL,
                    NULL);
XmStringFree (next);
XmStringFree (find);
XmStringFree (replace);
XmStringFree (clear);
XtManageChild (menubar);

/* create a form work are */
form = XmCreateForm (main_window, “form", NULL, 0);

/* create horizontal RowColumn inside the form */
n = 0;
XtSetArg (args[n], XmNorientation, XmHORIZONTAL);      n++;
XtSetArg (args[n], XmNpacking, XmPACK_TIGHT);          n++;
XtSetArg (args[n], XmNtopAttachment, XmATTACH_FORM);   n++;
XtSetArg (args[n], XmNleftAttachment, XmATTACH_FORM);  n++;
XtSetArg (args[n], XmNrightAttachment, XmATTACH_FORM); n++;
search_panel = XmCreateRowColumn (form, "search_panel", args, n);

/* Create two TextField widgets with Labels... */
label_w = XmCreateLabelGadget (search_panel, "Search Pattern:",
                                NULL, 0);
XtManageChild (label_w);
search_text = XmCreateTextField (search_panel, "search_text", NULL, 0);
XtManageChild (search_text);
label_w = XmCreateLabelGadget (search_panel, "Replace Pattern:"
                                NULL, 0);
XtManageChild (label_w);
```

```
    replace_text = XmCreateTextField (search_panel, "replace_text",
                                      NULL, 0);
    XtManageChild (replace_text);
    XtManageChild (search_panel);

    n = 0;
    XtSetArg (args[n], XmNeditable, False);                 n++;
    XtSetArg (args[n], XmNcursorPositionVisible, False);    n++;
    XtSetArg (args[n], XmNshadowThickness, 0);              n++;
    XtSetArg (args[n], XmNleftAttachment, XmATTACH_FORM);   n++;
    XtSetArg (args[n], XmNrightAttachment, XmATTACH_FORM);  n++;
    XtSetArg (args[n], XmNbottomAttachment, XmATTACH_FORM); n++;
    text_output = XmCreateTextField (form, "text_output", args, n);
    XtManageChild (text_output);

    n = 0;
    XtSetArg (args[n], XmNrows, 10);                          n++;
    XtSetArg (args[n], XmNcolumns, 80);                       n++;
    XtSetArg (args[n], XmNeditMode, XmMULTI_LINE_EDIT);       n++;
    XtSetArg (args[n], XmNtopAttachment, XmATTACH_WIDGET);    n++;
    XtSetArg (args[n], XmNtopWidget, search_panel);           n++;
    XtSetArg (args[n], XmNleftAttachment, XmATTACH_FORM);     n++;
    XtSetArg (args[n], XmNrightAttachment, XmATTACH_FORM);    n++;
    XtSetArg (args[n], XmNbottomAttachment, XmATTACH_WIDGET); n++;
    XtSetArg (args[n], XmNbottomWidget, text_output);         n++;
    text_edit = XmCreateScrolledText (form, "text_edit", args, n);
    XtManageChild (text_edit);
    XtManageChild (form);
    XtManageChild (main_window);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app_context);
}

/* file_select_cb() -- callback routine for "OK" button in
** FileSelectionDialogs.
*/
void file_select_cb (Widget dialog, XtPointer client_data, XtPointer call_data)
{
    char         buf[256], *filename, *text;
    struct stat  statb;
    long         len;
    FILE         *fp;
    int          reason = (int) client_data;
    XmFileSelectionBoxCallbackStruct *cbs;

    cbs = (XmFileSelectionBoxCallbackStruct *) call_data;

    if (!(filename = XmStringUnparse (cbs->value, XmFONTLIST_DEFAULT_TAG,
                                XmCHARSET_TEXT, XmCHARSET_TEXT, NULL, 0,
                                XmOUTPUT_ALL)))
        return; /* must have been an internal error */
    if (*filename == NULL) {
        XtFree (filename);
        XBell (XtDisplay (text_edit), 50);
```

```
        XmTextSetString (text_output, "Choose a file.");
        return; /* nothing typed */
    }
    if (reason == FILE_SAVE) {
        if (!(fp = fopen (filename, "w"))) {
            perror (filename);
            sprintf (buf, "Can't save to %s.", filename);
            XmTextSetString (text_output, buf);
            XtFree (filename);
            return;
        }
        /* saving -- get text from Text widget... */
        text = XmTextGetString (text_edit);
        len = XmTextGetLastPosition (text_edit);
        /* write it to file (check for error) */
        if (fwrite (text, sizeof (char), len, fp) != len)
            strcpy (buf, "Warning: did not write entire file!");
        else {
            /* make sure a newline terminates file */
            if (text[len-1] != '\n')
                fputc ('\n', fp);
            sprintf (buf, "Saved %ld bytes to %s.", len, filename);
        }
    }
    else {/* reason == FILE_OPEN */
        /* make sure the file is a regular text file and open it */
        if (stat (filename, &statb) == -1 || (statb.st_mode & S_IFMT) != S_IFREG
                        || !(fp = fopen (filename, "r"))) {
            perror (filename);
            sprintf (buf, "Can't read %s.", filename);
            XmTextSetString (text_output, buf);
            XtFree (filename);
            return;
        }
        /* put the contents of the file in the Text widget by
        ** allocating enough space for the entire file, reading the
        ** file into the space, and using XmTextSetString() to show
        ** the file.
        */
        len = statb.st_size;
        if (!(text = XtMalloc ((unsigned)(len+1)))) /* +1 for NULL */
            sprintf (buf, "%s: XtMalloc (%ld) failed", len, filename);
        else {
            if (fread (text, sizeof (char), len, fp) != len)
                sprintf (buf, "Warning: did not read entire file!");
            else
                sprintf (buf, "Loaded %ld bytes from %s.", len, filename);
            text[len] = 0; /* NULL-terminate */
            XmTextSetString (text_edit, text);
        }
    }
    XmTextSetString (text_output, buf);
    /* purge output message */
    /* free all allocated space. */
```

```
    XtFree (text);
    XtFree (filename);
    (void) fclose (fp);
    XtUnmanageChild (dialog);
}

/* popdown_cb() -- callback routine for "Cancel" button. */
void popdown_cb (Widget w, XtPointer client_data, XtPointer call_data)
{
    XtUnmanageChild (w);
}

/* file_cb() -- a menu item from the "File" pulldown menu was selected */
void file_cb (Widget w, XtPointer client_data, XtPointer call_data)
{
    static Widget  open_dialog, save_dialog;
    Widget         dialog = NULL;
    XmString       button, title;
    int            reason = (int) client_data;

    if (reason == FILE_EXIT)
        exit (0);

    XmTextSetString (text_output, NULL);   /* clear message area */

    if (reason == FILE_OPEN && open_dialog)
        dialog = open_dialog;
    else if (reason == FILE_SAVE && save_dialog)
        dialog = save_dialog;
    if (dialog) {
        XtManageChild (dialog);
        /* make sure that dialog is raised to top of window stack */
        XMapRaised (XtDisplay (dialog), XtWindow (XtParent (dialog)));
        return;
    }
    dialog = XmCreateFileSelectionDialog (text_edit, "Files", NULL, 0);
    XtAddCallback (dialog, XmNcancelCallback, popdown_cb, NULL);
    XtAddCallback (dialog, XmNokCallback, file_select_cb,
                   (XtPointer) reason);
    if (reason == FILE_OPEN) {
        button = XmStringCreateLocalized ("Open");
        title = XmStringCreateLocalized ("Open File");
        open_dialog = dialog;
    }
    else {/* reason == FILE_SAVE */
        button = XmStringCreateLocalized ("Save");
        title = XmStringCreateLocalized ("Save File");
        save_dialog = dialog;
    }
    XtVaSetValues (dialog, XmNokLabelString, button, XmNdialogTitle, title,
                   NULL);
    XmStringFree (button);
    XmStringFree (title);
    XtManageChild (dialog);
}
```

```
/* search_cb() -- a menu item from the "Search" pulldown menu selected */
void search_cb (Widget w, XtPointer client_data, XtPointer call_data)
{
    char             *search_pat, *p, *string, *new_pat, buf[256];
    XmTextPosition   pos = 0;
    int              len, nfound = 0;
    int              search_len, pattern_len;
    int              reason = (int) client_data;
    Boolean          found = False;

    XmTextSetString (text_output, NULL);   /* clear message area */
    if (reason == SEARCH_CLEAR) {
        pos = XmTextGetLastPosition (text_edit);
        XmTextSetHighlight (text_edit, 0, pos, XmHIGHLIGHT_NORMAL);
        return;
    }
    if (!(string = XmTextGetString (text_edit)) || !*string) {
        XmTextSetString (text_output, "No text to search.");
        return;
    }
    if (!(search_pat = XmTextGetString (search_text)) ||
                  !*search_pat) {
        XmTextSetString (text_output, "Specify a search pattern.");
        XtFree (string);
        return;
    }
    new_pat = XmTextGetString (replace_text);
    search_len = strlen (search_pat);
    pattern_len = strlen (new_pat);

    if (reason == SEARCH_FIND_NEXT) {
        pos = XmTextGetCursorPosition (text_edit) + 1;
        found = XmTextFindString (text_edit, pos, search_pat,
                                  XmTEXT_FORWARD, &pos);

        if (!found)
            found = XmTextFindString (text_edit, 0, search_pat,
                                      XmTEXT_FORWARD, &pos);
        if (found)
            nfound++;
    }
    else {/* reason == SEARCH_SHOW_ALL || reason == SEARCH_REPLACE */
        do {
            found = XmTextFindString (text_edit, pos, search_pat,
                                      XmTEXT_FORWARD, &pos);
            if (found) {
                nfound++;
                if (reason == SEARCH_SHOW_ALL)
                    XmTextSetHighlight (text_edit, pos,
                                        pos + search_len,
                                        XmHIGHLIGHT_SELECTED);
                else
                    XmTextReplace (text_edit, pos, pos + search_len,
                                        new_pat);
```

```
                pos++;
            }
        } while (found);
    }

    if (nfound == 0)
        XmTextSetString (text_output, "Pattern not found.");
    else {
        switch (reason) {
            case SEARCH_FIND_NEXT :
                sprintf (buf, "Pattern found at position %ld.", pos);
                XmTextSetInsertionPosition (text_edit, pos);
                break;
            case SEARCH_SHOW_ALL :
                sprintf (buf, "Found %d occurrences.", nfound);
                break;
            case SEARCH_REPLACE :
                sprintf (buf, "Made %d replacements.", nfound);
        }

        XmTextSetString (text_output, buf);
    }

    XtFree (string);
    XtFree (search_pat);
    XtFree (new_pat);
}

/* edit_cb() -- the callback routine for the items in the edit menu */
void edit_cb (Widget widget, XtPointer client_data, XtPointer call_data)
{
    Boolean      result = True;
    int          reason = (int) client_data;
    XEvent       *event;
    Time         when;

    event = ((XmPushButtonCallbackStruct *) call_data)->event;

    XmTextSetString (text_output, NULL); /* clear message area */
    if (event != NULL && reason == EDIT_CUT || reason == EDIT_COPY ||
                         reason == EDIT_CLEAR) {
        switch (event->type) {
            case ButtonRelease : when = event->xbutton.time; break;
            case KeyRelease    : when = event->xkey.time; break;
            default            : when = CurrentTime; break;
        }
    }
    switch (reason) {
        case EDIT_CUT    : result = XmTextCut (text_edit, when);
                           break;
        case EDIT_COPY   : result = XmTextCopy (text_edit, when);
                           break;
        case EDIT_PASTE  : result = XmTextPaste (text_edit);
                           /* FALLTHROUGH */
```

```
            case EDIT_CLEAR  : XmTextClearSelection (text_edit, when);
                               break;
        }

        if (result == False)
            XmTextSetString (text_output, "There is no selection.");
    }
```

Text Callbacks

The Text and TextField widgets use callback routines in the same way as other Motif widgets. The widgets provide callbacks for a number of different purposes, such as text modification, activation, and selection ownership. Some of the routines, such as those that monitor keyboard input, may be invoked rather frequently. In the next few sections, we introduce several of the callback routines for the widgets.

The Activation Callback

We begin by exploring the callback routine that is most commonly used for single-line Text widgets and TextField widgets. This callback is the `XmNactivateCallback`, which is invoked when the user presses RETURN in a TextField widget or a single-line Text widget. The callback is not called for multiline Text widgets. The callback routine for an `XmNactivateCallback` receives the common `XmAnyCallbackStruct` as the `call_data` parameter to the function. The callback reason is always `XmCR_ACTIVATE`. Example 18-9 shows a callback function for some TextField widgets.*

Example 18-9 The text_box.c program

```
/* text_box.c -- demonstrate simple use of XmNactivateCallback
** for TextField widgets. Create a rowcolumn that has rows of Form
** widgets, each containing a Label and a Text widget. When
** the user presses Return, print the value of the text widget
** and move the focus to the next text widget.
*/

#include <Xm/TextF.h>
#include <Xm/LabelG.h>
#include <Xm/Form.h>
#include <Xm/RowColumn.h>

char *labels[] = { "Name:", "Address:", "City:", "State:", "Zip:" };

main (int argc, char *argv[])
{
    Widget         toplevel, text_w, form, rowcol, label_w;
    XtAppContext   app;
    int            i;
```

* `XtVaAppInitialize`() is considered deprecated in X11R6.

```
    void           print_result(Widget, XtPointer, XtPointer);
    Arg            args[8];
    int            n;

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

    rowcol = XmCreateRowColumn (toplevel, "rowcol", NULL, 0);

    for (i = 0; i < XtNumber (labels); i++) {
        n = 0;
        XtSetArg (args[n], XmNfractionBase, 10);       n++;
        XtSetArg (args[n], XmNnavigationType, XmNONE); n++;
        form = XmCreateForm (rowcol, "form", args, n);

        n = 0;
        XtSetArg (args[n], XmNtopAttachment, XmATTACH_FORM);       n++;
        XtSetArg (args[n], XmNbottomAttachment, XmATTACH_FORM);    n++;
        XtSetArg (args[n], XmNleftAttachment, XmATTACH_FORM);      n++;
        XtSetArg (args[n], XmNrightAttachment, XmATTACH_POSITION); n++;
        XtSetArg (args[n], XmNrightPosition, 3);                   n++;
        XtSetArg (args[n], XmNalignment, XmALIGNMENT_END);         n++;
        XtSetArg (args[n], XmNnavigationType, XmNONE);             n++;
        label_w = XmCreateLabelGadget (form, labels[i], args, n);
        XtManageChild (label_w);

        n = 0;
        XtSetArg (args[n], XmNtraversalOn, True);                 n++;
        XtSetArg (args[n], XmNleftAttachment, XmATTACH_POSITION); n++;
        XtSetArg (args[n], XmNleftPosition, 4);                   n++;
        XtSetArg (args[n], XmNrightAttachment, XmATTACH_FORM);    n++;
        XtSetArg (args[n], XmNnavigationType, XmTAB_GROUP);       n++;
        text_w = XmCreateTextField (form, "text_w", args, n);
        XtManageChild (text_w);

        /* When user hits return, print the label+value of text_w */
        XtAddCallback (text_w, XmNactivateCallback, print_result,
                       (XtPointer) labels[i]);
        XtManageChild (form);
    }

    XtManageChild (rowcol);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

/* print_result() -- callback for when the user hits return in the
** TextField widget.
*/

void print_result (Widget text_w, XtPointer client_data,
                   XtPointer call_data)
{
```

```
        char *value = XmTextFieldGetString (text_w);
        char *label = (char *) client_data;

        printf ("%s %s\n", label, value);
        XtFree (value);

        XmProcessTraversal (text_w, XmTRAVERSE_NEXT_TAB_GROUP);
    }
```

The program displays a data form using a RowColumn widget that manages several rows of Form widgets. Each Form contains a Label and a TextField widget, as shown in Figure 18-7.

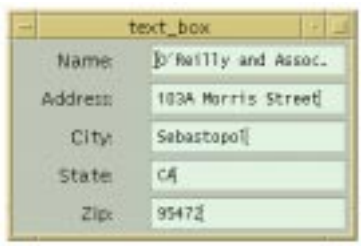


Figure 18-7 Output of the text_box program

When the user enters a value for a field and presses RETURN, the `print_result()` callback routine is invoked. The routine prints the value of the field and advances the keyboard focus to the next widget using `XmProcessTraversal()`. This function takes a widget and a traversal direction as its two parameters. We use the `XmTRAVERSE_NEXT_TAB_GROUP` direction because each TextField widget is a tab group in and of itself, so we need to move to the next tab group, rather than to the next item in the same tab group. See the *Keyboard Traversal* Section of Chapter 8 for more information on tab groups.

When a single-line Text widget or a TextField widget is used as part of a predefined Motif dialog, the `XmNactivateCallback` for the widget is automatically hooked up to the *OK* button in the dialog. As a result, the same callback is called when the user presses RETURN in the widget or when the user selects the *OK* button. This convenience can confuse an unsuspecting programmer who may find that his callback is being invoked twice. It is also possible to overestimate what the Motif toolkit is going to do and expect a callback to be invoked when it isn't. The point is to be sure to verify that these callbacks are getting called at the appropriate times. See Chapter 6, *Selection Dialogs*, for examples of this feature in SelectionDialogs, PromptDialogs, and CommandDialogs.

Text Modification Callbacks

In this section, we discuss the callback routines that can be used to monitor and control text modification. Monitoring occurs both when the user types into a Text widget and when the text is changed using a convenience routine such as `XmTextInsert()`. These callbacks

work for both single-line and multiline Text widgets, as well as TextField widgets. Since the text in a widget is modified by each keystroke, the modification callbacks are invoked quite frequently.

There are two callbacks for text modification: XmNmodifyVerifyCallback is called before the text is modified, and XmNvalueChangedCallback is called after the text has been changed. Depending on the needs of an application, either or both callbacks may be used on the same widget. You should never call XtVaSetValues() in one of these callbacks on the widget that is being modified because the state of the widget is unstable during these callbacks. Avoid adding or deleting callbacks or changing resources, especially the XmNvalue resource, in a callback routine. If a recursive loop occurs, you may get very unpredictable results.

Installing an XmNmodifyVerifyCallback function is useful when you need to monitor or change the user's input before it actually gets inserted into a Text widget. In Example 18-10, we demonstrate using this callback to convert text to uppercase.*

Example 18-10 The allcaps.c program

```
/* allcaps.c -- demonstrate the XmNmodifyVerifyCallback for
** Text widgets by using one to convert all typed input to
** capital letters.
*/

#include <Xm/Text.h>
#include <Xm/LabelG.h>
#include <Xm/RowColumn.h>
#include <ctype.h>

void allcaps(Widget, XtPointer, XtPointer);

main (int argc, char *argv[])
{
    Widget        toplevel, text_w, rowcol, label_w;
    XtAppContext  app;
    Arg           args[2];

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

    XtSetArg (args[0], XmNorientation, XmHORIZONTAL);
    rowcol = XmCreateRowColumn (toplevel, "rowcol", args, 1);

    label_w = XmCreateLabelGadget (rowcol, "Enter Text:", NULL, 0);
    XtManageChild (label_w);
    text_w = XmCreateText (rowcol, "text_w", NULL, 0);
    XtManageChild (text_w);
```

* XtVaAppInitialize() is considered deprecated in X11R6.

```
    XtAddCallback (text_w, XmNmodifyVerifyCallback, allcaps, NULL);

    XtManageChild (rowcol);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

/* allcaps() -- convert inserted text to capital letters. */
void allcaps (Widget text_w, XtPointer client_data, XtPointer call_data)
{
    int len;
    XmTextVerifyCallbackStruct *cbs =
                            (XmTextVerifyCallbackStruct *) call_data;

    if (cbs->text->ptr == NULL)
        return;
    /* convert all input to upper-case if necessary */
    for (len = 0; len < cbs->text->length; len++)
        if (islower (cbs->text->ptr[len]))
            cbs->text->ptr[len] = toupper (cbs->text->ptr[len]);
}
```

The program creates a RowColumn widget that contains a Label and a Text widget, as shown in Figure 18-8.

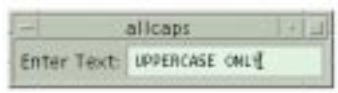


Figure 18-8 Output of the allcaps program

The Text widget uses the `allcaps()` routine as its `XmNmodifyVerifyCallback` function. The routine is actually quite simple, but there are a lot of details to examine. The `call_data` parameter to the function is of type `XmTextVerifyCallbackStruct`. This data structure provides information about the modification that may be done to the text. The data structure is defined as follows:

```
typedef struct {
    int             reason;
    XEvent          *event;
    Boolean         doit;
    XmTextPosition  currInsert, newInsert;
    XmTextPosition  startPos, endPos;
    XmTextBlock     text;
} XmTextVerifyCallbackStruct;
```

With an `XmNmodifyVerifyCallback`, the `reason` field has the value `XmCR_MODIFYING_TEXT_VALUE`. The `event` field contains the `XEvent` that caused the callback to be invoked; this field is `NULL` if the modification is programmatic, for example, if the text is changed through a convenience function*. The values for `currInsert` and `newInsert` are always the same for a modification callback. These fields specify the

location of the insertion cursor, so they are only different for the XmNmotionVerifyCallback when the user moves the insertion point.

The values for startPos and endPos indicate the range of text that is affected by the modification. For insertion, these values are always the same. However, for text deletion or replacement, the values specify the beginning and end of the text about to be deleted. For example, if the user selects some text and presses the BACKSPACE key, the startPos and endPos values indicate the boundaries of the text about to be deleted. We discuss text deletion in detail in an up coming section.

The text field points to a data structure that describes the text about to be added to the widget. The field is a pointer of type XmTextBlock, which is defined as follows:

```
typedef struct {
    char           *ptr;
    int            length;
    XmTextFormat   format;
} XmTextBlockRec, *XmTextBlock;
```

The text being added is accessible through ptr; it is dynamically allocated using XtMalloc() for each callback invocation. The ptr field is not NULL-terminated, so you should not use strlen() or strcpy() to copy the data. The length is stored in the length field, so if you want to copy the text, you should use strncpy(). If the user is deleting text, length is 0. While ptr should also be NULL in this case, the field isn't always set this way, so you shouldn't rely on it. The format field specifies the width of the text characters and can have the value FMT8BIT or FMT16BIT.

Let's review the simple case of adding new text, as demonstrated in Example 18-10. When new text is inserted into the Text widget, the values for currInsert, newInsert, startPos, and endPos all have the same value, which is the position in the widget where the new text will be added. Since the new text has not yet been added to the value of the widget, the application can change the value of ptr in the text block. In the allcaps() routine, we modify the input to be all capital letters by looping through the valid bytes in the ptr field of the text block that is going to be added, as shown in the following fragment:

```
for (len = 0; len < cbs->text->length; len++)
    if (islower (cbs->text->ptr[len]))
        cbs->text->ptr[len] = toupper (cbs->text->ptr[len]);
```

The islower() and toupper() macros are found in the *<ctype.h>* header file.

Since allcaps() is called each time new text is added to the widget, you might wonder how length can ever be more than one. If the user pastes a block of text into the widget,

* There is a persistent bug in the toolkit such that if the user pastes characters into a Text widget using the mouse, the event field is also NULL. It is therefore not possible to differentiate between a programmatic and mouse change of the Text contents by inspecting only the information contained within the callback data.

the entire block is added at once, so ptr points to that text, and length specifies the amount of text. Our loop handles both single-character typing and text-block paste operations.

Preventing Text Modification

Example 18-10 demonstrates how an application can modify the text that is entered by a user before it is displayed. An application may also want to filter the new text and prevent certain characters from being inserted. The easiest way to prevent a text modification is to set the doit field in the XmTextVerifyCallbackStruct to False. When the modification callback routine returns, the Text widget checks this field. If it has been set to False, the widget discards the new text, and the widget is left unmodified.

When a text modification is vetoed, the Text widget can sound the console bell to provide audio feedback informing the user that the input has been rejected. This action is dependent on the value of the XmNverifyBell resource. The default value is based on the value of the XmNaudibleWarning resource of the VendorShell, so it is set to True by default. You should allow a user to set this resource in a resource file, so he can turn off error notification if he doesn't want it. If you hard-code the resource value, users cannot control this feature. You should provide documentation with your application that explains how to set this resource or provide a way to set the value from the application.

Example 18-11 demonstrates a modification callback routine that filters input and prevents certain characters from being entered. The check_zip() routine would be used as the XmNmodifyVerifyCallback for a Text widget that prompts for a ZIP code. We want the user to type only digits; all other input should be ignored. We also want to keep the user from typing a string that is longer than five digits.

Example 18-11 The check_zip() routine

```
/* check_zip() -- limit the user to entering a ZIP code. */
void check_zip (Widget text_w, XtPointer client_data,
                    XtPointer call_data)
{
    XmTextVerifyCallbackStruct *cbs =
                        (XmTextVerifyCallbackStruct *) call_data;
    int len = XmTextGetLastPosition (text_w);

    if (cbs->startPos < cbs->currInsert) /* backspace */
        return;

    if (len == 5) {
        cbs->doit = False;
        return;
    }

    /* check that the new additions won't put us over 5 */
    if (len + cbs->text->length > 5) {
        cbs->text->ptr[5 - len] = 0;
        cbs->text->length = strlen (cbs->text->ptr);
```

```
        }

        for (len = 0; len < cbs->text->length; len++) {
            /* make sure all additions are digits. */
            if (!isdigit (cbs->text->ptr[len])) {
                /* not a digit-- move all chars down one and
                ** decrement cbs->text->length.
                */
                int i;

                for (i = len; (i+1) < cbs->text->length; i++)
                    cbs->text->ptr[i] = cbs->text->ptr[i+1];

                cbs->text->length--;
                len--;
            }
        }

        if (cbs->text->length == 0)
            cbs->doit = False;
    }
```

The first thing we do in `check_zip()` is to see if the user is backspacing, in which case we simply return. If text is not being deleted, then new text is definitely being added. Since the length of the current text is not available in the callback structure, we call `XmTextGetLastPosition()` to determine it. If the string is already five digits long, we don't want to add more digits, so we set `doit` to `False` and return.

Otherwise, we loop through the length of the new text and check for characters that are not digits. If any exist, we remove them by shifting all of the characters that follow down one place, overwriting the undesirable character. If we loop through all of the characters and find that none of them are digits, the length ends up being zero, so we set `doit` to `False`.

Handling Text Deletion

A modification callback can determine if the user is backspacing or deleting a large block of text by checking to see if `startPos` is less than `currInsert`. Alternatively, the routine could check to see if `text->length` is `0`. For backspacing, the values differ by one. If the user selects a large block of text and deletes the selection, the `XmNmodifyVerifyCallback` is invoked once to delete the text and may be invoked a second time if the user has typed new text to replace the selected text.

Our next example program demonstrates how to process character deletions in a text modification callback. Example 18-12 creates a single-line Text widget that prompts the user for a password. We don't provide any encryption for the password; we simply mask what the user is typing by displaying an asterisk (`*`) for each character. The actual text is stored in a separate internal variable. The challenge for this application is to capture the input text, store it internally, and modify the output, even for backspacing.*

Example 18-12 The password.c program

```
/* password.c -- prompt for a password. All input looks like
** a series of *'s. Store the actual data typed by the user in
** an internal variable. Don't allow paste operations. Handle
** backspacing by deleting all text from insertion point to the
** end of text.
*/

#include <Xm/TextF.h>
#include <Xm/LabelG.h>
#include <Xm/RowColumn.h>
#include <ctype.h>

void check_passwd(Widget, XtPointer, XtPointer);
char *passwd = (char *) 0; /* store user-typed passwd here. */

main (int argc, char *argv[])
{
    Widget         toplevel, text_w, label_w, rowcol;
    XtAppContext   app;
    Arg            args[2];

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

    XtSetArg (args[0], XmNorientation, XmHORIZONTAL);
    rowcol = XmCreateRowColumn (toplevel, "rowcol", args, 1);

    label_w = XmCreateLabelGadget (rowcol, "Password:", NULL, 0);
    XtManageChild (label_w);

    text_w = XmCreateTextField (rowcol, "text_w", NULL, 0);
    XtManageChild (text_w);

    XtAddCallback (text_w, XmNmodifyVerifyCallback, check_passwd, NULL);
    XtAddCallback (text_w, XmNactivateCallback, check_passwd, NULL);

    XtManageChild (rowcol);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

/* check_passwd() -- handle the input of a password. */
void check_passwd (Widget text_w, XtPointer client_data,
                   XtPointer call_data)
{
    char *new;
    int  len;
    XmTextVerifyCallbackStruct *cbs =
```

* `XtVaAppInitialize()` is considered deprecated in X11R6.

```
                                (XmTextVerifyCallbackStruct *) call_data;

        if (cbs->reason == XmCR_ACTIVATE) {
            printf ("Password: %s\n", passwd);
            return;
        }

        if (cbs->startPos < cbs->currInsert) {/* backspace */
            cbs->endPos = strlen (passwd); /* delete from here to end */
            passwd[cbs->startPos] = 0;
            /* backspace--terminate */
            return;
        }

        if (cbs->text->length > 1) {
            cbs->doit = False;   /* don't allow "paste" operations */
            return;              /* make the user *type* the password! */
        }

        new = XtMalloc (cbs->endPos + 2); /* new char + NULL terminator */

        if (passwd) {
            strcpy (new, passwd);
            XtFree (passwd);
        } else
            new[0] = NULL;

        passwd = new;
        strncat (passwd, cbs->text->ptr, cbs->text->length);
        passwd[cbs->endPos + cbs->text->length] = 0;

        for (len = 0; len < cbs->text->length; len++)
            cbs->text->ptr[len] = '*';
    }
```

As you can see in Figure 18-9, the Text widget only displays asterisks, no matter what the user has typed.

Figure 18-9 Output of the password program

We use the `check_passwd()` function for both the `XmNactivateCallback` and the `XmNmodifyVerifyCallback` callbacks. When the user presses RETURN, the routine prints what has been typed to `stdout`. If the user is not backspacing through the text, we know we can add the new text to `passwd`, which is the internal variable we use to store the text. Once the new text has been copied, we convert it into asterisks, so that the user cannot see what has been typed.

We need to handle two different cases for deletion. If the insertion cursor is at the end of the typed string and the user backspaces, we simply allow the action. If the user clicks somewhere in the middle of the string and then backspaces, we delete all of the characters from that point in the string to the end, since the user cannot see the characters that he is deleting.

To handle the different forms of text deletion, we test to see if `startPos` is less than `currInsert`. Since `startPos` and `endPos` specify the range of text that is being deleted, we can change these values and effectively delete more text than the user originally intended. By setting `endPos` to the string length of the internal variable `passwd`, we handle both of the cases that we just described. If we had wanted to, we could also have set `startPos` to `0` and deleted all of the text.

Extending Text Modification

We can expand on the ZIP code example that we used for filtering non-digits from typed input by providing an input field for an area code and phone number. The format for a US phone number is as follows:

```
123-456-7890
```

We want to filter out all non-digits for a phone number, but we also want to add the dash character (`-`) automatically as it is needed. For example, after the user enters three digits, the Text widget should automatically insert a dash, so that the next character expected from the user is still a digit. Similarly, when the user backspaces and deletes a dash character, the widget should delete the preceding digit as well. Table 18-2 shows how the interaction should work.

Table 18-2 Phone Number Input Interaction

User Types	Text Widget Displays
4	4
1	41
5	415-
4	415-4
BACKSPACE	415-
BACKSPACE	41

We can continue to use the same type of algorithm that we used in `check_zip()` to filter digits, and we can use some of the code from `check_passwd()` to handle backspacing. The only remaining problem is adding the necessary dash characters. Since we are using US phone numbers, we know that the dashes should occur after the third and seventh digits. Therefore, when `currInsert` is either 2 or 6, the new digit should be added first, followed by the dash. Example 18-13 shows the program that implements this functionality.*

Example 18-13 The prompt_phone.c program

```
/* prompt_phone.c -- a complex problem for XmNmodifyVerifyCallback.
** Prompt for a phone number by filtering digits only from input.
** Don't allow paste operations and handle backspacing.
*/

#include <Xm/Text.h>
#include <Xm/LabelG.h>
#include <Xm/RowColumn.h>
#include <ctype.h>

void check_phone(Widget, XtPointer, XtPointer);

main (int argc, char *argv[])
{
    Widget         toplevel, text_w, label_w, rowcol;
    XtAppContext   app;
    Arg            args[2];

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

    XtSetArg (args[0], XmNorientation, XmHORIZONTAL);
    rowcol = XmCreateRowColumn (toplevel, "rowcol", args, 1);

    label_w = XmCreateLabelGadget (rowcol, "Phone Number:", NULL, 0);
    XtManageChild (label_w);

    text_w = XmCreateText (rowcol, "text_w", NULL, 0);
    XtManageChild (text_w);

    XtAddCallback (text_w, XmNmodifyVerifyCallback, check_phone, NULL);

    XtManageChild (rowcol);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

/* check_phone() -- handle phone number input. */
void check_phone (Widget text_w, XtPointer client_data,
                  XtPointer call_data)
{
    char c;
    int len = XmTextGetLastPosition (text_w);
    XmTextVerifyCallbackStruct *cbs =
                                (XmTextVerifyCallbackStruct *) call_data;

    /* no backspacing or typing in the middle of string */
```

* XtVaAppInitialize() is considered deprecated in X11R6. Note that this program does not work using a TextField: due to a bug, deleting the dash character paints the text contents blank.

```
        if (cbs->currInsert < len) {
            cbs->doit = False;
            return;
        }

        if (cbs->text->length == 0) {/* backspace */
            if (cbs->startPos == 3 || cbs->startPos == 7)
                cbs->startPos--;
            /* delete the hyphen too */
            return;
        }

        if (cbs->text->length > 1) {/* don't allow clipboard copies */
            cbs->doit = False;
            return;
        }

        /* don't allow non-digits or let the input exceed 12 chars */
        if (!isdigit (c = cbs->text->ptr[0]) || len >= 12)
            cbs->doit = False;
        else if (len == 2 || len == 6) {
            cbs->text->ptr = XtRealloc (cbs->text->ptr, 2);
            cbs->text->length = 2;
            cbs->text->ptr[0] = c;
            cbs->text->ptr[1] = '-';
        }
    }
```

There are a couple of ways that you could think to add the dashes. One way would be to use the `XmNvalueChangedCallback` to keep track of the phone number after it has been entered and then use `XmTextInsert()` to add the dashes when appropriate. The problem with this approach is that `XmTextInsert()` activates the `XmNmodifyVerifyCallback` function again, so the dash would be subject to the input filtering.

As a result, the only way to handle the situation is to actually add the dashes in the `XmNmodifyVerifyCallback` routine at the same time the digits are added. This approach involves modifying the `ptr` and `length` fields of the `XmTextBlock` structure in the `XmTextVerifyCallbackStruct`. The `check_phone()` routine checks the current length of the phone number. If it is either two or six characters long, the routine reallocates `ptr` to hold two characters, adds the dash, and increments `length` to account for the dash.

When the Text widget adds the digit and the dash, it positions the insertion cursor at the end of the new text. Although we haven't demonstrated its use, the `XmNvalueChangedCallback` is useful when you need to keep track of the changes in a Text widget, but you don't need to monitor or change the input before it is displayed. This callback is invoked after the text has been modified in any way, which means that it is called for each insertion and deletion. The `call_data` parameter to the routine is of type `XmAnyCallbackStruct`; the `reason` field is always `XmCR_VALUE_CHANGED`.

The check_phone() routine is fairly simple, in that it only allows text insertions and deletions that occur when the insertion cursor is at the end of the text. While it is possible to handle modifications in the middle of the text, the code quickly becomes a large bowl of spaghetti. We do not allow clipboard copies of more than one character at a time for the same reason. Our routine is sufficient for demonstration purposes, but for a real application, you should handle these cases.

The Cursor Movement Callback

The XmNmotionVerifyCallback can be used to monitor the position of the insertion cursor. This callback is invoked when the user moves the location cursor using the mouse or the arrow keys, when the user drags the mouse or multi-clicks to extend the text selection, or when the application calls a Text widget function that moves the cursor or adds, deletes, or replaces text. However, if the cursor does not move as a result of a function being called, the callback is not invoked.The XmNmotionVerifyCallback allows an application to intercept and prevent cursor movement.

The XmNmotionVerifyCallback uses the XmTextVerifyCallbackStruct as its callback structure, just like the XmNmodifyVerifyCallback. However, for motion callbacks, the reason is XmCR_MOVING_INSERT_CURSOR and the startPos, endPos, and text fields are invalid. The doit field can be set to False to reject requests to reposition the insertion cursor.

If the cursor motion occurs as a result of a user action, the event field should point to an XEvent structure describing the action that caused the cursor position to be modified. When the cursor moves as a result of an application action, the field should be set to NULL. However, the event field is currently set to NULL regardless of what caused the cursor motion. This bug makes it impossible to tell the difference between a cursor motion performed by the user and one caused by the application.

We can use the XmNmotionVerifyCallback to tie up a loose end in *prompt_phone.c*. To make the text verification simpler, we don't want to allow the user to move the insertion cursor except by entering digits or backspacing. Example 18-14 shows a new version of the check_phone() routine that prevents cursor movement.

Example 18-14 The new check_phone() routine

```
main (int argc, char *argv[])
{
    Widget text_w;
    ...
    XtAddCallback (text_w, XmNmotionVerifyCallback, check_phone, NULL);
    ...
}

/* check_phone() -- handle phone number input. */
void check_phone (Widget text_w, XtPointer client_data,
```

```
                    XtPointer call_data)
{
    char c;
    int len = XmTextGetLastPosition (text_w);
    XmTextVerifyCallbackStruct *cbs =
                            (XmTextVerifyCallbackStruct *) call_data;

    if (cbs->reason == XmCR_MOVING_INSERT_CURSOR) {
        if (cbs->newInsert != len)
            cbs->doit = False;
        return;
    }

    /* no backspacing or typing in the middle of string */
    if (cbs->currInsert < len) {
        cbs->doit = False;
        return;
        }
    ...
}
```

We check the value of `newInsert` against the length of the current string to determine whether or not the intended cursor position is at the end of the text string. If it is not, we set `doit` to `False` to prevent the cursor movement. The `XmNmotionVerifyCallback` function can also be used to monitor pointer dragging for text selections.

Focus Callbacks

The `XmNfocusCallback` and `XmNlosingFocusCallback` callback routines can be used to monitor when a Text widget gains and loses the keyboard focus. A Text widget can receive the input focus if the user intentionally shifts the focus to the widget or if the application moves the focus using `XmProcessTraversal()`. When a widget gains the input focus and the insertion cursor is not visible, we can make it visible and cause the widget to automatically scroll to the current cursor location by installing an `XmNfocusCallback` routine that calls `XmTextShowCursorPosition()`, as shown in the following code fragment:

```
{
    Widget text_w;
    void gain_focus(Widget, XtPointer, XtPointer);
    ...
    text_w = XmCreateScrolledText(...);
    XtAddCallback (text_w, XmNfocusCallback, gain_focus, NULL);
    ...
}

void gain_focus (Widget text_w, XtPointer client_data,
                    XtPointer call_data)
{
    XmTextShowCursorPosition (text_w, XmTextGetCursorPosition (text_w));
}
```

The XmNfocusCallback is passed a callback structure of type XmAnyCallbackStruct with the callback reason set to XmCR_FOCUS.

The XmNlosingFocusCallback callback can be used to monitor when the Text widget loses its focus. The callback structure passed to the callback function is an XmTextVerifyCallbackStruct. All of the fields except the text field are valid, and the reason field is set to XmCR_LOSING_FOCUS.

Text Widget Internationalization

In Motif, the Text and TextField widgets support internationalized input and output. The internationalization capabilities of the widgets are layered on top of the functionality originally provided in X11R5, which is based on the ANSI-C locale model. An internationalized application uses a library that reads a locale database at runtime to get information about the user's language environment. An application that uses the X Toolkit establishes its language environment (or locale) by registering a language procedure using XtSetLanguageProc(), as explained in Chapter 2, *The Motif Programming Model*. See Volume 4, *X Toolkit Intrinsics Programming Manual* for more information on the localization of an Xt-based application.

Text Representation

One of the important characteristics of a locale is the encoding used to represent the character set for the locale. A character set is simply a set of characters, while an encoding is a numeric representation of these characters. A charset (not the same as a character set) is an encoding in which all of the characters use the same number of bits. The Latin-1 charset (ISO8859-1) defines an encoding for all of the characters used in Western languages. However, not all languages can be represented by a single charset. Japanese text commonly contains words written using the Latin alphabet, as well as phonetic characters from the *katakana* and *hirigana* alphabets, and ideographic *kanji* characters. Each of these character sets has its own charset. The phonetic and Latin charsets are 8-bits wide, while the ideographic charset is 16-bits wide. Since the charsets must be combined into a single encoding for Japanese text, the encoding uses shift sequences to specify the character set for each character in a string.

When an encoding contains shift sequences and characters of non uniform width, strings can still be stored in a standard NULL-terminated array of characters; this representation is known as a *multibyte string*. Strings can also be stored using a *wide-character* type (wchar_t in ANSI-C) in which each character has a fixed size and occupies one array element. ANSI-C provides functions that convert between multibyte and wide-character strings and the text output routines in X support both types of strings. Multibyte strings are usually more compact than wide-character strings, but wide-character strings are easier to work with. If an internationalized application performs any text manipulation, it must take

care to handle all strings properly. Fortunately, many applications can do internationalized text input and output without performing any manipulations on the text.

Multibyte strings are NULL-terminated, while there is no single convention for the termination of wide-character strings. The following C string-handling routines are safe to use with multibyte strings: strcat(), strcmp(), strcpy(), strlen(), and strncmp(). The string comparison routines are only useful to check for byte-for-byte equality; use strcoll() to compare strings for sorting. None of the C string-handling routines work with wide-character strings.

Multibyte strings can be written to a file or an output stream. If the terminal is operating in the current locale, printing a multibyte string to stdout or stderr causes the correct text to be displayed. Multibyte strings can also be read from a file or the stdin input stream. If the file is encoded in the current locale, or the terminal is operating in the current locale, the strings that are read are meaningful. For a more complete description of working with multibyte and wide-character strings, see Volume 1, *Xlib Programming Manual*.

The Motif Text and TextField widgets provide two resources for specifying their textual data: XmNvalue andXmNvalueWcs. The XmNvalue resource specifies the text string as a char * value, so it can be used to set the value of the widget to a multibyte string. XmNvalueWcs specifies the string as a wchar_t * value, so it is used to set the value to a wide-character string. This resource cannot be specified in a resource file. If XmNvalue and XmNvalueWcs are both defined, the value of XmNvalueWcs takes precedence.

Regardless of which resource you set, the widgets store the text internally as a multibyte string. The widgets take care of converting between multibyte strings and wide-character strings when necessary. As a result, you can set the text string using the XmNvalue resource and retrieve it with XtVaGetValues() using the XmNvalueWcs resource.

The Text widget provides the following convenience routines for manipulating the text value as a wide-character string:

```
Boolean     XmTextFindStringWcs ( Widget             widget,
                                  XmTextPosition     start,
                                  wchar_t            *wc,
                                  XmTextDirection    dir,
                                  XmTextPosition     *pos)
wchar_t     *XmTextGetSelectionWcs (Widget widget)
wchar_t     *XmTextGetStringWcs (Widget widget)
int         XmTextGetSubstringWcs ( Widget           widget,
                                    XmTextPosition   start,
                                    int              num_chars,
                                    int              buf_size,
                                    wchar_t          *buffer)
void        XmTextInsertWcs ( Widget           widget,
                              XmTextPosition   position,
                              wchar_t          *wc)
void        XmTextReplaceWcs ( Widget           widget,
                               XmTextPosition   from,
```

```
                                XmTextPosition   to,
                                wchar_t          *wc)
void       XmTextSetStringWcs (Widget widget, wchar_t *wc)
```

These routines work for both Text and TextField widgets. The TextField also provides corresponding functions that only work with TextField widgets. All of these routines function identically to their regular character string counterparts, except that they take or return wide-character string values. If you have specified the text string using XmNvalue, you can still use the wide-character string routines because they handle any necessary string conversions. For more information on the different wide-character routines, see Volume 6B, *Motif Reference Manual*.

The widgets also provide a wide-character version of the text modification callback, XmNmodifyVerifyCallbackWcs. This callback is invoked before the value of the widget is modified, so an application can use it to monitor changes in the widget. The callback is passed a callback structure of type XmTextVerifyCallbackStructWcs, which is defined as follows:

```
typedef struct {
    int              reason;
    XEvent           *event;
    Boolean          doit;
    XmTextPosition   currInsert, newInsert;
    XmTextPosition   startPos, endPos;
    XmTextBlockWcs   text;
} XmTextVerifyCallbackStructWcs;
```

With this structure the reason field has the value XmCR_MODIFYING_TEXT_VALUE. All of the fields have the same meaning as the fields in the regular XmTextVerifyCallbackStruct, except that the text field is a pointer of type XmTextBlockWcs. This structure is defined as follows:

```
typedef struct {
    wchar_t      *wcsptr;
    int          length;
} XmTextBlockRecWcs, *XmTextBlockWcs;
```

If callback routines are registered for both the XmNmodifyVerifyCallback and the XmNmodifyVerifyCallbackWcs, the routines for the XmNmodifyVerifyCallback are invoked first. The resulting data, which may have been modified, is passed to the XmNmodifyVerifyCallbackWcs routines.

Text Output

The Text and TextField widgets do not use compound strings, so their text output functionality is based directly on Xlib's internationalized text output capabilities. To support languages that use multiple charsets, X11R5 introduced the XFontSet abstraction for its text output routines. An XFontSet contains all of the fonts necessary to display text in the current locale. The new text output routines work with font sets, so they can render

text for locales that require multiple charsets. See Volume 1, *Xlib Programming Manual*, for more information on internationalized text output.

Each of the widgets has a XmNrenderTable resource for specifying the font that it uses.* Since the widgets do not use compound strings, they cannot use font list tags to display text using different fonts as described in Chapter 25, *Compound Strings*. However, the render table can specify a rendition which contains a font set, so the widgets can display text using multiple character sets in a locale that requires them. The widgets pick a font by searching the render table for a rendition that has the tag XmFONTLIST_DEFAULT_TAG. If the search finds such a rendition that contains a font set, it is used. Otherwise, the widgets use the first font set specified in the font list. If the render table does not contain a font set, the first font is used. If you specify a rendition entry with the tag XmFONTLIST_DEFAULT_TAG, make sure that it is appropriate for the encoding of the current locale.†

Text Input

Converting user keystrokes into text in the encoding of the current locale is the most difficult task of internationalization. An internationalized program cannot assume any particular mapping between keystrokes and input characters, since it must run in any locale on a single workstation, using a single keyboard. The mapping between keystrokes and Japanese characters is very different and much more complex than the mapping between keystrokes and Latin characters, for example. When there are more characters in the codeset of a locale than there are keys on a keyboard, some sort of *input method* is required for mapping between multiple keystrokes and input characters.

All of the characters for English can be entered using the standard keyboard; the SHIFT key makes it possible to enter both lower case and upper case letters as well as the number and punctuation characters. For many European languages, the most common accented characters may appear directly on a keyboard, but there are still a number of other characters that cannot be entered with any single shifted or unshifted keystroke. In these cases, the input method is typically implemented in the keyboard hardware using a special key that puts the keyboard in “compose” mode in which one or more of the following keystrokes are combined into a single character.

The Asian ideographic languages are what make internationalized text input complicated. Japanese and Korean both have phonetic alphabets that are small enough to be mapped onto a keyboard. While it is sometimes adequate to leave text in this representation, the user usually wants the final text to be in the full ideographic language. Input methods for these languages often have the user type the phonetic symbols for a particular word or words and then signal that the composition or pre-editing is complete. At this point, the input method

* The XmNfontList resource is deprecated as of Motif 2.0.

† This is different to the way in which compound strings are rendered: if a font or tag is absent when rendering a compound string, callbacks may be invoked. See Chapter 24, *Render Tables*, for more details.

can look up the string of phonetic characters in a dictionary and convert it to the equivalent character or characters in the ideographic language. Multiple characters can have the same phonetic representation, so the user may still have to select the desired character.

Since input methods can be large and complex and they vary from locale to locale, it does not make sense to link every application with a generic input method that is localized at runtime. The X Input Method (`XIM`) abstraction supports the model of an *input manager* that is run as a separate process and that communicates with the X server and with the application. An application that needs to use an input method calls `XOpenIM()` to establish a connection to the input method that is appropriate for the current locale.

An input method needs to provide feedback to the user, so X defines three areas for interaction:

- The *status area* is an output-only window that displays information about the state of the input method interaction.
- The *pre-edit area* displays the intermediate text while the user is composing a character.
- The *auxiliary area* is used to display any dialog boxes or popup menus that are needed by the input method.

An application generally provides the status and pre-edit areas to the input method, which is responsible for their contents. The auxiliary area is managed entirely by the input method. The location of the pre-edit area depends on the interaction style used between the input method and the application. X defines the following four interaction styles:

- The *root-window* style, where the input method displays the pre-edit data in a window that is a child of the root window.
- The *off-the-spot* style, in which the input method displays the data at a fixed location in the application window, often at the bottom of the window.
- The *over-the-spot* style, where the input method displays the data in a window of its own that is placed over the current insertion point.
- The *on-the-spot* style, in which the input method directs the application to display the pre-edit data, so the application can display the data however it wants.

An application must choose an interaction style that is supported by the input method and it must provide the pre-edit and status areas as required by that style.

Just as the X server can display multiple windows for a single client, an input method can maintain multiple *input contexts* for an application. A text editor that supports multiple editing windows within a single top-level window could create an input context for each window or share a single context among all of the windows. The function `XCreateIC()` creates an X Input Context (`XIC`) that keeps track of information about the input context,

such as the interaction style, the windows used for the pre-edit and status areas, and the font set for the text.

When an application gets a KeyPress event, it needs to use that event in a call to XmbLookupString() or XwcLookupString() to get the multibyte or wide-character string encoded in the current locale. These routines are analogous to XLookupString(), but this routine can only return Latin-1 strings, so it is not appropriate for internationalized input.

The support for input methods in Xlib is designed to be incorporated within toolkits and widgets. Accordingly, the internationalized text input capabilities of the Motif Text and TextField widgets are layered on top of the input method mechanism. Fortunately, the widgets encapsulate most of the lower-level functionality, so you don't need to understand the details of the Xlib implementation. For a more complete description of the Xlib functionality, see Volume 1, *Xlib Programming Manual*.

Motif leaves it to the hardware vendors to supply input methods, so the toolkit does not provide any itself. If you need to provide internationalized text input, consult the documentation for your system for information about the input methods that it supports. Alternately, you can build one of the contributed input methods provided as part of X11. X11R5 as shipped from MIT contains two separate implementations of the input method facilities. The Xsi implementation is the default on all but Sony machines, which use the Ximp implementation. Each implementation defines its own protocol for communication between Xlib and input methods. Ximp and Xsi each come with contributed input methods that are not compatible with each other. For X11R6, the X Consortium standardized the input method implementation, allowing for dynamic input method server connectivity. The details of this are beyond the scope of this manual; you are referred to the *Programmer's Supplement for Release 6* of Volumes 1, 2, 4, and 5 for a complete discussion of the topic.

When you create an editable Text or TextField widget, it automatically provides a connection to the input method for the current locale. The VendorShell widget plays a role in internationalization as it defines the XmNinputMethod, XmNpreeditType, and XmNinputPolicy resources for specifying the input method, the interaction style, and the input context creation policy respectively*. A Text or TextField widget is always created as an ancestor of a VendorShell, so the widget can access these resources to set up the connection to the input method. The resources are defined by the VendorShell because it handles the geometry management of the pre-edit and status areas for the input method.

The XmNinputMethod resource specifies the input method portion of the locale modifier that is set before an input method is opened. The format of the value for this resource is vendor-defined. The XmNpreeditType resource sets the interaction style used by the input method. The syntax, possible values, and default value of this resource are also

* XmNinputPolicy is only available from Motif 2.0 onwards.

vendor-dependent. The XmNinputPolicy resource specifies whether an input context is to be created on a per-shell basis (XmPER_SHELL) or for each widget which requests connection to an input method (XmPER_WIDGET).

In Motif 1.2, only the over-the-spot, off-the-spot, and root-window interaction styles are supported. Motif 2.0 also supports the on-the-spot style. Under the off-the-spot style, the VendorShell positions the pre-edit and status areas below the application's main window but inside the shell. The VendorShell handles the geometry management for the areas and places a separator between the main window and the input method area. If the application sets or gets the XmNheight of the shell using XtVaSetValues() or XtVaGetValues(), the height includes the height of the input method area. With the over-the-spot style, the VendorShell still displays the status area at the bottom of the application's top-level window, but the pre-edit area is positioned over the insertion cursor in the Text widget. The Text widget passes the insertion position to the input method, so that the pre-edit area moves as with the insertion cursor.

The Motif toolkit implements its internationalized text input functionality using the following public routines:*

```
void  XmImCloseXIM (Widget widget)
void  XmImFreeXIC (Widget widget, XIC xic)
XIC   XmImGetXIC (Widget widget, XmInputPolicy policy, ArgList argv,
                  Cardinal argc)
XIM   XmImGetXIM (Widget widget)
int   XmImMbLookupString (Widget widget, XKeyPressedEvent *event,
                  char *buffer, int num_bytes, KeySym *keysym, int *status)
void  XmImMbResetIC (Widget widget, char **mb_text)
void  XmImRegister (Widget widget, unsigned int reserved)
void  XmImSetFocusValues (Widget widget, ArgList argv, Cardinal argc)
void  XmImSetValues (Widget widget, ArgList argv, Cardinal argc)
XIC   XmImSetXIC (Widget widget, XIC xic)
void  XmImUnregister (Widget widget)
void  XmImUnsetFocus (Widget widget)
void  XmImVaSetFocusValues (Widget widget, resource-value-list, NULL)
void  XmImVaSetValues (Widget widget, resource-value-list, NULL)
```

These routines simplify the interaction with the lower-level XIM and XIC constructs provided by Xlib. If you need to provide text input in another widget, such as a DrawingArea, you have to handle opening an input method, creating an input context, and obtaining input from the input method yourself. A description of each is found in Volume 6B, *Motif Reference Manual*.

* XmImCloseXIM(), XmImFreeXIC(), XmImGetXIC(), XmImSetXIC() are only available from Motif 2.0 onwards.

Summary

The Motif Text and TextField widgets can be used to provide an application with sophisticated text entry capabilities. The widgets come with a full set of convenience routines that make it easy to perform a number of standard text editing tasks. However, these widgets work best when they are left alone to do their jobs. While they are highly configurable, the little bits of fine tuning you add may cause your code to grow twice as much to accommodate the new features and the necessary error checking.

Exercises

The following exercises are designed to expand on the ideas described in this chapter and introduce some new directions for using Text widgets.

1. Using the `XmNmodifyVerifyCallback`, you can add more data to a Text widget than what is typed by the user. This technique is useful for supporting advanced editing features such as file or word completion. The user should be able to enter the leading part of a word and then type a special character that completes the word automatically, based on a predefined list of words in */usr/dict/words*. Write an `XmNmodifyVerifyCallback` routine that checks each character that is typed and, upon receipt of the special character, looks backwards in the text until it finds whitespace and checks this word against the words in the list. If there is a match, modify the text to complete the word.
2. The function `XmTextSetHighlight()` can be used to highlight text in the same fashion as if the user had selected it. This routine is useful for emphasizing different pieces of text. Based upon the previous exercise, write a simple spell-checker program. Use a PushButton or a menu item to get all of the text from a Text widget and check the words against */usr/dict/words*. Highlight all of the words that are not found in the dictionary so that the user can find them quickly.
3. Modify the *allcaps.c* program to use the `XmNgainFocusCallback` and `XmNlosingFocusCallback` callback routines. When the widget loses the focus, all of the characters should be converted to lower case, and when the input focus is gained, the characters should revert to upper case.
4. The `XmNsource` resource specifies an `XmTextSource`, which is an internal object that contains all of the information about the text in a Text widget. You can set or get the value for this resource using `XtVaSetValues()` and `XtVaGetValues()`. Since the data type is opaque to the programmer, you cannot create your own source, but you can get one from an existing Text widget. By getting the `XmNsource` from one Text widget and setting it in another, you can have two Text widgets that edit the same text. Write a program that does just that.

In this chapter:
- *Menu Types*
- *Creating Simple Menus*
- *Designing Menu Systems*
- *General Menu Creation Techniques*
- *Summary*
- *Exercises*

19

Menus

This chapter describes the different types of menus provided by the Motif toolkit. It also presents a number of ways to create menus in an application and talks about the issues involved in designing menu systems.

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

Menu Types

PulldownMenus that are posted from the MenuBar are the most common menus in an application.Figure 19-1 shows an example of a PulldownMenu.The menu pops up when the user presses the first mouse button on a CascadeButton.* As described in Chapter 4, *The Main Window*, CascadeButtons may be displayed as titles in a MenuBar or as menu items in a PulldownMenu. When the CascadeButton is a child of a MenuBar, the menu drops down below the button when the user clicks on it. When the CascadeButton is an item in an existing menu, the new menu pops up to the right of the item; it is sometimes referred to as a cascading menu or a pullright menu.

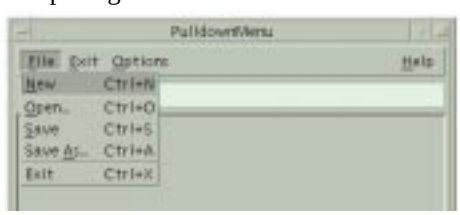


Figure 19-1 A PulldownMenu

* The button that posts the menu is typically user-settable, since left-handed users may want to reverse the default button bindings.

Under certain conditions, it may be inconvenient for the user to stop what she is doing, move the mouse to the MenuBar to pulldown a menu, and then move the mouse back to where she was working. Having to move the mouse away, even to another part of the same window, can reduce productivity. A PopupMenu is one solution to this problem as it can provide immediate access to application functionality. PopupMenus are posted using the third mouse button and can be displayed anywhere in an application. Rather than having to move the mouse, the user can simply press the third mouse button to cause a PopupMenu to appear on the spot. This type of menu does not need to be associated with a visible user-interface element. In fact, PopupMenus are usually popped up from a work area or another region that is not affiliated with a user-interface component like a PushButton or CascadeButton. The only drawback to this design is that there is no indication to the novice user that the menu exists. Figure 19-2 shows a PopupMenu.

Figure 19-2 A PopupMenu

The OptionMenu combines the strengths of a PulldownMenu and a PopupMenu. Like a PulldownMenu, it is posted from a CascadeButton, but like a PopupMenu, it can be placed where it is needed. The CascadeButton is used to display the default choice for the menu. When the user presses the button, the alternate choices are displayed in a menu, as shown in Figure 19-3. Like a PulldownMenu, an OptionMenu is invoked using the first mouse button, but it is displayed on top of its associated CascadeButton rather than below it.

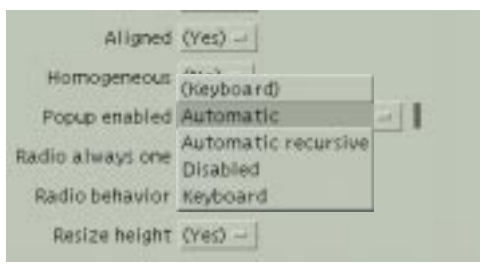


Figure 19-3 An OptionMenu

The use of the third mouse button to activate PopupMenus is in sharp contrast to PulldownMenus and OptionMenus, which are always invoked by the first mouse button. It may seem confusing to the user that some menus are invoked by the first button while others are invoked by the third. However, there is some consistency in the fact that PulldownMenus and OptionMenus are always attached to CascadeButtons, and buttons are always activated by the first mouse button. By specifying that PopupMenus use the third mouse button, the first mouse button is free to be used for other activities in an application work area, which is important since PopupMenus can be popped up anywhere in an application.

When the user posts a menu, it is only displayed until the user makes a selection, and then it is removed. A menu can have an additional feature that allows it to be torn off, so that it remains posted in its own window. The tear-off functionality is activated by a special tear-off button in the menu. The button displays a dashed line to indicate that you can tear off the menu, like you would tear a coupon out of a newspaper. When the user presses the tear-off button, the menu is placed in a separate window, and the user can make as many selections as she would like. Figure 19-4 shows a PulldownMenu that provides the tear-off capability.

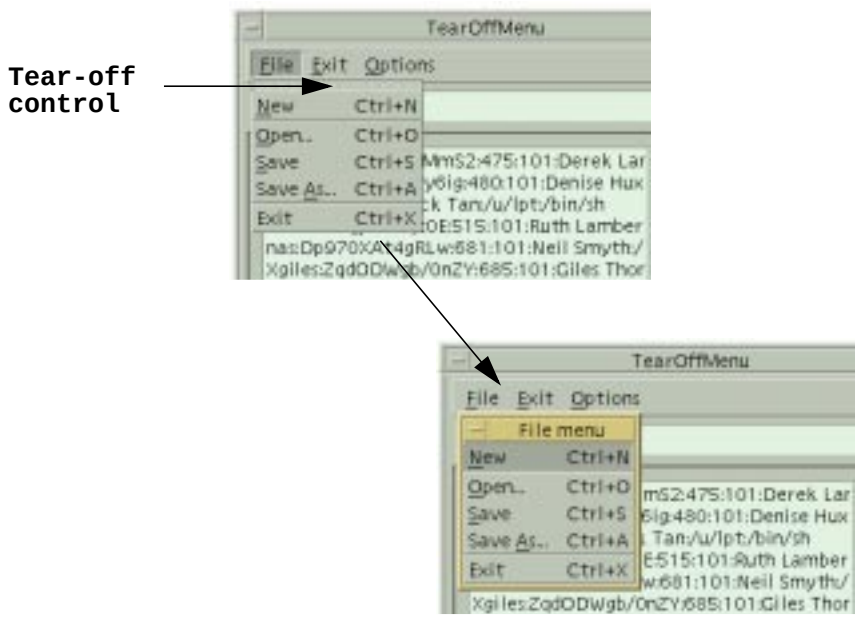


Figure 19-4 A PulldownMenu with tear-off functionality

To make menus even more convenient to use, menu items can have *mnemonics* and *accelerators* associated with them. These devices are keyboard equivalents that allow the user to activate menu items using the keyboard rather than the mouse. For example, in

Figure 19-1, the underlined letter in each menu item is its mnemonic. While the menu is posted, the user can type the specified character to activate that menu item. Accelerators are keystroke combinations that invoke a menu item even when the menu is not displayed. Accelerators typically use the CTRL or ALT key to distinguish them from ordinary keystrokes that are sent to the application. For example, again in Figure 19-1, the `Ctrl+X` accelerator allows the user to exit the application without accessing the menu.

Before we plunge into the details of menu creation, a word of warning to experienced X Toolkit programmers is in order. Motif does not use Xt's normal methods for creating and managing menus. In fact, you *cannot* use the standard Xt methods for menu creation or management without virtually re-implementing the Motif menu design.* In Xt, you would typically create an OverrideShell that contains a generic manager widget, followed by a set of PushButtons. To display the menu, you would pop up the shell using `XtPopup()`. The Motif toolkit abstracts the menu creation and management process using routines that make the shell opaque to the programmer.

Creating Simple Menus

In Chapter 4, *The Main Window*, we used the simple menu creation routines to build the MenuBar and its associated PulldownMenus. These routines are designed to be plug-and-play convenience routines; their only requirements are compound strings for the menu labels and a single callback function that is invoked when the user activates any of the menu items.

`XmVaCreateSimpleMenuBar()` creates a MenuBar, while `XmVaCreateSimplePulldownMenu()` generates a PulldownMenu and its associated items. These functions take a variable-length argument list of parameters that specify either the CascadeButtons for the MenuBar or the menu items for the PulldownMenu. You can also pass RowColumn-specific resource/value pairs to configure the RowColumn widget that manages the items in the menu. The functions are front ends for more primitive routines that actually create the underlying widgets, so they are convenient for many simple menu creation needs. You should review Chapter 4, for more information on how to use these functions.

Motif also provides simple creation routines for creating PopupMenus and OptionMenus. Both `XmVaCreateSimplePopupMenu()` and `XmVaCreateSimpleOptionMenu()` are very similar to the routines for creating PulldownMenus, so much of the information in Chapter 4 also applies to these functions.

* If you need to port an Athena or OPEN LOOK-based application to Motif, you will probably have to reimplement your menu design.

Popup Menus

The only difference between `XmVaCreateSimplePulldownMenu()` and `XmVaCreateSimplePopupMenu()` is that the latter routine does not have a *`button`* parameter for specifying the CascadeButton used to display the menu. Since PopupMenus are not associated with CascadeButtons, this parameter isn't necessary. Example 19-1 demonstrates the creation of a simple PopupMenu.*

Example 19-1 The simple_popup.c program

```
/* simple_popup.c -- demonstrate how to use a simple popup menu.
** Create a main window that contains a DrawingArea widget, which
** displays a popup menu when the user presses the third mouse button.
*/

#include <Xm/RowColumn.h>
#include <Xm/MainW.h>
#include <Xm/DrawingA.h>

main (int argc, char *argv[])
{
    XmString       line, square, circle, exit_b, exit_acc;
    Widget         toplevel, main_w, drawing_a, popup_menu;
    void           popup_cb(Widget, XtPointer, XtPointer);
    XtAppContext   app;
    Arg            args[4];
    int            n;

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

    /* Create a MainWindow widget that contains a DrawingArea in
    ** its work window.
    */
    n = 0;
    XtSetArg (args[n], XmNscrollingPolicy, XmAUTOMATIC); n++;
    main_w = XmCreateMainWindow (toplevel, "main_w", args, n);

    /* Create a DrawingArea -- no actual drawing will be done. */
    n = 0;
    XtSetArg (args[n], XmNwidth, 500);  n++;
    XtSetArg (args[n], XmNheight, 500); n++;
    drawing_a = XmCreateDrawingArea (main_w, "drawing_a", args, n);
    XtManageChild (drawing_a);

    line = XmStringCreateLocalized ("Line");
    square = XmStringCreateLocalized ("Square");
```

* `XtVaAppInitialize()` is considered deprecated in X11R6. The XmNpopupEnabled resource is modified in Motif 2.0 and later to support the values XmPOPUP_AUTOMATIC, XmPOPUP_AUTOMATIC_RECURSIVE, XmPOPUP_DISABLED, XmPOPUP_KEYBOARD.

```
    circle = XmStringCreateLocalized ("Circle");
    exit_b = XmStringCreateLocalized ("Exit");
    exit_acc = XmStringCreateLocalized ("Ctrl+C");
    popup_menu = XmVaCreateSimplePopupMenu (drawing_a, "popup", popup_cb,
                        XmNpopupEnabled, XmPOPUP_AUTOMATIC,
                        XmVaPUSHBUTTON, line, 'L', NULL, NULL,
                        XmVaPUSHBUTTON, square, 'S', NULL, NULL,
                        XmVaPUSHBUTTON, circle, 'C', NULL, NULL,
                        XmVaSEPARATOR,
                        XmVaPUSHBUTTON, exit_b, 'x', "Ctrl<Key>c", exit_acc,
                        NULL);
    XmStringFree (line);
    XmStringFree (square);
    XmStringFree (circle);
    XmStringFree (exit_b);
    XmStringFree (exit_acc);

    XtManageChild (main_w);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

/* popup_cb() -- invoked when the user selects an item in the popup menu */
void popup_cb (Widget menu_item, XtPointer client_data, XtPointer call_data)
{
    int item_no = (int) client_data;

    if (item_no == 3) /* Exit was selected -- exit */
        exit (0);

    /* Otherwise, just print the selection */
    puts (XtName (menu_item));
}
```

This program creates a standard MainWindow widget that contains a DrawingArea widget. The program does not do any drawing; it is just a skeleton that demonstrates how to attach a PopupMenu. The PopupMenu is created using `XmVaCreateSimplePopupMenu()` with the DrawingArea widget as its parent. The menu is popped up when the user presses the third mouse button in the DrawingArea, as shown in Figure 19-5.

Figure 19-5 Output of the simple_popup program

The menu contains four items, the last of which has the accelerator Ctrl<Key>C. Any time the user presses CTRL-C in the application, the callback routine associated with the menu is called as if the menu had been popped up and the *Exit* item had been selected. The popup_cb() routine either prints the name of the menu item or exits, depending on which item the user selected. Note that the name of the menu item does not correspond to its label. As described in Chapter 4, *The Main Window*, menu items are automatically given names of the form button_*n*, where *n* is assigned in order of menu item creation, starting at 0 (zero).

In Motif 1.2, PopupMenus are not automatically displayed by the toolkit: the programmer must install an event handler, or, for the DrawingArea, an XmNinputCallback, in order to catch ButtonPress events. Once caught, a call to XmMenuPosition() to place the menu at the cursor, followed by XtManageChild() of the menu itself are required to effect the posting.

In Motif 2.0 and later, this is not necessary because automatic popup is now supported. For the simple case, simply setting the XmNpopupEnabled resource to XmPOPUP_AUTOMATIC has the desired effect. Where there is potentially a choice between several popup menus in a given context, the resource XmNpopupHandlerCallback can be used to discriminate at the appropriate juncture. XmNpopupHandlerCallback is defined in both the Manager and Primitive classes, and is thus inherited by all widgets in the Motif set. Note that the resource is not defined for the Gadget class: you need to manipulate the required popup through the Manager parent. Where no callback is registered, the toolkit will select the menu to display.

The XmNpopupMenuHandler callback is passed a structure of type XmPopupHandlerCallbackStruct as the third parameter when invoked. The structure is defined as follows:

```
typedef struct
{
    int          reason;
    XEvent       *event;
    Widget       menuToPost;
    Boolean      postIt;
    Widget       target;
} XmPopupHandlerCallbackStruct;
```

The reason and event fields are the familiar elements found in all Motif callback structures; here, the reason field will have the value XmCR_POST or XmCR_REPOST. XmCR_POST is the normal value; XmCR_REPOST will occur if the menu is reposted as a result of event replay. menuToPost is the toolkit's suggestion of the menu to display: this element in the structure should be modified appropriately if a different popup menu is required. The target field holds the widget or gadget which the Manager believes is the source of the popup request. Lastly, postIt is a flag which indicates whether the posting action is to continue after the callback completes.

Example 19-2 creates two popups, and adds an XmNpopupHandlerCallback choose_cb onto the DrawingArea. This randomly picks one of the two menus to display, depending upon the x coordinate of the Button event: if even, it displays menuA, otherwise menuB.*

Example 19-2 The choice_popup.c program

```
/* choice_popup.c -- demonstrate how to use a popup menu handler.
** Create a main window that contains a DrawingArea widget, which
** chooses between two popup menus when the user presses the third
** mouse button.
*/

#include <Xm/RowColumn.h>
#include <Xm/MainW.h>
#include <Xm/DrawingA.h>

Widget menuA, menuB;

main (int argc, char *argv[])
{
    XmString       line, square, circle, exit, exit_acc;
    XmString       red, green, blue;
    Widget         toplevel, main_w, drawing_a;
    void           popup_cb(Widget, XtPointer, XtPointer);
    void           choose_cb(Widget, XtPointer, XtPointer);
    XtAppContext   app;
    Arg            args[4];
    int            n;

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

    /* Create a MainWindow widget that contains a DrawingArea in
    ** its work window.
    */
    n = 0;
    XtSetArg (args[n], XmNscrollingPolicy, XmAUTOMATIC); n++;
    main_w = XmCreateMainWindow (toplevel, "main_w", args, n);

    /* Create a DrawingArea -- no actual drawing will be done. */
    n = 0;
    XtSetArg (args[n], XmNwidth, 500);  n++;
    XtSetArg (args[n], XmNheight, 500); n++;
    drawing_a = XmCreateDrawingArea (main_w, "drawing_a", args, n);
    /* Callback to choose which popup menu to display */
    XtAddCallback (drawing_a, XmNpopupHandlerCallback, choose_cb, NULL);
    XtManageChild (drawing_a);

    line = XmStringCreateLocalized ("Line");
```

* XtVaAppInitialize() is considered deprecated in X11R6. The XmNpopupHandlerCallback resource, and the automatic popup of menus. are only available in Motif 2.0 and later.

```
    square = XmStringCreateLocalized ("Square");
    circle = XmStringCreateLocalized ("Circle");
    exit = XmStringCreateLocalized ("Exit");
    exit_acc = XmStringCreateLocalized ("Ctrl+C");
    menuA = XmVaCreateSimplePopupMenu (drawing_a, "menuA", popup_cb,
                            XmNpopupEnabled, XmPOPUP_AUTOMATIC,
                            XmVaPUSHBUTTON, line, 'L', NULL, NULL,
                            XmVaPUSHBUTTON, square, 'S', NULL, NULL,
                            XmVaPUSHBUTTON, circle, 'C', NULL, NULL,
                            XmVaSEPARATOR,
                            XmVaPUSHBUTTON, exit, 'x', "Ctrl<Key>c", exit_acc,
                            NULL);
    XmStringFree (line);
    XmStringFree (square);
    XmStringFree (circle);
    XmStringFree (exit);
    XmStringFree (exit_acc);

    red = XmStringCreateLocalized ("Red");
    green = XmStringCreateLocalized ("Green");
    blue = XmStringCreateLocalized ("Blue");

    menuB = XmVaCreateSimplePopupMenu (drawing_a, "menuB", popup_cb,
                            XmNpopupEnabled, XmPOPUP_AUTOMATIC,
                            XmVaPUSHBUTTON, red, 'R', NULL, NULL,
                            XmVaPUSHBUTTON, green, 'G', NULL, NULL,
                            XmVaPUSHBUTTON, blue, 'B', NULL, NULL,
                            NULL);
    XmStringFree (red);
    XmStringFree (green);
    XmStringFree (blue);

    XtManageChild (main_w);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

/* popup_cb() -- invoked when the user selects an item in the popup menu */
void popup_cb (Widget menu_item, XtPointer client_data,
               XtPointer call_data)
{
    int item_no = (int) client_data;

    if (item_no == 3) /* Exit was selected -- exit */
        exit (0);

    /* Otherwise, just print the selection */
    puts (XtName (menu_item));
}

/* choose_cb() -- invoked when the user requests a popup menu */
void choose_cb (Widget menu_item, XtPointer client_data,
                    XtPointer call_data)
{
```

```
    XmPopupHandlerCallbackStruct *cbs =
                            (XmPopupHandlerCallbackStruct *) call_data;
    XButtonPressedEvent *bp = (XButtonPressedEvent *) cbs->event;

    if ((bp->x % 2) == 0) {
        cbs->menuToPost = menuA;
    }
    else {
        cbs->menuToPost = menuB;
    }
}
```

Cascading Menus

A cascading menu, or a pullright menu, is implemented as a PulldownMenu displayed from a menu item in another PulldownMenu or PopupMenu. The menu item that posts the cascading menu must be a CascadeButton. Example 19-3 demonstrates how to add a cascading menu using the simple menu routines. The program adds a *Line Width* menu item to the PopupMenu from Example 19-1. This menu item is a CascadeButton that posts a PulldownMenu created with `XmVaCreateSimplePulldownMenu()`.*

Example 19-3 The simple_pullright.c program

```
/* simple_pullright.c -- demonstrate how to make a pullright menu
** using simple menu creation routines. Create a main window that
** contains a DrawingArea widget that displays a popup menu when the
** user presses the third mouse button.
*/

#include <Xm/RowColumn.h>
#include <Xm/MainW.h>
#include <Xm/DrawingA.h>

main (int argc, char *argv[])
{
    XmString       line, square, circle, weight, exit, exit_acc;
    XmString       w_one, w_two, w_four, w_eight;
    Widget         toplevel, main_w, drawing_a, cascade, popup_menu,
                   pullright;
    void           popup_cb(Widget, XtPointer, XtPointer);
    void           set_width(Widget, XtPointer, XtPointer);
    XtAppContext   app;
    Arg            args[4];
    int            n;

    XtSetLanguageProc (NULL, NULL, NULL);
```

* `XtVaAppInitialize()` is considered deprecated in X11R6. The XmNpopupEnabled resource is modified in Motif 2.0 and later to support the values XmPOPUP_AUTOMATIC, XmPOPUP_AUTOMATIC_RECURSIVE, XmPOPUP_DISABLED, XmPOPUP_KEYBOARD.

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

/* Create a MainWindow widget that contains a DrawingArea in
** its work window.
*/
n = 0;
XtSetArg (args[0], XmNscrollingPolicy, XmAUTOMATIC); n++;
main_w = XmCreateMainWindow (toplevel, "main_w", args, n);

/* Create a DrawingArea -- no actual drawing will be done. */
n = 0;
XtSetArg (args[n], XmNwidth, 500); n++;
XtSetArg (args[n], XmNheight, 500); n++;
drawing_a = XmCreateDrawingArea (main_w, "drawing_a", args, n);
XtManageChild (drawing_a);

line = XmStringCreateLocalized ("Line");
square = XmStringCreateLocalized ("Square");
circle = XmStringCreateLocalized ("Circle");
weight = XmStringCreateLocalized ("Line Width");
exit = XmStringCreateLocalized ("Exit");
exit_acc = XmStringCreateLocalized ("Ctrl+C");
popup_menu = XmVaCreateSimplePopupMenu (drawing_a, "popup", popup_cb,
                        XmNpopupEnabled, XmPOPUP_AUTOMATIC,
                        XmVaPUSHBUTTON, line, 'L', NULL, NULL,
                        XmVaPUSHBUTTON, square, 'S', NULL, NULL,
                        XmVaPUSHBUTTON, circle, 'C', NULL, NULL,
                        XmVaCASCADEBUTTON, weight, 'W',
                        XmVaSEPARATOR,
                        XmVaPUSHBUTTON, exit, 'x', "Ctrl<Key>c", exit_acc,
                        NULL);
XmStringFree (line);
XmStringFree (square);
XmStringFree (circle);
XmStringFree (weight);
XmStringFree (exit);

/* create pullright for "Line Width" button -- this is the 4th item! */
w_one = XmStringCreateLocalized ("1");
w_two = XmStringCreateLocalized ("2");
w_four = XmStringCreateLocalized ("4");
w_eight = XmStringCreateLocalized ("8");
pullright = XmVaCreateSimplePulldownMenu (popup_menu, "pullright",
                        3 /* menu item offset */, set_width,
                        XmVaPUSHBUTTON, w_one, '1', NULL, NULL,
                        XmVaPUSHBUTTON, w_two, '2', NULL, NULL,
                        XmVaPUSHBUTTON, w_four, '4', NULL, NULL,
                        XmVaPUSHBUTTON, w_eight, '8', NULL, NULL,
                        NULL);
XmStringFree (w_one);
XmStringFree (w_two);
XmStringFree (w_four);
XmStringFree (w_eight);
```

```
    XtManageChild (main_w);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

/* popup_cb() -- invoked when the user selects an item in the popup menu */
void popup_cb (Widget menu_item, XtPointer client_data,
               XtPointer call_data)
{
    int item_no = (int) client_data;
    if (item_no == 4) /* Exit was selected -- exit */
        exit (0);
    /* Otherwise, just print the selection */
    puts (XtName (menu_item));
}

/* set_width() -- called when items in the Line Width pullright menu
** are selected.
*/
void set_width (Widget menu_item, XtPointer client_data,
               XtPointer call_data)
{
    int item_no = (int) client_data;
    printf ("Line weight = %d\n", 1 << item_no);
}
```

In the call to `XmVaCreateSimplePulldownMenu()`, the PopupMenu is specified as the parent of the cascading menu. The *`button`* parameter is set to `3` to indicate that the fourth item in the PopupMenu posts the cascading menu. Figure 19-6 shows the output of the program.

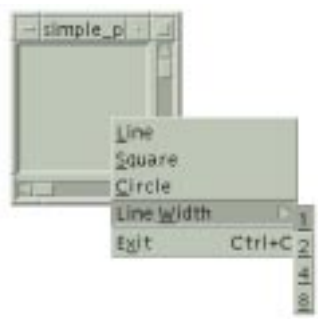


Figure 19-6 Output of the simple_pullright program

Option Menus

An OptionMenu is similar to a PulldownMenu in that they are both associated with CascadeButtons. However, there are also several major differences between the two types

of menus. In an OptionMenu, the CascadeButton is not part of a MenuBar. Instead, it is created as the child of a RowColumn widget that also contains a Label gadget.

Another difference is that the menu pops up on top of the CascadeButton, instead of dropping down from it. The label on the CascadeButton is one of the elements in the menu; the CascadeButton displays the current menu selection. The Motif toolkit handles the management of the PulldownMenu for the OptionMenu, so its handle is not available to you, nor does it need to be. Because of the design of the OptionMenu, it cannot have cascading menus.

Example 19-4 demonstrates the use of `XmVaCreateSimpleOptionMenu()`. The program uses a DrawingArea again, but now the user selects the drawing style from an OptionMenu that is displayed above the DrawingArea*.

Example 19-4 The simple_option.c program

```
/* simple_option.c -- demonstrate how to use a simple option menu.
** Display a drawing area. The user selects the drawing style from
** the option menu.
*/

#include <Xm/RowColumn.h>
#include <Xm/MainW.h>
#include <Xm/ScrolledW.h>
#include <Xm/DrawingA.h>
#include <Xm/PushB.h>

main (int argc, char *argv[])
{
    XmString       draw_shape, line, square, circle;
    Widget         toplevel, main_w, rc, sw, drawing_a, option_menu, pb;
    void           option_cb(Widget, XtPointer, XtPointer);
    void           exit(int);
    XtAppContext   app;
    Arg            args[4];
    int            n;

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

    /* Create a MainWindow widget that contains a RowColumn
    ** widget as its work window.
    */
    main_w = XmCreateMainWindow (toplevel, "main_w", NULL, 0);
    rc = XmCreateRowColumn (main_w, "rowcol", NULL, 0);

    /* Inside RowColumn is the Exit pushbutton, the option menu and the
    ** scrolled window that contains the drawing area.
```

* `XtVaAppInitialize`() is considered deprecated in X11R6.

```
    */
    pb = XmCreatePushButton (rc, "Exit", NULL, 0);
    XtAddCallback (pb, XmNactivateCallback, (void (*)()) exit, NULL);
    XtManageChild (pb);

    draw_shape = XmStringCreateLocalized ("Draw Mode:");
    line = XmStringCreateLocalized ("Line");
    square = XmStringCreateLocalized ("Square");
    circle = XmStringCreateLocalized ("Circle");
    option_menu = XmVaCreateSimpleOptionMenu (rc,
                                "option_menu", draw_shape, 'D',
                                0 /*initial menu selection*/, option_cb,
                                XmVaPUSHBUTTON, line, 'L', NULL, NULL,
                                XmVaPUSHBUTTON, square, 'S', NULL, NULL,
                                XmVaPUSHBUTTON, circle, 'C', NULL, NULL,
                                NULL);
    XmStringFree (line);
    XmStringFree (square);
    XmStringFree (circle);
    XmStringFree (draw_shape);
    XtManageChild (option_menu);
    /* Create a DrawingArea inside a ScrolledWindow */
    n = 0;
    XtSetArg (args[n], XmNscrollingPolicy, XmAUTOMATIC); n++;
    sw = XmCreateScrolledWindow (rc, "sw", args, n);

    n = 0;
    XtSetArg (args[n], XmNwidth, 500);  n++;
    XtSetArg (args[n], XmNheight, 500); n++;
    drawing_a = XmCreateDrawingArea (sw, "drawing_area", args, n);
    XtManageChild (drawing_a);
    XtManageChild (sw);
    XtManageChild (rc);
    XtManageChild (main_w);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

/* option_cb() -- invoked when the user selects an item in the
** option menu
*/
void option_cb (Widget menu_item, XtPointer client_data,
                XtPointer call_data)
{
    int item_no = (int) client_data;
    puts (XtName (menu_item));
}
```

The layout of the application is different from that in the previous examples because we use a separate ScrolledWindow for the DrawingArea. The RowColumn widget that contains the *Exit* button, the OptionMenu, and the ScrolledWindow is the work area for the MainWindow. Figure 19-7 shows the output of the program both before and after the

OptionMenu is displayed. Notice how the label of the CascadeButton changes as you select alternate values from the menu.

Figure 19-7 Output of the simple_option program

Designing Menu Systems

The advantages of the simple menu creation routines are clear. It is easy to create menus with them, the code is extremely readable, and the job gets done without much room for error. Once the code is written, it is easy to modify the callback function, labels, mnemonics, and accelerators used by a menu.

There are also some disadvantages to using the simple menu creation functions. One problem is that they require a great deal of bulk to create a single menu. If an application needs to create a large number of menus, it has to use a lot of redundant code because the simple creation routines make it difficult to build a looping construct or a function to automate the process. Since the creation routines name the widgets using non-unique names, it is difficult to specify labels, mnemonics, and accelerators in a resource file. If these values are set using a creation routine, this point is irrelevant because the routines hard-code the values. The simple creation routines also make it impossible to specify different callback functions for menu items.

To get around the shortcomings of the simple creation routines, we are going to build a new system that is just as simple to use, but more dynamic and easy to modify. Before we can build our new system, we need to examine the advanced Motif menu creation routines and discuss the overall design of a menu system. We are going to start with the MenuBar and PulldownMenus because almost every application uses these components. Furthermore, everything there is to know about menus can be adapted from the design of a menu system that uses these menus.

Let’s begin by examining the steps that you need to take to create a MenuBar and its associated PulldownMenus:

1. Create a RowColumn widget for use as a MenuBar with `XmCreateMenuBar()`.

2. Create each PulldownMenu using `XmCreatePulldownMenu ()`.
3. Create the menu items (PushButtons, ToggleButtons, Separators, etc.) for each PulldownMenu.
4. Create a CascadeButton for each menu in the MenuBar and attach the associated PulldownMenu to it.
5. Manage the MenuBar with `XtManageChild()`.

The program in Example 19-5 demonstrates these steps by creating a MenuBar that contains a single *File* PulldownMenu.*

Example 19-5 The file_menu.c program

```
/* file_menu.c -- demonstrate how to create a menu bar and pulldown
** menu using the Motif creation routines.
*/

#include <Xm/RowColumn.h>
#include <Xm/MainW.h>
#include <Xm/CascadeB.h>
#include <Xm/SeparatoG.h>
#include <Xm/PushBG.h>

main (int argc, char *argv[])
{
    Widget        toplevel, main_w, menu_w, file_w, cascade_w, push_b, sep_w;
    XmString      label_str;
    XtAppContext  app;
    Arg           args[4];
    int           n;

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

    n = 0;
    XtSetArg (args[n], XmNscrollingPolicy, XmAUTOMATIC); n++;
    main_w = XmCreateMainWindow (toplevel, "main_w", args, n);
    menu_w = XmCreateMenuBar (main_w, "MenuBar", NULL, 0);

    /* create the "File" Menu */
    file_w = XmCreatePulldownMenu (menu_w, "FilePullDown", NULL, 0);

    /* create the "File" button (attach Menu via XmNsubMenuId) */
    label_str = XmStringCreateLocalized ("File");

    n = 0;
    XtSetArg (args[n], XmNmnemonic, 'F');          n++;
    XtSetArg (args[n], XmNlabelString, label_str); n++;
```

* `XtVaAppInitialize`() is considered deprecated in X11R6.

```
        XtSetArg (args[n], XmNsubMenuId, file_w);      n++;
        cascade_w = XmCreateCascadeButton (menu_w, "File", args, n);
        XtManageChild (cascade_w);
        XmStringFree (label_str);

        /* Now add the menu items */
        push_b = XmCreatePushButtonGadget (file_w, "Open", NULL, 0);
        XtManageChild (push_b);
        push_b = XmCreatePushButtonGadget (file_w, "Save", NULL, 0);
        XtManageChild (push_b);
        sep_w = XmCreateSeparatorGadget (file_w, "separator", NULL, 0);
        XtManageChild (sep_w);
        push_b = XmCreatePushButtonGadget (file_w, "Exit", NULL, 0);
        XtManageChild (push_b);

        XtManageChild (menu_w);
        XtManageChild (main_w);
        XtRealizeWidget (toplevel);
        XtAppMainLoop (app);
    }
```

The code follows the steps that we just outlined. The MenuBar is created as a child of the MainWindow, and the PulldownMenu is created as a child of the MenuBar. The CascadeButton acts as the *File* title item in the MenuBar, so it is also created as the child of the MenuBar. Both the menu title and the PulldownMenu are children of the MenuBar. The CascadeButton sets its `XmNsubMenuId` resource to the PulldownMenu so that when the button is selected, it knows which PulldownMenu to display. When you create a PulldownMenu using the simple menu creation routine, it sets this resource behind the scenes.

We also set the label of the CascadeButton using the `XmNlabelString` resource. This value is a compound string, just as in the simple creation function. If we had not set the label directly, the name of the widget itself would appear as the label, and we could override it with a specification in a resource file. Since we are not using the simple creation routine, we can choose whether or not we hard-code the label for the CascadeButton. After we create the items in the menu, we manage the MenuBar using `XtManageChild()`. The output of Example 19-5, both before and after the PulldownMenu is posted, is shown in Figure 19-8.

Figure 19-8 Output of the file_menu program

Menu Titles

The titles in a MenuBar are actually the labels of the CascadeButtons. The labels can be specified using the `XmNlabelString` resource, either in the application code or in a resource file. Every CascadeButton must have a submenu associated with it via the `XmNsubMenuId` resource. When the user selects the CascadeButton, the associated PulldownMenu is displayed. You should never attach a callback function directly to a CascadeButton in the MenuBar as it would confuse the user. Callback functions should only be attached to menu items in PulldownMenus that are posted from the MenuBar.

The PulldownMenu that is associated with a CascadeButton is created using `XmCreatePulldownMenu()`. This routine returns the RowColumn widget that manages the menu items. The routine creates the RowColumn as a child of a MenuShell widget. Since the routine returns the RowColumn widget, the resource list provided to the function only sets resources for the RowColumn widget, not for the MenuShell that contains it.

Menu titles should not be dynamically created or destroyed. An application should not make the MenuBar disappear or add new titles to the MenuBar while the application is running. All of the titles in the MenuBar must be available to the user when the MainWindow is visible. You can, however, deactivate an entire menu by changing the `XmNsensitive` resource on the CascadeButton widget that acts as its title, as discussed in the *Building Pulldown Menus* Section.

Menu Items

The items in a menu are actually the labels of the PushButtons that make up the menu. Unlike the *File* title item in the MenuBar, we chose not to use hard-coded values for the menu item strings, so the strings can be set in a resource file. While our menu only contains PushButton gadgets, a PulldownMenu can also contain ToggleButtons, Separators, and CascadeButtons.

You can install a callback routine for each of the items in a menu, or you can install an XmNentryCallback for the RowColumn widget to act on behalf of all the menu items. This resource specifies a callback function that overrides the XmNactivateCallback used by Pushbuttons and the XmNvalueChangedCallback used byToggleButtons. Using this resource generates a design that is similar to the simple menu routines described earlier. See Chapter 8, *Manager Widgets*, for details on this generic RowColumn resource.

As with the title items, menu items should not be dynamically created or destroyed since it may confuse the user. However, there is one exception to this guideline. If a menu contains items that keep track of a dynamic list of objects, such as the open files in a text editor, the menu items should change to reflect the current state of the application.

Mnemonics

Mnemonics help users traverse the menu system and select actual menu items without having to use the mouse. In Example 19-5, we used the XmNmnemonic resource to attach the mnemonic "F" to the *File* menu, which allows the user to use the key sequence ALT-F to open or close the menu without using the mouse. The XmNmnemonic resource is defined by the Label class, but it is only used by PushButtons, ToggleButtons, and CascadeButtons when these objects are used in a menu system.

A mnemonic is represented visually by the underlining of the mnemonic character in the label string. In this case, the "F" in the word "File" is underlined. If the label does not contain the mnemonic character, there is no visual feedback for the mnemonic, but it still functions. When a mnemonic is specified, the character can be either uppercase or lowercase, but the distinction only affects which letter is underlined. For operational purposes, mnemonics are case insensitive.

Our example only provided a mnemonic for the entire menu, but mnemonics can be set on menu items as well. When a PulldownMenu is displayed, the user can activate a menu item simply by typing the letter represented by its mnemonic. (The ALT key is not used once the menu is displayed.) If the user activates a menu item using a mnemonic, the callback function for the menu is called just as if the user had selected it with the mouse.

Mnemonics are set on MenuBar titles and menu items in the same way. To illustrate, let's add a mnemonic to the *Exit* item in our *File* menu. We could, as in Example 19-5, set the mnemonic directly in the creation of the item, as follows:

```
n = 0;
XtSetArg (args[n], XmNmnemonic, 'x'); n++;
push_b = XmCreatePushButtonGadget (file_w, "Exit", args, n);
```

Strictly speaking, since the internal representation of an XmNmnemonic is as a KeySym, and not a char, the following method of assigning a mnemonic to a widget, using the Xt conversion mechanisms, is preferred:

```
XrmValue from_value, to_value; /* For resource conversion */
```

```
...
n = 0;
from_value.addr = "x";
from_value.size = strlen(from_value.addr) + 1;
to_value.addr = NULL;
XtConvertAndStore (file_w, XmRString, &from_value, XmRKeySym, &to_value);
if (to_value.addr) {
    XtSetArg (args[n], XmNmnemonic, (*((KeySym*) to_value.addr))); n++;
}
push_b = XmCreatePushButtonGadget (file_w, "Exit", args, n);
```

While these methods accomplish the task, one problem with them is that the mnemonic is hard-coded in the widget, while the label is not. Consider the following resource specification in a resource file:

```
*Exit.labelString: Quit
```

This resource sets the label for the item button to “Quit”, but since the mnemonic for the button is hard-coded to “x”, there is visual feedback, and the mnemonic itself is counter-intuitive.

The best way to handle this situation is to specify both the label string and the mnemonic in the same place: a resource file or application code. For example:

```
*Exit.labelString: Exit
*Exit.mnemonic: x
```

Setting both of these resources in the same way helps ensure that an application has a consistent interface.

Accelerators

The purpose of menu accelerators is to provide the user with the ability to activate menu items in a PulldownMenu without having to display the menu at all. In Figure 19-1, the *Quit* menu item displayed the accelerator `Ctrl+C` to indicate that the user could press the CTRL-C keyboard sequence to activate that menu item and quit the application.

To install a accelerator on a menu item, use the `XmNaccelerator` resource to specify the accelerator translation and `XmNacceleratorText` to provide visual feedback to the user.* These resources are defined by the Label class, but they only work for PushButtons and ToggleButtons in menus. The syntax for the accelerator is exactly the same as for a translation table, except that you do not specify an action function with the event sequence. The accelerator for the *Quit* button in Figure 19-1 is specified as `"Ctrl<Key>C"`. (For

* A side effect of the implementation of Motif accelerators is that you cannot install your own accelerators using the standard methods provided by the X Toolkit Intrinsics (such as `XtInstallAccelerators()` or `XtInstallAllAccelerators()`).These functions will not work, and you may interfere with the Motif accelerator mechanism by attempting to use them.

information on how to specify translation tables, see Volume 4, *X Toolkit Intrinsics Programming Manual*.

However, the string that is displayed for the accelerator is not the same as the accelerator translation because it would be confusing for most users. Instead, you should display something like `"^C"`, `"Ctrl-C"`, or `"Ctrl+C"`, as these make it reasonably clear what the user is expected to type. (The latter is the convention recommended by the *Motif Style Guide*, though all three forms are frequently used.) Since this resource specifies displayable text, you cannot use a common C string; the text must be given as a compound string.

For example, the following code demonstrates how to install an accelerator for the *Exit* button in Example 19-5.

```
XmString accel_text = XmStringCreateLocalized ("Ctrl+C");
...
n = 0;
XtSetArg (args[n], XmNaccelerator, "Ctrl<Key>C");   n++;
XtSetArg (args[n], XmNacceleratorText, accel_text); n++;
push_b = XmCreatePushButtonGadget (file_w, "Exit", args, n);
XmStringFree (accel_text);
```

As with mnemonics, the resources for the accelerator itself and the text used to display the accelerator can either be set directly in application code or specified in a resource file. Both of the resources should be specified in the same way, so that they are always consistent.

The Help Menu

Motif specifies various ways for the user to get help. She can use the HELP or F1 keys on the keyboard, the *Help* button in a dialog box, or the *Help* title on the MenuBar. This title provides the highest level of help for your application, so it should not provide too much detail about lower-level functions in the program. When you create a PulldownMenu for this title, it should provide items that give the user access to the help system. Figure 19-9 shows a common *Help* menu.

Figure 19-9 A Help menu from the MenuBar

The choices shown in Figure 19-9 are recommended by the *Motif Style Guide*; if they apply to your application, you should use them.There is usually an item on the *Help* menu that gives the user a brief overview of how to use the help system. You should consult the *Motif Style Guide* for details on what kind of help each of the above selections should provide. It is usually a good idea to have an item that displays an index of the type of help that is available in an application. An example of help index dialog is shown in Figure 19-10. See Chapter 27, *Advanced Dialog Programming*, for a discussion of help dialogs.

Figure 19-10 A Help index dialog

Creating a *Help* menu is just like creating any other menu, except that once you have created the CascadeButton, you should set the `XmNmenuHelpWidget` resource for the MenuBar. This resource specifies which CascadeButton is placed to the far right in the MenuBar, which is where the *Style Guide* states that the *Help* menu must be positioned. Example 19-6 contains a routine that demonstrates how to build a *Help* menu and attach it to the MenuBar. In this example, we present an alternate approach to creating MenuBar titles and their associated PulldownMenus.

Example 19-6 The BuildHelpMenu() routine

```
void BuildHelpMenu (Widget menu_b)
{
    void         do_help(Widget, XtPointer, XtPointer);
    Widget       help_m, widget;
    Arg          args[4];
    int          i, n;
    static char  *h_items[] = { "On Context", NULL, "On Help", "On Window",
                                "On Keys", "Index", "Tutorial", "On Version" };
```

```
        /* Help menu */
        help_m = XmCreatePulldownMenu (menu_b, "HelpPullDown", NULL, 0);

        n = 0;
        XtSetArg (args[n], XmNsubMenuId, help_m); n++;
        widget = XmCreateCascadeButton (menu_b, "Help", args, n);
        XtManageChild (widget);

        /* tell the MenuBar that this is the help widget */
        XtVaSetValues (menu_b, XmNmenuHelpWidget, widget, NULL);

        /* Now add the menu items to the pulldown menu */
        for (i = 0; i < XtNumber (h_items); i++) {
            if (h_items[i] != NULL) {
                widget = XmCreatePushButtonGadget (help_m, h_items[i], NULL, 0);
                XtAddCallback (widget, XmNactivateCallback, do_help,
                               (XtPointer) h_items[i]);
            }
            else
                widget = XmCreateSeparatorGadget (help_m, "sep", NULL, 0);

            XtManageChild (widget);
        }
    }
```

Much of the work required to create a PulldownMenu is involved in creating the menu items. We can optimize the code by using a loop that creates individual items based on the names provided in a static array. If you want to add a new help item to the list, you just need to add its name to the `h_items` list. A `NULL` entry causes a Separator gadget to be added to the menu. In Example 19-6, we specify the same callback function for each item in the menu; the `client_data` is the same as the name of the menu item. In the *General Menu Creation Techniques* Section we expand on this approach to build arbitrary menus for the MenuBar.

Sensitivity

As we mentioned earlier, MenuBar titles and menu items should not be dynamically created or destroyed. They may, however, be activated or deactivated using `XtSetSensitive()`. When a CascadeButton or a menu item is *insensitive*, it is greyed out, and the user is unable to display the associated menu or activate the menu item.

For CascadeButtons, insensitivity has the additional effect of preventing the user from accessing any of the items on the associated menu, including access through mnemonics and accelerators, since the menu cannot be displayed. The menu and all its items are completely unavailable until the sensitivity of the CascadeButton is reset. An alternate way to disable an entire menu is to set the PulldownMenu pane insensitive. This approach has the advantage of still allowing the user to display the menu and see all the items, while making the items unavailable.

For example, take an editor program. If the user is not editing a file, it doesn't make sense to have the *Save* item in the *File* menu selectable. Once the user starts editing a file, the *Save* button is sensitized so that the user can select it. Since the user cannot select the item until its sensitivity is reset, it is important that the application do so at the appropriate time. Another less realistic example, but one that we can demonstrate, involves a menu item that pops up a dialog. As long as that dialog is up, the user cannot reselect the menu item again. For purposes of this demonstration, let's say that the *Open* item pops up a FileSelectionDialog and desensitizes itself. When the dialog is dismissed, the menu item is re-sensitized.*

To implement this behavior, we specify a callback routine for the *Open* menu item that creates a FileSelectionDialog and sets the item insensitive. We also specify a callback routine for the dialog box that resets the menu item's sensitivity. The code fragment in Example 19-7 shows these callback routines.

Example 19-7 The reset_sensitive() and open_callback() routines

```
/* reset_sensitive() -- generalized routine that resets the
** sensitivity on the widget passed as the client_data parameter
** in a call to XtAddCallback().
*/
void reset_sensitive (Widget w, XtPointer client_data, XtPointer call_data)
{
    Widget reset_widget = (Widget) client_data;

    XtSetSensitive (reset_widget, True);
}

/* open_callback() -- the callback routine for when the "Open"
** menu item is selected from the "File" title in the MenuBar.
*/
void open_callback (Widget menu_item, XtPointer client_data,
                    XtPointer call_data)
{
    Widget dialog, parent = menu_item;

    /* Get the window manager shell widget associated with item */
    while (!XtIsWMShell (parent))
        parent = XtParent (parent);

    /* turn off the sensitivity for the Open button... */
    XtSetSensitive (menu_item, False);

    dialog = XmCreateFileSelectionDialog (parent, "files", NULL, 0);

    /* Add callback routines to respond to OK button selection here. */
```

* This behavior is not a great design. The dialog really should be cached, and the menu item should remain sensitive. If the item is reselected, the dialog should be re-mapped or raised to the top of the window stack, if necessary.

```
        /* Make sure that if the dialog is popped down or destroyed, the
        ** menu_item's sensitivity is reset.
        */
        XtAddCallback (XtParent (dialog), /* dialog's _parent_ */
        XmNpopdownCallback, reset_sensitive, (XtPointer) menu_item);
        XtAddCallback (dialog, XmNdestroyCallback,
                               reset_sensitive, (XtPointer) menu_item);
        XtManageChild (dialog);
    }
```

The `open_callback()` function is called whenever the user activates the *Open* menu item on the *File* menu. The first thing `open_callback()` does is find the nearest WMShell widget associated with the menu item. We do not want the MenuShell here, as we need a non-transient widget to act as the parent for the FileSelectionDialog. If the menu item is used as the parent for the dialog, when the menu is popped down, the dialog is also popped down because it is a secondary window.

We set the menu item's sensitivity to `False`, which prevents the user from selecting the item again. In order to be notified when the FileSelectionDialog is dismissed, we add callback routines for `XmNpopdownCallback` and `XmNdestroyCallback`. In both cases, the *Open* menu item needs to be reset so that the user can select it again. The only thing in `open_callback()` is a callback function that opens the selected file when the user selects the *OK* button. This functionality is beyond the scope of this chapter; see Chapter 6, *Selection Dialogs*, for details.

Tear-Off Menus

Motif provides a feature that allows menus to be torn off and placed in separate windows. From the user's perspective, tear-off menus make it easy to make repeated menu selections. Normally, when the user posts a menu, it is only displayed until she makes a selection, and then it is removed. If the menu has been torn off, however, it is displayed in a separate window, and the user can make as many selections as she wants without having to repost it each time.

Tear-off behavior is provided for all of the Motif menu types, but the behavior is disabled by default. When tear-off functionality is enabled in a menu, the first item in the menu is a tear-off button. The button displays a dashed line to indicate that the user can tear off the menu, much as she would tear a coupon out of a newspaper. If the user selects the tear-off button, the menu is placed in a separate window with limited window manager decorations. The window can be moved, so the user can position it in a convenient location. The menu remains torn off until the user cancels the menu by pressing the ESCAPE key within the window.

Tear-off functionality is controlled by the `XmNtearOffModel` resource of the RowColumn widget. This resource is only valid when the RowColumn is being used as a PulldownMenu or a PopupMenu. The resource can have one of the following values:

XmTEAR_OFF_ENABLED or XmTEAR_OFF_DISABLED. By default, the resource is set to XmTEAR_OFF_DISABLED, so if you want to provide tear-off functionality in the menus in your application, you must set the resource for all of your menu panes. Figure 19-4 showed a PulldownMenu both before and after being torn off.

You can use the following resource specification to enable tear-off functionality for all menus:*

```
*tearOffModel: TEAR_OFF_ENABLED
```

Some applications use menus in such a way that they need to keep track of when the menu is popped up and popped down. For example, an application might use some ToggleButtons in a PulldownMenu to allow the user to set state variables for the program. If the application also provides another interface for changing the variables, such as a command-line, the application needs to know when the menu is popped up so that it can make sure the ToggleButtons are set appropriately. If you think that your application state may be affected by the user enabling tear-off functionality in her resource files, you should either explicitly disable the feature in your code, or install some callbacks to keep track of the tear-off state.

The RowColumn widget provides two callback resources that allow an application to keep track of tear-off menus. The XmNtearOffMenuActivateCallback routine is called when a menu is torn off; XmNtearOffMenuDeactivateCallback is called when the torn-off menu is dismissed. These callbacks provide a way for you to perform any special processing that is necessary for handling tear-off menus.

Motif also provides access to the tear-off button with the XmGetTearOffControl() routine. This routine takes a menu pane and returns the widget ID of the tear-off button in the menu, if there is one. Otherwise the routine returns NULL.The tear-off button has a Separator-like appearance; you can specify its background, foreground, and top and bottom shadow colors using the standard resources, as well as the XmNseparatorType resource. You can also set these resources in a resource file using the name of the button, which is TearOffControl.

In Motif 2.0 and later, the title of the dialog which contains the torn-off menu can be set through the RowColumn resource XmNtearOffTitle. This resource is a compound strings value, and you are referred to Chapter 25, *Compound Strings*, for more information on.

* Motif 1.2 does not install a resource converter for the XmNtearOffModel resource: you need to call XmRepTypeInstallTearOffModelConverter() manually. Motif 2.0 installs a converter for the type automatically, and thus XmRepTypeInstallTearOffModelConverter() is deprecated in later versions of the toolkit.

General Menu Creation Techniques

Now we have addressed each of the fundamental elements of the MenuBar and the resources used to provide the user with the appropriate feedback. Using this information, we can generalize the way we build MenuBars, enabling us to create arbitrarily large MenuBars and PulldownMenus using a substantially smaller amount of code.

In the examples that follow, we use many of the recommended elements for a standard Motif MenuBar. You can adjust the algorithms and data structures to fit the needs of your own application. Although we use hard-coded values for widget resources, this technique is by no means a requirement, nor should it be construed as recommended usage. If you choose to specify resources in a resource file, you should write an application defaults file that contains the appropriate resource values.

Building Pulldown Menus

Let's begin by identifying each of the attributes of a menu item:

- Label
- Mnemonic
- Accelerator
- Accelerator text
- Callback routine
- Callback data

Using this information, we can construct a data structure that describes all of the important aspects of a menu item. We define the `MenuItem` structure as follows:

```
typedef struct _menu_item {
    char         *label;          /* the label for the item */
    WidgetClass  *class;          /* pushbutton, label, separator,... */
    char         mnemonic;        /* mnemonic; NULL if none */
    char         *accelerator;    /* accelerator; NULL if none */
    char          *accel_text;     /* to be converted to compound string */
    void         (*callback)();   /* routine to call; NULL if none */
    XtPointer    callback_data;   /* client_data for callback() */
} MenuItem;
```

To create a PulldownMenu, all we need to do is initialize an array of `MenuItem` structures and pass it to a routine that iterates through the array and creates the items using the appropriate information. For example, the following declaration describes the elements for a *File* menu:

```
MenuItem file_items[] = {
{"New", &xmPushButtonGadgetClass, 'N', NULL, NULL, do_open, NEW},
{"Open...", &xmPushButtonGadgetClass, 'O', NULL, NULL, do_open, OPEN},
{"Save", &xmPushButtonGadgetClass, 'S', NULL, NULL, do_save, SAVE},
```

```
{"Save As...", &xmPushButtonGadgetClass, 'A',  NULL, NULL, do_save, SAVE_AS},
{"Print...", &xmPushButtonGadgetClass, 'P', NULL, NULL, do_print, NULL},
{"", &xmSeparatorGadgetClass, NULL, NULL, NULL, NULL, NULL},
{"Exit", &xmPushButtonGadgetClass, 'x', "Ctrl<Key>C", "Ctrl+C", do_quit, NULL},
{NULL, NULL, NULL, NULL, NULL, NULL, NULL}
};
```

Each element in the `MenuItem` data structure is filled with default values for each menu item. If a resource value is not meaningful, or is not going to be hard-coded, we initialize the field to `NULL`. If you don't need a callback function or client data for an item, the field may be set to `NULL`. The only field that cannot be `NULL` in a valid entry is the widget class. The final terminating `NULL` entry indicates the end of the list.

We have not specified any accelerators except for the *Exit* item. The Separator gadget is completely unspecified, since none of the resources even apply to Separators. This design makes modification and maintenance very simple. If you want to add an accelerator for the *Save* item, all you need to do is change the appropriate fields in the data structure, instead of having to search through the source code looking for where that item is created.

One particular point of interest is the way the `WidgetClass` field is initialized. It is declared as a pointer to a widget class rather than just a widget class, so we initialize the field with the address of the widget class variable that is declared in the widget's header file. The use of `&xmPushButtonGadgetClass` is one such example. The structure must be initialized this way because the compiler requires a specific value in order to initialize a static data structure. The `xmPushButtonWidgetClass` pointer does not have a value until the program is actually running, but the address of the variable does have a value. Once the program is running, the pointer can be dereferenced to access the real PushButton widget class.

Now we can write a routine that uses the `MenuItem` data structure to create a PulldownMenu. The `BuildPulldownMenu()` function is shown in Example 19-8. The routine loops through each element in an array of pre-initialized `MenuItem` structures and creates menu items based on the information.

Example 19-8 The BuildPulldownMenu() routine

```
Widget BuildPulldownMenu (Widget parent, char *menu_title,
                          char menu_mnemonic, Boolean tear_off,
                          MenuItem *items)
{
    Widget       pulldown, cascade, widget;
    int          i, n;
    XmString     str;
    Arg          args[4];

    pulldown = XmCreatePulldownMenu (parent, "_pulldown", NULL, 0);

    if (tear_off)
        XtVaSetValues (pulldown, XmNtearOffModel, XmTEAR_OFF_ENABLED, NULL);
```

```
        str = XmStringCreateLocalized (menu_title);
        n = 0;
        XtSetArg (args[n], XmNsubMenuId, pulldown); n++;
        XtSetArg (args[n], XmNlabelString, str); n++;
        XtSetArg (args[n], XmNmnemonic, menu_mnemonic); n++;
        cascade = XmCreateCascadeButton (parent, menu_title, args, n);
        XtManageChild (cascade);
        XmStringFree (str);

        /* Now add the menu items */
        for (i = 0; items[i].label != NULL; i++) {
            widget = XtVaCreateManagedWidget (items[i].label, *items[i].class,
                                              pulldown, NULL);
            if (items[i].mnemonic)
                XtVaSetValues (widget, XmNmnemonic, items[i].mnemonic, NULL);
            if (items[i].accelerator) {
                str = XmStringCreateLocalized (items[i].accel_text);
                XtVaSetValues (widget, XmNaccelerator, items[i].accelerator,
                                    XmNacceleratorText, str, NULL);
                XmStringFree (str);
            }
            if (items[i].callback)
                XtAddCallback (widget, XmNactivateCallback, items[i].callback,
                                       (XtPointer) items[i].callback_data);
        }

        return cascade;
    }
```

The function takes five parameters. `parent` is a handle to a MenuBar widget that must have already been created, `menu_title` indicates the title of the menu, `menu_mnemonic` specifies the mnemonic, `tear_off` indicates whether or not the menu can be torn off, and `items` is an array of `MenuItem` structures.

The first thing the routine does is create a PulldownMenu. Since the name of this widget is not terribly important, we use a predefined name, prefixed with an underscore, to indicate that the name is not intended to be referenced in a resource file. This use of the underscore is our own convention, by the way, not one adopted by the X Toolkit Intrinsics. We came up with this "unwritten rule" because Xt has no such naming conventions for widgets that do not wish to have their resources specified externally.

After creating the PulldownMenu, the routine creates the CascadeButton that acts as the title for the menu on the MenuBar. The name of the widget is taken from the second parameter, `menu_title`. The routine also sets the mnemonic and the `XmNtearOffModel` resource at this point. All MenuBar titles should have mnemonics associated with them.

Now the function loops through the array of `MenuItem` structures creating menu items until it finds an entry with a `NULL` name. We use this value as an end-of-menu indicator in

our initialization. When each widget is created, the mnemonic, accelerator, and callback function are added only if they are specified in the MenuItem structure.

BuildPulldownMenu() must be called from another function that passes the appropriate data structures and other parameters. In our design, this would be the routine that creates the MenuBar itself. Example 19-9 shows the code for the CreateMenuBar() routine. This simple function creates a MenuBar widget, calls BuildPulldownMenu() for each menu, manages the MenuBar, and returns it to the calling function.

Example 19-9 The CreateMenuBar() routine

```
Widget CreateMenuBar (Widget MainWindow)
{
    Widget mbar, widget;
    Widget BuildPulldownMenu (Widget, char *, char, Boolean , MenuItem *);

    mbar = XmCreateMenuBar (MainWindow, "MenuBar", NULL, 0);

    (void) BuildPulldownMenu (mbar, "File", 'F', True, file_items);
    (void) BuildPulldownMenu (mbar, "Edit", 'E', True, edit_items);
    (void) BuildPulldownMenu (mbar, "View", 'V', True, view_items);
    (void) BuildPulldownMenu (mbar, "Options", 'O', True, options_items);
    widget = BuildPulldownMenu (mbar, "Help", 'H', True, help_items);

    XtVaSetValues (mbar, XmNmenuHelpWidget, widget, NULL);
    XtManageChild (mbar);
    return mbar;
}
```

Each call to BuildPulldownMenu() passes an array of pre-initialized MenuItem structures. The *Help* menu is a special case, so we set the XmNmenuHelpWidget resource to let the MenuBar know which item it is. By setting the resource to the CascadeButton returned by the function, the MenuBar knows that this button should be placed to the far right. The only parameter to the CreateMenuBar() function is the MainWindow widget that is the parent of the MenuBar that is returned.

Building Cascading Menus

We can add pullright menus to our menu creation methodology quite easily by adding to the MenuItem data structure and making a slight modification to the CreatePulldownMenu() function. As we learned from the simple menu creation routines, a cascading menu is really a PulldownMenu that is associated with a CascadeButton. We also know that we can attach a menu to a CascadeButton by setting the XmNsubMenuId resource to the handle of the PulldownMenu. We begin by modifying the MenuItem structure as follows:

```
typedef struct _menu_item {
    char               *label;        /* the label for the item */
    WidgetClass        *class;        /* pushbutton, label, separator... */
```

```
    char               mnemonic;      /* mnemonic; NULL if none */
    char               *accelerator;  /* accelerator; NULL if none */
    char               *accel_text;   /* to convert to compound string */
    void               (*callback)(); /* routine to call; NULL if none */
    XtPointer          callback_data; /* client_data for callback() */
    struct _menu_item  *subitems; /* pullright menu items, if not NULL */
} MenuItem;
```

The new field at the end of the structure is a pointer to another array of `MenuItem` structures. If this pointer is not `NULL`, the menu item has a cascading submenu that is described by `subitems`. Example 19-10 shows an example of creating a cascading menu. This program uses a modified version of `BuildPulldownMenu()` that calls itself to create cascading menus.*

Example 19-10 The build_menu.c program

```
/* build_menu.c -- Demonstrate the BuildPulldownMenu() routine and
** how it can be used to build pulldown -and- pullright menus.
** Menus are defined by declaring an array of MenuItem structures.
*/

#include <Xm/RowColumn.h>
#include <Xm/MainW.h>
#include <Xm/DrawingA.h>
#include <Xm/CascadeBG.h>
#include <Xm/PushB.h>
#include <Xm/PushBG.h>
#include <Xm/ToggleB.h>
#include <Xm/ToggleBG.h>

typedef struct _menu_item {
    char              *label;        /* the label for the item */
    WidgetClass       *class;        /* pushbutton, label, separator... */
    char              mnemonic;      /* mnemonic; NULL if none */
    char              *accelerator;  /* accelerator; NULL if none */
    char               *accel_text;   /* to be converted to compound string */
    void              (*callback)(); /* routine to call; NULL if none */
    XtPointer         callback_data; /* client_data for callback() */
    struct _menu_item *subitems;     /* pullright menu items, if not NULL */
} MenuItem;

/* Pulldown menus are built from cascade buttons, so this function
** also includes pullright menus. Create the menu, the cascade button
** that owns the menu, and then the submenu items.
*/
Widget BuildPulldownMenu (Widget parent, char *menu_title,
                          char menu_mnemonic,
                          Boolean tear_off,
                          MenuItem *items)
{
    Widget       PullDown, cascade, widget;
```

* `XtVaAppInitialize()` is considered deprecated in X11R6.

```
int          i;
XmString     str;
Arg          args[8];
int          n;

PullDown = XmCreatePulldownMenu (parent, "_pulldown", NULL, 0);

if (tear_off)
    XtVaSetValues (PullDown, XmNtearOffModel, XmTEAR_OFF_ENABLED, NULL);
str = XmStringCreateLocalized (menu_title);
n = 0;
XtSetArg (args[n], XmNsubMenuId, PullDown); n++;
XtSetArg (args[n], XmNlabelString, str); n++;
XtSetArg (args[n], XmNmnemonic, menu_mnemonic); n++;
cascade = XmCreateCascadeButtonGadget (parent, menu_title, args, n);
XtManageChild (cascade);
XmStringFree (str);

/* Now add the menu items */
for (i = 0; items[i].label != NULL; i++) {
    /* If subitems exist, create the pull-right menu by calling this
    ** function recursively. Since the function returns a cascade
    ** button, the widget returned is used.
    */
    if (items[i].subitems)
        widget = BuildPulldownMenu (PullDown, items[i].label, items[i].
                                    mnemonic, tear_off, items[i].subitems);
    else
        widget = XtVaCreateManagedWidget (items[i].label,
                                          *items[i].class,
                                          PullDown,
                                          NULL);

    /* Whether the item is a real item or a cascade button with a
    ** menu, it can still have a mnemonic.
    */
    if (items[i].mnemonic)
        XtVaSetValues (widget, XmNmnemonic, items[i].mnemonic, NULL);

    /* any item can have an accelerator, except cascade menus. But,
    ** we don't worry about that; we know better in our declarations.
    */
    if (items[i].accelerator) {
        str = XmStringCreateLocalized (items[i].accel_text);
        XtVaSetValues (widget, XmNaccelerator, items[i].accelerator,
                                       XmNacceleratorText, str, NULL);
        XmStringFree (str);
    }

    if (items[i].callback) {
        String resource;

        if (XmIsToggleButton (widget) || (XmIsToggleButtonGadget (widget))
            resource = XmNvalueChangedCallback;
```

```
            else
                resource = XmNactivateCallback;

            XtAddCallback (widget, resource, items[i].callback,
                           (XtPointer) items[i].callback_data);
        }
    }

    return cascade;
}

/* callback functions for menu items declared later... */
void set_weight (Widget widget, XtPointer client_data, XtPointer call_data)
{
    int weight = (int) client_data;
    printf ("Setting line weight to %d\n", weight);
}

void set_color (Widget widget, XtPointer client_data, XtPointer call_data)
{
    char *color = (char *) client_data;
    printf ("Setting color to %s\n", color);
}

void set_dot_dash (Widget widget, XtPointer client_data, XtPointer call_data)
{
    int dot_or_dash = (int) client_data;
    printf ("Setting line style to %s\n", dot_or_dash? "dot" : "dash");
}

MenuItem weight_menu[] = {
    { " 1 ", &xmPushButtonGadgetClass, '1', NULL, NULL, set_weight,
                (XtPointer) 1, (MenuItem *) NULL },
    { " 2 ", &xmPushButtonGadgetClass, '2', NULL, NULL, set_weight,
                (XtPointer) 2, (MenuItem *) NULL },
    { " 3 ", &xmPushButtonGadgetClass, '3', NULL, NULL, set_weight,
                (XtPointer) 3, (MenuItem *) NULL },
    { " 4 ", &xmPushButtonGadgetClass, '4', NULL, NULL, set_weight,
                (XtPointer) 4, (MenuItem *) NULL },
    { NULL. NULL, NULL, NULL, NULL, NULL, NULL, NULL }
};

MenuItem color_menu[] = {
    { "Cyan", &xmPushButtonGadgetClass, 'C', "Alt<Key>C", "Alt+C", set_color,
                (XtPointer) "cyan", (MenuItem *) NULL },
    { "Yellow", &xmPushButtonGadgetClass, 'Y', "Alt<Key>Y", "Alt+Y", set_color,
                (XtPointer) "yellow", (MenuItem *) NULL },
    { "Magenta", &xmPushButtonGadgetClass, 'M', "Alt<Key>M", "Alt+M",
                set color,(XtPointer) "magenta", (MenuItem *) NULL },
    { "Black", &xmPushButtonGadgetClass, 'B', "Alt<Key>B", "Alt+B", set_color,
                (XtPointer) "black", (MenuItem *) NULL },
    { NULL. NULL, NULL, NULL, NULL, NULL, NULL, NULL }
};
```

```
MenuItem style_menu[] = {
    { "Dash", &xmPushButtonGadgetClass, 'D', NULL, NULL, set_dot_dash,
                (XtPointer) 0, (MenuItem *) NULL },
    { "Dot", &xmPushButtonGadgetClass, 'o', NULL, NULL, set_dot_dash,
                (XtPointer) 1, (MenuItem *) NULL },
    { NULL. NULL, NULL, NULL, NULL, NULL, NULL, NULL }
};

MenuItem drawing_menus[] = {
    { "Line Weight", &xmCascadeButtonGadgetClass, 'W', NULL, NULL, 0, 0,
                weight_menu },
    { "Line Color", &xmCascadeButtonGadgetClass, 'C', NULL, NULL, 0, 0,
                color_menu },
    { "Line Style", &xmCascadeButtonGadgetClass, 'S', NULL, NULL, 0, 0,
                style_menu },
    { NULL. NULL, NULL, NULL, NULL, NULL, NULL, NULL }
};

main (int argc, char *argv[])
{
    Widget         toplevel, main_w, menubar, drawing_a;
    XtAppContext   app;
    Arg            args[4];
    int            n;

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

    /* Create a MainWindow widget that contains a DrawingArea in
    ** its work window.
    */
    n = 0;
    XtSetArg (args[n], XmNscrollingPolicy, XmAUTOMATIC); n++;
    main_w = XmCreateMainWindow (toplevel, "main_w", args, n);
    menubar = XmCreateMenuBar (main_w, "menubar", NULL, 0);
    BuildPulldownMenu (menubar, "Lines", 'L', True, drawing_menus);
    XtManageChild (menubar);

    /* Create a DrawingArea -- no actual drawing will be done. */
    n = 0;
    XtSetArg (args[n], XmNwidth, 500);  n++;
    XtSetArg (args[n], XmNheight, 500); n++;
    drawing_a = XmCreateDrawingArea (main_w, "drawing_a", args, n);
    XtManageChild (drawing_a);
    XtManageChild (main_w);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}
```

The majority of this program is composed of the new version of `BuildPulldownMenu()` and the menu and submenu declarations. All the menus and menu items are declared in

reverse order because the cascading menu declaration must exist before the menu is actually referenced. The output of the program is shown in Figure 19-11.

Figure 19-11 Output of the build_menu program

All we have to do to get `BuildPulldownMenu()` to create a cascading menu is add code that checks whether or not the current menu has a submenu. If it does, the routine calls itself to create the submenu. Because the function creates and returns a CascadeButton, the return value can be used as the menu item in the menu that is currently being built. We have to create the cascading menu first because it has to exist before it can be attached to a CascadeButton. Recursion handles this problem for us by creating the deepest submenus first, which ensures that all the necessary submenus are built before their CascadeButtons require them.

We also added support for ToggleButtons to this version of `BuildPulldownMenu()`, even though our menus do not contain any ToggleButtons. The only change that we have to make here involves the callback function. Since ToggleButtons have an `XmNvalueChangedCallback`, while PushButtons have an `XmNactivateCallback`, we check the class of the item being added and specify the appropriate callback resource in our call to `XtAddCallback()`.

Building Popup Menus

To further demonstrate the flexibility of our design and to exploit the similarities between PulldownMenus, PopupMenus, and cascading menus, we can easily modify the `BuildPulldownMenu()` routine to support any of these menu types. We only need to specify a new parameter indicating which of the two menu types to use. Since Motif already defines the values `XmMENU_PULLDOWN` and `XmMENU_POPUP` in *<Xm/Xm.h>*, we use those values. We have also given the function a more generic name, `BuildMenu()`, as shown in Example 19-11.*

* Automatic popup as provided by the toolkit is only available from Motif 2.0 onwards.

Example 19-11 The BuildMenu() routine

```
Widget BuildMenu (Widget parent, int menu_type, char *menu_title,
                  char menu_mnemonic, Boolean tear_off, MenuItem *items)
{
    Widget       menu, cascade, widget;
    int          i;
    XmString     str;
    Arg          args[8];
    int          n;

    if (menu_type == XmMENU_PULLDOWN)
        menu = XmCreatePulldownMenu (parent, "_pulldown", NULL, 0);
    else {
        n = 0;
        XtSetArg (args[n], XmNpopupEnabled, XmPOPUP_AUTOMATIC_RECURSIVE); n++;
        menu = XmCreatePopupMenu (parent, "_popup", args, n);
    }

    if (tear_off)
        XtVaSetValues (menu, XmNtearOffModel, XmTEAR_OFF_ENABLED, NULL);

    if (menu_type == XmMENU_PULLDOWN) {
        str = XmStringCreateLocalized (menu_title);
        n = 0;
        XtSetArg (args[n], XmNsubMenuId, menu);         n++;
        XtSetArg (args[n], XmNlabelString, str);        n++;
        XtSetArg (args[n], XmNmnemonic, menu_mnemonic); n++;
        cascade = XmCreateCascadeButtonGadget (parent, menu_title, args, n);
        XtManageChild (cascade);
        XmStringFree (str);
    }

    /* Now add the menu items */
    for (i = 0; items[i].label != NULL; i++) {
        /* If subitems exist, create the pull-right menu by calling this
        ** function recursively. Since the function returns a cascade
        ** button, the widget returned is used.
        */
        if (items[i].subitems)
            widget = BuildMenu (menu, XmMENU_PULLDOWN,
                                items[i].label,
                                items[i].mnemonic,
                                tear_off,
                                items[i].subitems);
        else
            widget = XtVaCreateManagedWidget (items[i].label,
                                *items[i].class,
                                menu,
                                NULL);

        /* Whether the item is a real item or a cascade button with a
        ** menu, it can still have a mnemonic.
        */
```

```
            if (items[i].mnemonic)
                XtVaSetValues (widget, XmNmnemonic, items[i].mnemonic, NULL);

            /* any item can have an accelerator, except cascade menus. But,
            ** we don't worry about that; we know better in our declarations.
            */
            if (items[i].accelerator) {
                str = XmStringCreateLocalized (items[i].accel_text);
                XtVaSetValues (widget, XmNaccelerator, items[i].accelerator,
                               XmNacceleratorText, str, NULL);
                XmStringFree (str);
            }

            if (items[i].callback) {
                String resource;

                if (XmIsToggleButton (widget) || (XmIsToggleButtonGadget (widget))
                    resource = XmNvalueChangedCallback;
                else
                    resource = XmNactivateCallback;

                XtAddCallback (widget, resource, items[i].callback,
                                    (XtPointer) items[i].callback_data);
            }
        }

        return (menu_type == XmMENU_PULLDOWN? cascade: menu);
    }
```

All of the original functionality is maintained; we only added a few lines to support popup menus. Namely, when `XmMENU_POPUP` is passed as the `menu_type` parameter, the function `XmCreatePopupMenu()` is called, and the menu itself is returned. Otherwise the routine returns a CascadeButton. If any of the menu items have cascading menus, we continue what we were doing before for submenus.

Now we can build PopupMenus, but what we really need to talk about is when you should use PopupMenus in an application. The *Motif Style Guide* has very little to say about when and how popup menus should be used. One guideline is that PopupMenus should only be used as a redundant means of activating application functionality, since they do not make themselves apparent to the user. The single requirement is that PopupMenus use the third mouse button, which leads to the question: how do you get the necessary events on an arbitrary widget so that you can pop up a menu?

The design of PopupMenus in the Motif 1.2 toolkit requires you to dig into lower-level Xt event-handling mechanisms in order to post a PopupMenu. In Motif 2.0 and later, this is not necessary: we can set the `XmNpopupEnabled` resource to `XmPOPUP_AUTOMATIC` or `XmPOPUP_AUTOMATIC_RECURSIVE` on the menu, and the toolkit does the rest. In Example 19-11, we specified the value as `XmPOPUP_AUTOMATIC_RECURSIVE` so that widgets in the hierarchy can inherit our popup menus: we may wish to display a popup menu for a Gadget, and inherit the popup from the manager parent. If we need to choose

between a number of popup menus at any given point, we only need to register an XmNpopupHandlerCallback on the widget where the popup is required. Example 19-12 demonstrates how to display a PopupMenu for an arbitrary widget.

In the example, we parent the popup menu off the RowColumn rowcol: pressing the 3rd mouse button over either the PushButton widget or gadget children displays the menu. If we only wanted to display the menu for the PushButton widget, we would parent the menu off the button itself. This program uses the BuildMenu() routine from Example 19-11, so we do not show it in this example.*

Example 19-12 The popups.c program

```
/* popups.c -- demonstrate the use of a popup menus in an arbitrary
** widget. Display two PushButtons. The second one has a popup
** menu attached to it that is activated with the third
** mouse button.
*/

#include <Xm/LabelG.h>
#include <Xm/PushBG.h>
#include <Xm/PushB.h>
#include <Xm/ToggleBG.h>
#include <Xm/ToggleB.h>
#include <Xm/SeparatoG.h>
#include <Xm/RowColumn.h>
#include <Xm/FileSB.h>
#include <Xm/CascadeBG.h>

Widget       toplevel;
extern void  exit(int);
void         open_dialog_box(Widget, XtPointer, XtPointer);

/* callback for pushbutton activation */
void put_string (Widget w, XtPointer client_data, XtPointer call_data)
{
    String str = (String) client_data;
    puts (str);
}

typedef struct _menu_item {
    char               *label;
    WidgetClass        *class;
    char               mnemonic;
    char               *accelerator;
    char               *accel_text;
    void               (*callback)();
    XtPointer          callback_data;
    struct _menu_item  *subitems;
```

* XtVaAppInitialize() is considered deprecated in X11R6. There is no XmNpopupHandlerCallback resource in Motif 1.2: an event handler must be installed in order to display a popup menu, which means that Gadgets did not support popup menus in this version of the toolkit.

```
} MenuItem;

MenuItem file_items[] = {
    {"File Items", &xmLabelGadgetClass, NULL, NULL, NULL, NULL, NULL, NULL},
    {"_sep1", &xmSeparatorGadgetClass, NULL, NULL, NULL, NULL, NULL, NULL},
    {"New", &xmPushButtonGadgetClass, 'N', NULL, NULL, put_string, "New",
                            NULL},
    {"Open...", &xmPushButtonGadgetClass, 'O', NULL, NULL, open_dialog_box,
                            (XtPointer) XmCreateFileSelectionDialog, NULL},
    {"Save", &xmPushButtonGadgetClass, 'S', NULL, NULL, put_string, "Save",
                            NULL},
    {"Save As...", &xmPushButtonGadgetClass, 'A', NULL, NULL, open_dialog_box,
                            (XtPointer) XmCreateFileSelectionDialog, NULL},
    {"Exit", &xmPushButtonGadgetClass, 'x', "Ctrl<Key>C", "Ctrl+C", exit, NULL,
                            NULL},
    {NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL}
};

/* build_menu.c -- Demonstrate the BuildMenu() routine and
** how it can be used to build pulldown -and- pullright menus.
** Menus are defined by declaring an array of MenuItem structures.
*/

/* Pulldown menus are built from cascade buttons, so this function
** also includes pullright menus. Create the menu, the cascade button
** that owns the menu, and then the submenu items.
*/
Widget BuildMenu (Widget parent, int menu_type, char *menu_title,
                char menu_mnemonic, Boolean tear_off, MenuItem *items)
{
    Widget       menu, cascade, widget;
    int          i;
    XmString     str;
    Arg          args[8];
    int          n;

    if (menu_type == XmMENU_PULLDOWN)
        menu = XmCreatePulldownMenu (parent, "_pulldown", NULL, 0);
    else {
        n = 0;
        XtSetArg (args[n], XmNpopupEnabled, XmPOPUP_AUTOMATIC_RECURSIVE); n++;
        menu = XmCreatePopupMenu (parent, "_popup", args, n);
    }

    if (tear_off)
        XtVaSetValues (menu, XmNtearOffModel, XmTEAR_OFF_ENABLED, NULL);

    if (menu_type == XmMENU_PULLDOWN) {
        str = XmStringCreateLocalized (menu_title);
        n = 0;
        XtSetArg (args[n], XmNsubMenuId, menu);         n++;
        XtSetArg (args[n], XmNlabelString, str);        n++;
        XtSetArg (args[n], XmNmnemonic, menu_mnemonic); n++;
        cascade = XmCreateCascadeButtonGadget (parent, menu_title, args, n);
```

```
            XtManageChild (cascade);
            XmStringFree (str);
        }

        /* Now add the menu items */
        for (i = 0; items[i].label != NULL; i++) {
            /* If subitems exist, create the pull-right menu by calling this
            ** function recursively. Since the function returns a cascade
            ** button, the widget returned is used.
            */

            if (items[i].subitems)
                widget = BuildMenu (menu, XmMENU_PULLDOWN,
                                    items[i].label,
                                    items[i].mnemonic,
                                    tear_off,
                                    items[i].subitems);
            else
                widget = XtVaCreateManagedWidget (items[i].label, *items[i].class,
                                                  menu, NULL);

            /* Whether the item is a real item or a cascade button with a
            ** menu, it can still have a mnemonic.
            */
            if (items[i].mnemonic)
                XtVaSetValues (widget, XmNmnemonic, items[i].mnemonic, NULL);

            /* any item can have an accelerator, except cascade menus. But,
            ** we don't worry about that; we know better in our declarations.
            */
            if (items[i].accelerator) {
                str = XmStringCreateLocalized (items[i].accel_text);

                XtVaSetValues (widget, XmNaccelerator, items[i].accelerator,
                               XmNacceleratorText, str, NULL);
                XmStringFree (str);
            }

            if (items[i].callback) {
                String resource;

            if (XmIsToggleButton (widget) || XmIsToggleButtonGadget (widget))
                resource = XmNvalueChangedCallback;
            else
                resource = XmNactivateCallback;

            XtAddCallback (widget, resource, items[i].callback,
                                    (XtPointer) items[i].callback_data);
            }
        }

        return (menu_type == XmMENU_PULLDOWN)? cascade: menu;
    }
```

```
main (int argc, char *argv[])
{
    Widget         button, rowcol, popup;
    XtAppContext   app;

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

    /* Build a RowColumn to contain two PushButtons */
    rowcol = XtVaCreateManagedWidget ("rowcol", xmRowColumnWidgetClass,
                                    toplevel, NULL);

    /* The first PushButton is a gadget, just to show that the Motif 2.x
    ** popup routines work for Gadgets as well as widgets.
    */
    button = XmCreatePushButtonGadget (rowcol, "Button 1", NULL, 0);
    XtAddCallback (button, XmNactivateCallback, put_string, "Button 1");
    XtManageChild (button);

    /* This PushButton is a widget.
    */
    button = XmCreatePushButton (rowcol, "Button 2", NULL, 0);
    XtAddCallback (button, XmNactivateCallback, put_string, "Button 2");
    XtManageChild (button);

    /* build the menu... */
    popup = BuildMenu (rowcol, XmMENU_POPUP, "Stuff", NULL, True, file_items);

    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

/* open_dialog_box() -- callback for some of the menu items declared
** in the MenuItem struct. The client data is the creation function
** for the dialog. Associate the dialog with the menu
** item via XmNuserData so we don't have to keep a global and
** don't have to repeatedly create one.
*/
void open_dialog_box (Widget w, XtPointer client_data, XtPointer call_data)
{
    Widget (*func)() = (Widget (*)()) client_data;
    Widget dialog = NULL;

    /* first see if this menu item's dialog has been created yet */
    XtVaGetValues (w, XmNuserData, &dialog, NULL);

    if (!dialog) {
        /* menu item hasn't been chosen yet -- create the dialog.
        ** Use the toplevel as the parent because we don't want the
        ** parent of a dialog to be a menu item.
        */
        dialog = (*func)(toplevel, "dialog", NULL, 0);
        XtVaSetValues (XtParent (dialog), XmNtitle, XtName (w), NULL);
```

```
            XtVaSetValues (dialog, XmNautoUnmanage, True, NULL);

            /* store the newly created dialog in the XmNuserData for the menu
            ** item for easy retrieval next time. (see get-values above.)
            */
            XtVaSetValues (w, XmNuserData, dialog, NULL);
        }
        XtManageChild (dialog);

        /* If the dialog was already open, XtPopup does nothing. In
        ** this case, at least make sure the window is raised to the top
        ** of the window tree (or as high as it can get).
        */
        XRaiseWindow (XtDisplay (dialog), XtWindow (XtParent (dialog)));
    }
```

The output of the program is shown in Figure 19-12.

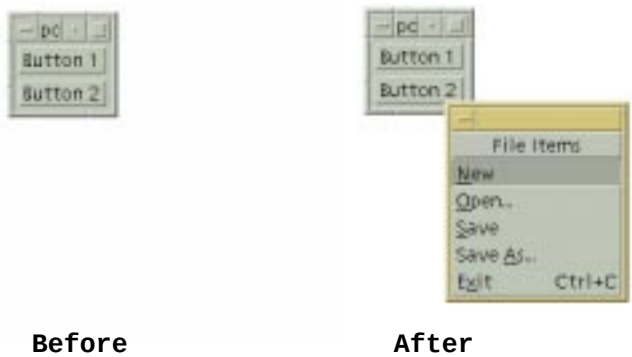


Figure 19-12 Output of the popups program

The program displays two PushButtons, one of which is a gadget and the other a widget. In Motif 1.2, we would need to catch `ButtonPress` events by specifically asking for them using `XtAddEventHandler()`. This routine requires a widget because it needs a window. To add an event handler for a gadget, you would have to install it on the gadget's parent, which is a manager widget. Any time a `ButtonPress` event occurs in the manager, the event handler would be called, so the event handler would have to check the coordinates of the event and see if it happened within the boundaries of the gadget. In Motif 2.0 and later, none of this is necessary: the toolkit does it all for us, and we only need to specify the `XmNpopupEnabled` and `XmNpopupHandlerCallback` resources as required. Motif 1.2 code will still work, however: `XtAddEventHandler()` takes the following form:

```
void XtAddEventHandler ( Widget           w,
                         EventMask        event_mask,
                         Boolean          nonmaskable,
                         XtEventHandler   proc,
                         XtPointer        client_data)
```

The *widget* parameter specifies the widget on which the event handler is to be installed, while *event_mask* identifies the events that are being handled. We would specify ButtonPressMask to indicate that we are interested in ButtonPress events. The *nonmaskable* argument indicates whether or not the event handler should be called on non-maskable events. We would specify False since we are not interested in the events. The final arguments specify the event handler routine and the client data that is passed to it. This routine would have to position the required popup menu (XmMenuPosition()), and display it (XtManageChild()). These are now called by Motif for us internally. See Volume 4, *X Toolkit Intrinsics Programming Manual*, for a complete list of event masks and more detailed information about XtAddEventHandler().

The RowColumn widget has a resource that you can set on PopupMenus called XmNmenuPost, which allows you to specify an alternate button to post the menu.

You may have noticed that the PopupMenu shown in Figure 19-12 has accelerators associated with it. These accelerators only take effect if the input focus is in the widget that contains the menu.

Building Option Menus

In this final section on generalized menu creation methods, we examine how to create OptionMenus using the BuildMenu() function. In this case, the underlying function is XmCreateOptionMenu(), which is another convenience routine provided by the Motif toolkit. The routine creates a RowColumn widget that manages the Label and CascadeButton widgets that define the OptionMenu, but we must create the actual PulldownMenu ourselves. The final version of the BuildMenu() function is shown in Example 19-13.

Example 19-13 The build_option.c program

```
/* build_option.c -- The final version of BuildMenu() is used to
** build popup, option, pulldown -and- pullright menus. Menus are
** defined by declaring an array of MenuItem structures as usual.
*/

#include <Xm/MainW.h>
#include <Xm/ScrolledW.h>
#include <Xm/PanedW.h>
#include <Xm/RowColumn.h>
#include <Xm/DrawingA.h>
#include <Xm/CascadeBG.h>
#include <Xm/ToggleB.h>
#include <Xm/ToggleBG.h>
#include <Xm/PushB.h>
#include <Xm/PushBG.h>

typedef struct _menu_item {
    char           *label;        /* the label for the item */
```

```
    WidgetClass    *class;        /* pushbutton, label, separator... */
    char           mnemonic;      /* mnemonic; NULL if none */
    char           *accelerator;  /* accelerator; NULL if none */
    char           *accel_text;   /* to be converted to compound string */
    void           (*callback)(); /* routine to call; NULL if none */
    XtPointer       callback_data; /* client_data for callback() */
    struct _menu_item  *subitems; /* pullright menu items, if not NULL */
} MenuItem;

/* Build popup, option and pulldown menus, depending on the menu_type.
** It may be XmMENU_PULLDOWN, XmMENU_OPTION or XmMENU_POPUP. Pulldowns
** return the CascadeButton that pops up the menu. Popups return the menu.
** Option menus are created, but the RowColumn that acts as the option
** "area" is returned unmanaged. (The user must manage it.)
** Pulldown menus are built from cascade buttons, so this function
** also builds pullright menus. The function also adds the right
** callback for PushButton or ToggleButton menu items.
*/
Widget BuildMenu (Widget parent, int menu_type, char *menu_title,
char menu_mnemonic, Boolean tear_off, MenuItem *items)
{
    Widget       menu, cascade, widget;
    int          i;
    XmString     str;
    Arg          args[4];
    int          n;

    if (menu_type == XmMENU_PULLDOWN || menu_type == XmMENU_OPTION)
        menu = XmCreatePulldownMenu (parent, "_pulldown", NULL, 0);
    else if (menu_type == XmMENU_POPUP) {
        n = 0;
        XtSetArg (args[n], XmNpopupEnabled, XmPOPUP_AUTOMATIC_RECURSIVE); n++;
        menu = XmCreatePopupMenu (parent, "_popup", args, n);
    }
    else {
        XtWarning ("Invalid menu type passed to BuildMenu()");
        return NULL;
    }

if (tear_off)
    XtVaSetValues (menu, XmNtearOffModel, XmTEAR_OFF_ENABLED, NULL);

    /* Pulldown menus require a cascade button to be made */
    if (menu_type == XmMENU_PULLDOWN) {
        str = XmStringCreateLocalized (menu_title);
        n = 0;
        XtSetArg (args[n], XmNsubMenuId, menu); n++;
        XtSetArg (args[n], XmNlabelString, str); n++;
        XtSetArg (args[n], XmNmnemonic, menu_mnemonic); n++;
        cascade = XmCreateCascadeButtonGadget (parent, menu_title, args, n);
        XtManageChild (cascade);
        XmStringFree (str);
    }
    else if (menu_type == XmMENU_OPTION) {
```

```
    /* Option menus are a special case, but not hard to handle */
    str = XmStringCreateLocalized (menu_title);
    n = 0;
    XtSetArg (args[n], XmNsubMenuId, menu); n++;
    XtSetArg (args[n], XmNlabelString, str); n++;

    /* This really isn't a cascade, but this is the widget handle
    ** we're going to return at the end of the function.
    */
    cascade = XmCreateOptionMenu (parent, menu_title, args, n);
    XmStringFree (str);
}

/* Now add the menu items */

for (i = 0; items[i].label != NULL; i++) {
    /* If subitems exist, create the pull-right menu by calling this
    ** function recursively. Since the function returns a cascade
    ** button, the widget returned is used.
    */

    if (items[i].subitems) {
        if (menu_type == XmMENU_OPTION) {
            XtWarning ("You can't have submenus from option menus.");
            continue;
        }
        else {
            widget = BuildMenu (menu, XmMENU_PULLDOWN,
            items[i].label,
            items[i].mnemonic,
            tear_off,
            items[i].subitems);
        }
    }
    else {
        widget = XtVaCreateManagedWidget (items[i].label,
        *items[i].class, menu, NULL);
    }

    /* Whether the item is a real item or a cascade button with a
    ** menu, it can still have a mnemonic.
    */

    if (items[i].mnemonic)
        XtVaSetValues (widget, XmNmnemonic, items[i].mnemonic, NULL);

    /* any item can have an accelerator, except cascade menus. But,
    ** we don't worry about that; we know better in our declarations.
    */

    if (items[i].accelerator) {
        str = XmStringCreateLocalized (items[i].accel_text);
        XtVaSetValues (widget, XmNaccelerator, items[i].accelerator,
                       XmNacceleratorText, str, NULL);
```

```
            XmStringFree (str);
        }
        if (items[i].callback) {
            String resource;

            if (XmIsToggleButton (widget) || XmIsToggleButtonGadget (widget))
                resource = XmNvalueChangedCallback;
            else
                resource = XmNactivateCallback;

            XtAddCallback (widget, resource, items[i].callback,
                           (XtPointer) items[i].callback_data);
        }
    }

    /* for popup menus, just return the menu; pulldown menus, return
    ** the cascade button; option menus, return the thing returned
    ** from XmCreateOptionMenu(). This isn't a menu, or a cascade button!
    */

    return (menu_type == XmMENU_POPUP? menu: cascade);
}

MenuItem drawing_shapes[] = {
    { "Lines", &xmPushButtonGadgetClass, 'L', NULL, NULL, 0, 0, NULL },
    { "Circles", &xmPushButtonGadgetClass, 'C', NULL, NULL, 0, 0, NULL },
    { "Squares", &xmPushButtonGadgetClass, 'S', NULL, NULL, 0, 0, NULL },
    { NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL }
};

main (int argc, char *argv[])
{
    Widget            toplevel, main_w, pane, sw, drawing_a, menu,
                      option_menu;
    Arg               args[4];
    int               n;
    XtAppContext      app;
    XtWidgetGeometry  geom;

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

    main_w = XmCreateMainWindow (toplevel, "main_w", NULL, 0);

    /* Use a PanedWindow widget as the work area of the main window */
    pane = XmCreatePanedWindow (main_w, "pane", NULL, 0);

    /* create the option menu -- don't forget to manage it. */
    option_menu = BuildMenu (pane, XmMENU_OPTION, "Shapes", 'S', True,
                             drawing_shapes);
    XtManageChild (option_menu);

    /* Set the OptionMenu so that it can't be resized */
```

```
    geom.request_mode = CWHeight;
    XtQueryGeometry (option_menu, NULL, &geom);
    XtVaSetValues (option_menu, XmNpaneMinimum, geom.height, XmNpaneMaximum,
                   geom.height, NULL);

    /* The scrolled window (which contains the drawing area) is a child
    ** of the PanedWindow; its sibling, the option menu, cannot be resized,
    ** so if the user resizes the toplevel shell, *this* window will resize.
    */
    n = 0;
    XtSetArg (args[n], XmNscrollingPolicy, XmAUTOMATIC); n++;
    sw = XmCreateScrolledWindow (pane, "sw", args, n);

    /* Create a DrawingArea -- no actual drawing will be done. */
    n = 0;
    XtSetArg (args[n], XmNwidth, 500);  n++;
    XtSetArg (args[n], XmNheight, 500); n++;
    drawing_a = XmCreateDrawingArea (sw, "drawing_a", args, n);
    XtManageChild (drawing_a);

    XtManageChild (sw);
    XtManageChild (pane);
    XtManageChild (main_w);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}
```

There are two particularly interesting features of this program. First, of course, is the modification of the `BuildMenu()` function. As the comments in the code indicate, the function now fully supports all of the Motif menu types.We use `XmCreatePulldownMenu()` to create the menu pane that is posted from the CascadeButton of the OptionMenu.This menu pane is attached to the OptionMenu by setting the `XmNsubMenuId` as usual. As we loop through the menu items that are to be placed in the menu, we prevent the creation of a pullright menu in an OptionMenu, as cascading menus are not allowed in OptionMenus.

When `BuildMenu()` is used to create an OptionMenu, the function returns the RowColumn widget that is returned by `XmCreateOptionMenu()`, even though it is not really a CascadeButton as the variable name might indicate. The calling function needs the RowColumn widget so that it can manage the OptionMenu by calling `XtManageChild()`. (The call to `XtManageChild()` might be another automated part of `BuildMenu()` if you want to modify it.)

The other interesting feature of the program is the layout of the MainWindow. The MainWindow widget has a single PanedWindow widget as its child because we wish to retain the vertical stacking relationship between the OptionMenu and the DrawingArea. Another advantage of using the PanedWindow is that we can set the maximum and minimum height of each pane. The user can resize the entire window using the window

manager, but we don't want the OptionMenu to change size, so we allow the ScrolledWindow to absorb the size fluctuations.

Summary

Menus are basically simple objects that provide the user with access to application functionality. While the simple menu creation routines are handy for basic prototyping and other simple application constructs, their usefulness is limited once you begin to develop larger-scale applications.

We have described the design of a general menu creation routine, so it should be clear that you only need two things to create an arbitrary number of menus: predefined arrays of `MenuItem` structures and the `BuildMenu()` function. Since initializing an array of `MenuItem` objects is very simple, our method is convenient and also more powerful than the simple menu creation routines. We have defined our own data type and generalized the routine to build menus so that you can use and modify these functions however you like, to conform to the needs of your application.

Exercises

This chapter could go on forever discussing more and more things you can do with menus. However, the goal was to present you with the fundamental concepts and design considerations behind menus. From this information, you should be able to teach yourself new techniques that we haven't touched upon.In that spirit, you should be able to do the following exercises based on the material covered in this chapter.

1. Create a MainWindow widget that has a MenuBar that contains at least the *File*, *Edit*, and *Help* menus, an OptionMenu, and a PopupMenu that pops up from a DrawingArea widget. First implement the menus using the simple menu creation routines, and then implement them using the `BuildMenu()` function.
2. Initialize a `MenuItem` structure whose fields are all set to `NULL` except for the menu items' names, callback routines, and widget classes, and then write a resource file that generates a usable menu.
3. Modify the `MenuItem` structure and the `BuildMenu` routine so that you can specify the initial sensitivity for menu items.
4. Modify `BuildMenu()` to recognize when the menu it is about to build is a RadioBox. You may choose to implement this behavior by passing a new parameter to the function or by examining the children in the `MenuItem` list to see if they are ToggleButtons. You will need to modify the `MenuItem` structure by adding another `Boolean` field to allow each element to indicate whether it is a radio button or a plain ToggleButton. See Chapter 4, *The Main Window*, for a discussion of RadioBoxes in menus.

In this chapter:

- *Interclient Communication*
- *Shell Resources*
- *VendorShell Resources*
- *Handling Window Manager Messages*
- *Session Management*
- *Customized Protocols*
- *Summary*
- *Exercises*

20

Interacting With the Window Manager

This chapter provides additional information on the relationship between shell widgets and the window manager. In particular, it focuses on the Motif window manager, mwm, although the information given pertains equally to the CDE desktop manager, as well as other ICCCM-compliant managers. It discusses shell widget resources and describes how to use functions in the Motif toolkit to add and modify window manager protocols.

This chapter provides technical details about how Motif applications can interact with the window manager. It discusses when and how to interpret special window manager events and client messages, how to set shell resources that act as hints to the window manager, and how to add protocols for communication between the application and the window manager. In the course of the discussion, we cover the major features of the X Toolkit Intrinsics' WMShell widget class, which handles basic window manager communication, and Motif's VendorShell widget, which handles window manager events that are specific to the Motif window manager (*mwm*), and the CDE desktop which is derived from it. In the discussions which follow, wherever the text refers to *mwm*, you should assume that the CDE desktop manager *dtwm* is also included unless otherwise stated. The majority of the differences between the CDE desktop and the Motif window manager *mwm* are concerned with consistency of visuals across platforms in any case, and do not overly affect the way in which clients communicate using the standardized ICCCM mechanisms.

The material in this chapter is advanced; you should typically not interfere with the predefined interactions between an application and the Motif window manager. When you do so, you risk interfering with the uniform look and feel that is at the heart of a graphical user interface such as Motif. However, the material in this chapter should provide you with an understanding of some important concepts that may allow you to make your applications more robust.This chapter also discusses the use of protocols and client messages for window manager communication. These techniques can be used for communication between instances of the same application or between suites of cooperating applications.

Interclient Communication

The X Window System is designed so that any user-interface style can be imposed on the display. The X libraries (Xlib and Xt) provide the mechanisms for applications to decide for themselves how to display information and how to react to user-generated actions. It is left up to graphical user interface specifications such as Motif to standardize most of these decisions. However, in order to preserve a baseline of inter-operability, there are certain standards that an application must conform to if it is to be considered a "good citizen" of the desktop. These standards are referred to as interclient communication conventions. While X makes no suggestions about the way an application should look or act, it does have a lot to say about how it interacts with other applications on the user's display.

One such convention is that all applications must negotiate the sizes and positions of their windows with the window manager, rather than with one another. The window manager is, in essence, the ultimate ruler of the desktop. While it is mostly benevolent, its primary function is to prevent anarchy on the display. Communication with the window manager has various forms. Applications can talk directly to the window manager, or the window manager may initiate a conversation with an application. When the user selects a item from the window menu or issues other window manager commands, he or she initiates communication between the window manager and the application. Much of the communication between the window manager and the application is carried on in terms of *properties* and *protocols.*

A property is an arbitrary-length piece of data associated with a window. It is stored on the server identified by a unique integer value called an `Atom`.* An application sets properties on its windows as a way of communicating with the window manager or other applications. Some properties are referred to as "window manager hints" because the window manager doesn't have to obey them. For example, an application can specify the preferred size of its top-level window, but the window manager might use this value only in the absence of any other instructions from the user.

A window manager protocol is an agreed-upon procedure for the exchange of messages between the window manager and an application. Protocols are implemented with `ClientMessage` events; the window manager sends an event to the application, and the application takes the appropriate action. For example, a protocol exchange occurs when the user selects *Close* from the window menu to close an application window.

There are low-level Xlib routines for setting and getting the value of window properties. However, the various shell widgets provided by Xt and Motif define resources that access

* Atoms are used to avoid the overhead of passing property names as arbitrary-length strings. See Volume 1, *Xlib Programming Manual*, and Volume 4, *X Toolkit Intrinsics Programming Manual* for a detailed discussion of properties and atoms.

most of the predefined properties of interest in window manager/application interaction. These resources are the preferred interface to window properties.

The WMShell widget defines many of the generic properties that are used for communication with the window manager. For example, you can use WMShell resources to specify an icon pixmap and resize increment values. The VendorShell widget class is defined by Xt as the widget class in which a vendor can define appearance and behavior resources specific to its own window manager. As such, this widget class is customized by every vendor of Xt-compatible toolkits. In the case of Motif, the VendorShell class provides resources that control the layout and operation of the Motif window manager decorations, and it supports the Motif window manager protocols.

You never instantiate WMShell or VendorShell widgets; they exist only as supporting classes for other shells, such as TopLevelShells, SessionShells, and DialogShells*. However, you frequently need to set WMShell and VendorShell resources on other types of shell widgets. Remember that the MenuShell widget is not a subclass of VendorShell and WMShell, so it does not have the same provisions for window manager interaction. You can use the `XtIsVendorShell()` macro defined in *<X11/Intrinsic.h>*, to determine if a widget is a subclass of VendorShell. Similarly, the `XtIsWMShell()` macro indicates whether or not a widget is a subclass of WMShell. Once you have a handle to a shell widget, you can specify both generic and Motif-resources for it.

Shell Resources

As discussed in Chapter 3, *Overview of the Motif Toolkit*, the WMShell widget class handles standard window manager/application communications as established by the *Inter-Client Communications Conventions Manual* (ICCCM). This document can be found in *Appendix L of Volume 0, X Protocol Reference Manual*, by the X Consortium for all interclient communication. Such conventions are necessary because the window manager and a client application are two separate programs. Applications and window managers need to follow these standards to maintain order in the X world.

To give you an idea of the kinds of properties in which the window manager is interested, Table 20-1 shows a partial list of properties that are handled automatically by shells.

Table 20-1 Some Window Manager Properties

Atom	Meaning
WM_NAME	The name of the window
WM_CLASS	The class name of the window
WM_NORMAL_HINTS	Information about the size of the window

* The ApplicationShell widget class is considered deprecated in X11R6, and is superseded by the SessionShell.

Table 20-1 Some Window Manager Properties (continued)

Atom	Meaning
WM_ICON_NAME	The name of the icon for the window
WM_HINTS	Information about the icon pixmap, icon position, and input model for the window

Xlib provides functions for modifying the values of these atoms on a window so that you can change the visual appearance, size, position, or functionality of the window.* However, the job of the WMShell is to hide this interface from the programmer by providing resources that accomplish the same tasks. The next few sections describe how most of the common resources can be used. While we do not cover all of the WMShell resources here, most of the ones we have omitted are intuitive, so they do not require a great deal of explanation. See the WMShell reference page in Volume 6B, *Motif Reference Manual*, for a complete list of resources.

Shell Positions

You can position a shell at a specific location on the screen using the XmNx and XmNy resources. In addition, you can set the XmNx and XmNy resources of the immediate child of a shell widget to position the shell. This feature exists because Motif dialogs are designed to make their shell widgets invisible to the programmer. It is typically easier to set these resources directly on the child of a shell, as you are more likely to have a handle to that widget. The following code fragment shows how you can position a MessageDialog in the center of the screen:

```
Widget         dialog, parent;
Dimension      width, height;
Screen         screen = XtScreen (parent);
Position       x, y;

dialog = XmCreateMessageDialog (parent, "dialog", NULL, 0);
/* get width and height of dialog */
XtVaGetValues (dialog, XmNwidth, &width, XmNheight, &height, NULL);
/* center the dialog on the screen */
x = (WidthOfScreen (screen) / 2) - (width / 2);
y = (HeightOfScreen (screen) / 2) - (height / 2);
XtVaSetValues (dialog, XmNx, x, XmNy, y, NULL);
```

You can position a dialog in this way because the Motif BulletinBoard widget passes positional information to its shell parent. See Chapter 5, *Introduction to Dialogs*, and Chapter 7, *Custom Dialogs*, for further discussion. In most cases, you shouldn't be setting the XmNx and XmNy resources for a dialog because it is the job of the window manager to position shells. The user can also have some say in how placement should be handled. For

* See Volume 0, *Xlib Programming Manual*, for complete details on the properties that can be set on windows; see Volume 1, *X Toolkit Intrinsics Programming Manual* for details on how to set or get these properties.

example, if the user has set the `interactivePlacement` resource for *mwm* to `True`, he gets to place the window himself when it first appears. If you set the position of the window, then you are interfering with the positioning method preferred by the user.

Shell Sizes

In some situations, an application may want to prevent one of its windows from growing or shrinking beyond certain geometrical limits. For example, an application might want to keep a dialog box from getting so small that some of its elements are clipped. A paint application might want to restrict its top-level window from growing larger than the size of its canvas. An application can also constrain the increments by which the user can interactively resize the window. For example, *xterm* only allows itself to be resized in character-size increments, where the character size is defined by the font being used.

The WMShell defines the following resources that can be used to constrain the size of a window:

```
XmNminWidth     XmNmaxWidth     XmNminHeight     XmNmaxHeight
XmNwidthInc     XmNheightInc    XmNbaseWidth     XmNbaseHeight
```

The `XmNminWidth`, `XmNmaxWidth`, `XmNminHeight`, and `XmNmaxHeight` resources specify the minimum and maximum width and height for the shell. The `XmNwidthInc` and `XmNheightInc` resources control the pixel incrementals by which the window changes when it is being resized by the user. When *mwm* provides visual feedback during a resize operation, it specifies the width and height in terms of these increments, rather than pixels. The `XmNbaseWidth` and `XmNbaseHeight` resources specify the base values that are used when calculating the preferred size of the shell.

Example 20-1 demonstrates incremental resizing. The application displays a shell widget that contains a PushButton. When you click on the button, it displays the sizeof the window in pixels, but when you resize the window, the *mwm* feedback window displays the size in terms of `XmNwidthInc` and `XmNheightInc`.*

Example 20-1 The resize_shell.c program

```
/* resize_shell.c -- demonstrate the max and min heights and widths.
** This program should be run to really see how mwm displays the
** size of the window as it is resized.
*/

#include <Xm/PushB.h>

main (int argc, char *argv[])
{
    Widget           toplevel, button;
    XtAppContext     app;
```

* `XtVaAppInitialize`() is considered deprecated in X11R6.

```
    void            getsize(Widget, XtPointer, XtPointer);

    XtSetLanguageProc (NULL, NULL, NULL);
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv, NULL,
                            sessionShellWidgetClass,
                            XmNminWidth, 75,
                            XmNminHeight, 25,
                            XmNmaxWidth, 150,
                            XmNmaxHeight, 100,
                            XmNbaseWidth, 5,
                            XmNbaseHeight, 5,
                            XmNwidthInc, 5,
                            XmNheightInc, 5,
                            NULL);

    /* PushButton's callback prints the dimensions of the shell. */
    button = XmCreatePushButton (toplevel, "Print Size", NULL, 0);
    XtManageChild (button);
    XtAddCallback (button, XmNactivateCallback, getsize,
                    (XtPointer) toplevel);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

void getsize (Widget widget, XtPointer client_data, XtPointer call_data)
{
    Widget       shell = (Widget) client_data;
    Dimension    width, height;

    XtVaGetValues (shell, XmNwidth, &width, XmNheight, &height, NULL);
    printf ("Width = %d, Height = %d\n", width, height);
}
```

In our example, we arbitrarily specify the minimum and maximum extents of the shell. The width and height increments are each set to five, so the user can only resize the window in five-pixel increments. As the window is resized, the feedback window displays the size according to these incremental units, rather than using pixel values. If you run *resize_shell*, you can press the PushButton to print the size of the shell in pixels and compare that size with the size reported by the window manager. If you are going to specify the various size resources for a shell, it only makes sense to hard-code the values as we have done here. If you specify the resources in an app-defaults file, the user can override the settings, which defeats the whole point of setting them.

The problem with specifying minimum and maximum extents is that most real applications contain many components whose sizes cannot be computed easily, making it difficult to determine exactly how large or small the window should be. If the fonts and strings for PushButtons, Labels, and ToggleButtons can be set in a resource file, the equation becomes far too difficult to calculate before the window is actually created and displayed. Incremental width and height values are even more difficult to estimate because there are margins, border widths, and other resources to consider.

However, all is not lost. If you need to constrain the size of an application, you should consider whether the application's default initial size can be considered either its maximum or minimum size. If so, you can allow the window to come up using default size and trap for `ConfigureNotify` events on the shell widget. You can then use the default width and height reported in that event as your minimum or maximum size, as demonstrated in Example 20-2.*

Example 20-2 The set_minimum.c program

```
/* set_minimum.c -- demonstrate how to set the minimum size of a
** window to its initial size. This method is useful if your program
** is initially displayed at its minimum size, but it would be too
** difficult to try to calculate ahead of time what the initial size
** would be.
*/

#include <Xm/PushB.h>

void getsize(Widget, XtPointer, XtPointer);
void configure(Widget, XtPointer, XEvent *, Boolean *);

main (int argc, char *argv[])
{
    Widget           toplevel, button;
    XtAppContext     app;

    XtSetLanguageProc (NULL, NULL, NULL);
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv, NULL,
                            sessionShellWidgetClass,
                            XmNmaxWidth, 150,
                            XmNmaxHeight, 100,
                            XmNbaseWidth, 5,
                            XmNbaseHeight, 5,
                            XmNwidthInc, 5,
                            XmNheightInc, 5,
                            NULL);

    /* Add an event handler to trap the first configure event */
    XtAddEventHandler (toplevel, StructureNotifyMask, False, configure, NULL);

    /* PushButton's callback prints the dimensions of the shell. */
    button = XmCreatePushButton (toplevel, "Print Size", NULL, 0);
    XtManageChild (button);
    XtAddCallback (button, XmNactivateCallback, getsize,
                                              (XtPointer) toplevel);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

void getsize (Widget widget, XtPointer client_data, XtPointer call_data)
```

* `XtVaAppInitialize`() is considered deprecated in X11R6.

```
{
    Widget       shell = (Widget) client_data;
    Dimension    width, height;

    XtVaGetValues (shell, XmNwidth, &width, XmNheight, &height, NULL);

    printf ("Width = %d, Height = %d\n", width, height);
}

void configure (Widget shell, XtPointer client_data, XEvent *event,
                Boolean *unused)
{
    XConfigureEvent *cevent = (XConfigureEvent *) event;

    if (cevent->type != ConfigureNotify)
        return;

    printf ("Width = %d, Height = %d\n", cevent->width, cevent->height);

    XtVaSetValues (shell, XmNminWidth, cevent->width,
    XmNminHeight, cevent->height, NULL);

    XtRemoveEventHandler (shell, StructureNotifyMask, False, configure, NULL);
}
```

We use `XtAddEventHandler()` to add an event handler to the top-level shell for events that satisfy the `StructureNotifyMask`, which includes `ConfigureNotify` events indicating the window's dimensions. The `configure()` function is called when the window is initially sized, so we can use the `width` and `height` fields of the `XConfigureEvent` structure as values for the `XmNminWidth` and `XmNminHeight` resources for the shell. To prevent the event handler from being called each time the window is resized, the event handler removes itself using `XtRemoveEventHandler()`.

One problem with this technique occurs when the user has the `interactivePlacement` resource for *mwm* set to `True`. This specification allows the user to set the initial size and position of an application. However, once the user sets the initial size, she will never be able to make the window any smaller. Although interactive placement adheres to the constraints we have set, it cannot enforce a minimum size because we have not set one. Unfortunately, there is no way to allow interactive placement without allowing the user to resize the window.

The Shell widget class defines the `XmNallowShellResize` resource that is inherited by all of its subclasses. This resource specifies whether or not the shell allows itself to be resized when its widget children are resized, but it does not affect whether the user can resize the window. For example, if the number of items in a List widget grows, the widget tries to increase its own size, which causes a rippling effect that eventually reaches the top-level window. If `XmNallowShellResize` is `True` for this shell, it grows, subject to the window manager's approval, of course. However, if the resource is `False`, the shell does not even consult the window manager because it knows that it doesn't want to resize. This

resource only prevents the shell from resizing after it has been realized, so it does not interfere with the initial sizing of the shell.

The Shell's Icon

Shells can be in one of three states: normal, iconic, or withdrawn. When a shell is in its normal state, the user can interact with the user-interface elements in the expected way. If a shell is withdrawn, it is still active, but the user cannot interact with it directly. When a shell is iconic, its window is not mapped to the screen, but instead it displays a smaller image, or icon, that represents the entire window. The application is still running in this state, but the program does not expect any user interaction. The icon window usually displays a visual image that suggests some connection to the window from which it came. Some window managers, like *mwm*, also allow a label to be attached to the icon's window.

The XmNiconPixmap resource specifies the pixmap that is used when an application is in an iconic state. Example 20-3 shows a simple application that sets its icon pixmap.*

Example 20-3 The icon_pixmap.c program

```
#include <Xm/Xm.h>
#include <X11/bitmaps/mailfull>

main (int argc, char *argv[])
{
    Widget         toplevel;
    XtAppContext   app;
    Pixmap         bitmap;

    XtSetLanguageProc (NULL, NULL, NULL);

    /* size is irrelevant -- toplevel is iconified */
    /* it just can't be 0, or Xt complains */
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv, NULL,
                                    sessionShellWidgetClass,
                                    XmNwidth, 100,
                                    XmNheight, 100,
                                    XmNiconic, True,
                                    NULL);
    bitmap = XCreatePixmapFromBitmapData (XtDisplay (toplevel)
                            RootWindowOfScreen (XtScreen (toplevel)),
                            (char *) mailfull_bits,
                            mailfull_width
                            mailfull_height,
                            1, 0, 1);
    XtVaSetValues (toplevel, XmNiconPixmap, bitmap, NULL);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}
```

* XtVaAppInitialize() is considered deprecated in X11R6.

The program creates an ApplicationShell and sets the XmNiconic resource to True to cause the application to appear iconified. The bitmap variable is initialized to contain the bitmap described by the file */usr/include/X11/bitmaps/mailfull*, and the XmNiconPixmap resource for the shell is set to the bitmap.

When we set the XmNiconPixmap and XmNiconic resources, we are actually sending hints to the window manager that we would like the icon window to display the given pixmap and that we would like to be in the iconic state. These requests are called hints because the window manager does not have to comply with the requests. However, if the icon pixmap or iconic state is ignored, it is most likely a bug in the window manager, or an incomplete implementation of one, which is often the case for older versions of many window managers, including *mwm* (Version 1.0).

One work around for a window manager that ignores the icon pixmap is to set the XmNiconWindow resource. This resource sets the entire icon window, rather than just its image. In environments where the user may not be running the most up-to-date window manager, it may be best to create the icon window directly and then paint an image in that window. Example 20-4 contains a routine that demonstrates this technique.This routine creates a shell's icon window and can be called repeatedly to dynamically update its image.

Example 20-4 The SetIconWindow() routine.

```
void SetIconWindow (Widget shell, Pixmap image)
{
    Window          window, root;
    unsigned int    width, height, border_width, depth;
    int             x, y;
    Display         *dpy = XtDisplay (shell);

    /* Get the current icon window associated with the shell */
    XtVaGetValues (shell, XmNiconWindow, &window, NULL);

    if (!window) {
        /* If there is no window associated with the shell, create one.
        ** Make it at least as big as the pixmap we're
        ** going to use. The icon window only needs to be a simple window.
        */
        if (!XGetGeometry (dpy, image, &root, &x, &y, &width, &height,
                           &border_width, &depth) ||
                           !(window = XCreateSimpleWindow (dpy, root, 0, 0,
                           width, height, (unsigned)0, CopyFromParent,
                           CopyFromParent)))
        {
            XtVaSetValues (shell, XmNiconPixmap, image, NULL);
            return;
        }

        /* Now that the window is created, set it... */
        XtVaSetValues (shell, XmNiconWindow, window, NULL);
    }
```

```
        /* Set the window's background pixmap to be the image. */
        XSetWindowBackgroundPixmap (dpy, window, image);
        /* cause a redisplay of this window, if exposed */
        XClearWindow (dpy, window);
    }
```

SetIconWindow() takes two parameters: a shell and an image. If the icon window for shell has not yet been set, we create a window using XCreateSimpleWindow(). The size of the window is set to the size of the image, which is retrieved with XGetGeometry(). This function is used to get the size of the image, but it can be used on windows as well. In the unlikely event that one of these routines fails, we fall back to using XmNiconPixmap to specify the image and hope the window manager understands it. Otherwise, we set the XmNiconWindow resource to the window we just created.

We use the image pixmap to set the window's background pixmap, which saves us the hassle of rendering it using XCopyArea() or XCopyPlane(). If the shell widget already has an icon window, XSetWindowBackgroundPixmap() is still called so that the specified image is displayed. The final call to XClearWindow() causes the icon to be repainted. This call isn't necessary if the window has just been created, but it is necessary if the window is merely updated with a new image.

The XmNiconX and XmNiconY resources can be used to set the position of the icon window on the screen. However, you probably shouldn't set these resources arbitrarily without a really good reason. Most window managers deal with positioning icon windows, or leave the positioning for the user to specify, so it is best not to interfere.

The XmNtitle and XmNiconName resources specify the titles used for the application window and the icon window, respectively. These resources are set to regular character strings, not compound strings. These values are typically both set to the name of the program,argv[0], by default. The values also affect the WM_NAME property for the top-level window, which is important for session managers and other applications that monitor all top-level windows on a desktop. These programs look for the WM_NAME property to provide menus or buttons that allow the user to control the desktop in a GUI-like fashion, rather than through tty-like shells such as *xterm* and *csh*. It is best to let the user set the XmNtitle and XmNiconName resources, especially since Xt provides command-line options such as -name that can be used to set the title of an application.

VendorShell Resources

The VendorShell widget class is subclassed from WMShell, so all of the shell widget classes subclassed from VendorShell can use the resources described in the previous section. All of the Motif shells except for MenuShell are subclassed from VendorShell. The VendorShell is designed to be implemented by individual vendors so that they can define resources specific to their own window manager. For example, *mwm* has some window

manager features that are not found in other window managers. You need to be familiar with the Motif window manager in order to understand the discussion that follows.

Window Manager Decorations

The frame around an application's main window belongs to the window manager; the controls and window menu in it are not part of the application. The *mwm* window manager decorations for an application window are shown in Figure 20-1.

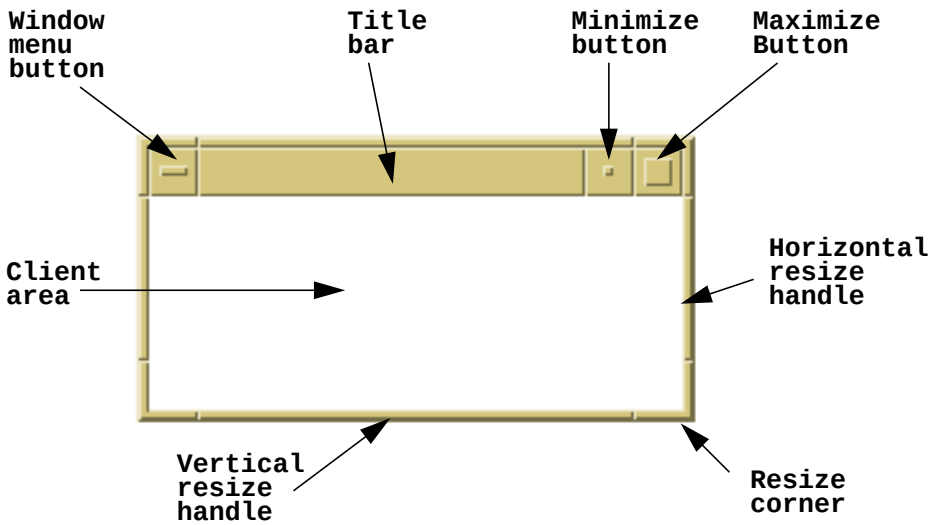


Figure 20-1 Motif window manager decorations

The user can set *mwm* resources to control which of these items are available for particular windows on the desktop. Also, *mwm* automatically controls which elements are visible for certain windows, in order to maintain compatibility with the *Motif Style Guide*. As such, we discourage you from modifying the decorations that are available on specific windows. Nevertheless, the VendorShell does provide the `XmNmwmDecorations` resource for use in exceptional cases. The resource can be set to an integer value that is made up of any of the following values:

`MWM_DECOR_BORDER`
: This value enables the window manager borders for the frame. These borders are decorative only; they are not resize handles. Except for non-rectangular windows or programs like a clock, all Motif-style applications should have decorative borders.

`MWM_DECOR_RESIZEH`
: This value enables the resize handles for the frame. If the resize handles are displayed, the decorative borders are forced to be displayed.

MWM_DECOR_TITLE
: This value enables the title bar for the window.

MWM_DECOR_MENU
: This value enables the window menu button on the title bar. If this item is on, the title bar is forced to be displayed.

MWM_DECOR_MAXIMIZE
: This value makes the maximize button visible. When this button is selected, the window is expanded to the largest size possible. The size of the window is constrained by the values for XmNmaxWidth and XmNmaxHeight. If these resources are not set, the window is expanded to the size of the screen.

MWM_DECOR_MINIMIZE
: This value makes the minimize button visible. This button does not shrink the window, but rather iconifies it. This item is turned off by default for TransientShell widgets (dialogs), since they cannot be iconified separately from their parent shells.

MWM_DECOR_ALL
: This value can be used to enable all of the window manager decorations.

All of these values are defined in <*Xm/MwmUtil.h*>, which must be included before any of them may be used. The values are bit masks, so they are meant to be ORed together. For example, if you have a customized dialog that you do not want to have resize handles, you can turn them off as shown in the following code fragment:

```
Widget   dialog_shell;
int      decor;

XtVaGetValues (dialog_shell, XmNmwmDecorations, &decor, NULL);
decor &= ~MWM_DECOR_RESIZEH;
XtVaSetValues (dialog_shell, XmNmwmDecorations, decor, NULL);
```

While the programmatic interface is available to make changes in the form described above, you really don't have to resort to this level of complexity. If you want to do something that is allowed by the *Motif Style Guide*, chances are that the Motif toolkit provides a more convenient way of doing it. For example, you can turn off the resize handles for a Motif dialog by setting the XmNnoResize resource to True, as shown in the following code:

```
Widget   dialog;
Arg      args[5];
int      n = 0;

XtSetArg (args[n], XmNnoResize, True); n++;
dialog = XmCreateFileSelectionDialog (parent, "dialog", args, n);
```

If Motif doesn't provide a convenience routine or a resource for doing what you want, chances are good that you shouldn't be doing it. On the other hand, you don't have to use the convenience method; if it seems appropriate, you can use the methods described here.

Window Menu Functions

The contents of the window menu can be modified using the XmNmwmFunctions resource defined by the VendorShell. This resource acts like XmNmwmDecorations, in that the value is an integer that may be set to one or more of the following values:

MWM_FUNC_RESIZE
: This value enables the *Size* item in the window menu. If this value isn't set, the resize handles for the window manager frame are disabled.

MWM_FUNC_MOVE
: This value enables the *Move* menu item. Disabling this item does not affect the window manager frame decorations for the window.

MWM_FUNC_MINIMIZE
: This value enables the *Minimize* menu item. Disabling this item causes the minimize button to be disabled as well.

MWM_FUNC_MAXIMIZE
: This value enables the *Maximize* menu item. Disabling this item causes the corresponding window frame decoration to be disabled.

MWM_FUNC_CLOSE
: This value enables the *Close* menu item. Disabling this item does not affect the window manager decorations for the window.

MWM_FUNC_ALL
: This value causes all of the standard items in the menu to be displayed and all the default functionality of the window manager to work.

It is important to remember that the user can specify these window menu functions, as well as new functions, in an *.mwmrc* file (See Motif Volume 3, *X Window System User's Guide, Motif Edition*). While your settings override any user specifications, you should only modify the window menu functions if it is absolutely necessary. A common misuse of this functionality is to disable the *Close* button. We strongly discourage disabling this button, as users expect it to be in the window menu. Rather than disable the button, you should link its functionality to another control in your application that has the same meaning. For example, if you are using a standard Motif dialog that provides *OK* and *Cancel* buttons, you can link the *Close* menu item to the *Cancel* button. We explain how to connect the functionality of these components in the next section.

Handling Window Manager Messages

A protocol is a set of rules that governs communication and data transfer. When the window manager sends a message to an application that follows a predefined protocol, the client application should respond accordingly. The ICCCM defines a number of protocols

for window managers and applications to follow. One such protocol involves the *Close* item in the window menu. When the user selects this item, the window manager sends the application a protocol message, and the application must comply. The message is delivered through the normal event-handling mechanisms provided by Xlib. The event that corresponds to this message is called a `ClientMessage` event. The message itself is an `Atom`, which is merely a unique integer that is used as an identifier. (The actual value is unimportant, since you only need to reference the value through the preprocessor macro, `WM_PROTOCOLS`.) The protocol itself takes the form of other atoms, depending on the nature of the message. Table 20-2 lists the atoms that are used as values for `WM_PROTOCOLS` client messages. Although this table is currently complete, it is expected to grow in future editions of the ICCCM.

Table 20-2 Protocol Atoms Defined by the ICCCM

Atom	Meaning
WM_TAKE_FOCUS	The window is getting the input focus.
WM_DELETE_WINDOW	The window is about to be deleted.
WM_SAVE_YOURSELF	The application should save its internal state.

Example 20-5 demonstrates how to use the `WM_DELETE_YOURSELF` protocol to link the *Close* item on the window menu with the *Cancel* button in a dialog.*

Example 20-5 The wm_delete.c program.

```
/* wm_delete.c -- demonstrate how to bind the Close button in the
** window manager's system menu to the "cancel" button in a dialog.
*/

#include <Xm/MessageB.h>
#include <Xm/PushB.h>
#include <Xm/Protocols.h>

main (int argc, char *argv[])
{
    Widget         toplevel, button;
    XtAppContext   app;
    void           activate(Widget, XtPointer, XtPointer);

    XtSetLanguageProc (NULL, NULL, NULL);
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv, NULL,
                                    sessionShellWidgetClass, NULL);
    button = XmCreatePushButton (toplevel, "Push Me", NULL, 0);
    XtManageChild (button);
    XtAddCallback (button, XmNactivateCallback, activate, NULL);
    XtRealizeWidget (toplevel);
```

* `XtVaAppInitialize`() is considered deprecated in X11R6. `XmMessageBoxGetChild`() is deprecated from Motif 2.0. `XmInternAtom`() is marked for deprecation from Motif 2.0.

```
    XtAppMainLoop (app);
}

/* Create and popup an ErrorDialog indicating that the user may have
** done something wrong. The dialog contains an OK and Cancel button,
** but he can still choose the Close button in the titlebar.
*/
void activate (Widget w, XtPointer client_data, XtPointer call_data)
{
    Widget     dialog, shell;
    void       response(Widget, XtPointer, XtPointer);
    XmString   t = XmStringCreateLocalized ("Warning: Delete All Files?");
    Atom       WM_DELETE_WINDOW;
    Arg        args[5];
    int        n;

    /* Make sure the VendorShell associated with the dialog does not
    ** react to the user's selection of the Close system menu item.
    */
    n = 0;
    XtSetArg (args[n], XmNmessageString, t); n++;
    XtSetArg (args[n], XmNdeleteResponse, XmDO_NOTHING); n++;
    dialog = XmCreateWarningDialog (w, "notice", args, n);
    XmStringFree (t);
    /* add callback routines for ok and cancel -- desensitize help */
    XtAddCallback (dialog, XmNokCallback, response, NULL);
    XtAddCallback (dialog, XmNcancelCallback, response, NULL);
    XtSetSensitive (XtNameToWidget (dialog, “Help”), False);
    XtManageChild (dialog);
    /* Add a callback for the WM_DELETE_WINDOW protocol */
    shell = XtParent (dialog);
    WM_DELETE_WINDOW = XInternAtom (XtDisplay (w), "WM_DELETE_WINDOW", False);
    XmAddWMProtocolCallback (shell, WM_DELETE_WINDOW, response,
                             (XtPointer) dialog);
}

/* callback for the OK and Cancel buttons in the dialog -- may also be
** called from the WM_DELETE_WINDOW protocol message sent by the wm.
*/
void response (Widget widget, XtPointer client_data, XtPointer call_data)
{
    XmAnyCallbackStruct *cbs = (XmAnyCallbackStruct *) call_data;
    Widget              dialog;

    if (cbs->reason == XmCR_OK)
        puts ("Yes");
    else
        puts ("No");
    if (cbs->reason == XmCR_PROTOCOLS)
        /* we passed the dialog as client data for the protocol callback */
        dialog = (Widget) client_data;
    else
        dialog = widget;
    XtDestroyWidget (dialog);
```

```
}
```

When you run the application and click on the button, a dialog is displayed. All the application does is print "Yes" or "No" to standard output based on whether the *OK* or *Cancel* button is pressed. However, if you select *Close* from the window menu for the dialog, the dialog disappears, and the "No" message is printed.

When the user selects the *Close* item on the window menu, the application is sent a `ClientMessage` event by the window manager indicating that the window is about to be deleted. The value associated with the `WM_PROTOCOLS` message is `WM_DELETE_WINDOW`. The application is now responsible for complying with the protocol in some way.

At the highest level of abstraction, the VendorShell resource `XmNdeleteResponse` can be used to control what the application does in response to the user's selection of the *Close* button. The default behavior for a dialog is that the window is dismissed; the value `XmUNMAP` is used, and the window is unmapped from the screen. By setting `XmNdeleteReponse` to `XmDESTROY`, the window is destroyed; this value is the default for ApplicationShells. However, if the resource is set to `XmDO_NOTHING`, the application declares that it is going to handle the action itself.

In Example 20-5, we use this value to handle the `WM_DELETE_WINDOW` protocol ourselves by setting up a callback routine that is called whenever the protocol is sent. But before we can set up the callback, we have to get the atom associated with the `WM_DELETE_WINDOW` protocol. We retrieve the atom using `XInternAtom()`, which takes the following form:

```
Atom XInternAtom (Display *display, char *atom_name, Boolean dont_create)
```

If the atom name described by the string *`atom_name`* exists, then the `Atom` is returned. If it does not exist and if *`dont_create`* is `True`, the function returns `None`. Otherwise, the routine creates and returns the atom.

Once we have the protocol atom, we can add a callback routine to respond to the client message event generated by that protocol. The function `XmAddWMProtocolCallback()` is used to install a callback routine invoked whenever the window manager sends a `WM_PROTOCOLS` client message to the application. If the protocol sent in the client message matches the protocol passed to `XmAddWMProtocolCallback()`, the associated function is called. In Example 20-5, we use the `response()` routine as the callback for the dialog buttons and the protocol. As a result, the *Close* item invokes the same callback as the *OK* and *Cancel* buttons.

The form of this callback routine is the same as any other Motif callback. The final parameter is a Motif-defined callback structure of some kind, where the `reason` field specifies why the callback was called. This field is provided because the same callback function may be invoked by more than one widget. In our example, the `response()` function's callback structure may have one of three different values for `reason`: `XmCR_OK` for the *OK* button, `XmCR_CANCEL` for the *Cancel* button, or `XmCR_PROTOCOLS` for the

Close button in the window menu. When the callback is invoked for the protocol message, the `event` field of the callback structure is an `XClientMessageEvent`.

The `widget` parameter passed to `response()` also varies depending on whether the routine is called from the dialog or from the *Close* button. When either *OK* or *Cancel* is pressed, the widget is the dialog itself. But the protocol callback routines are really processed by special *protocol widgets* that are attached to VendorShells.* When the protocol callback is invoked, the `widget` field is one of the special widgets, but this widget has no intrinsic meaning, so it can be ignored. We know that the activation of the `WM_DELETE_WINDOW` protocol causes a protocol widget to be passed as the `widget` parameter. Therefore, we pass a handle to the dialog widget as the client data to `XmAddWMProtocolCallback()` so that we have access to the dialog.

The purpose, of course, is to destroy the window, but our function could just as easily veto the operation and render the *Close* button inoperable. However, this technique is really not appropriate, as users expect to be able to use the *Close* button to remove a window. If the *Close* button is not going to unmap the window for some good reason, like an error, you should report the error in another dialog. If you are going to modify the default behavior of standard user-interface controls, you should keep the user informed about what you are doing.

Adding New Protocols

In general, you can attach a callback routine to any of the published protocols using the mechanisms we just described. You may also assign new protocols to send yourself special messages that are pertinent only to your application, as protocol messages can be passed from application to application, not just between the window manager and other clients. Handling arbitrary protocols is basically a matter of following these simple steps:

1. Create an atom or retrieve one from the X server using `XInternAtom()`.
2. Register the atom on the shell with `XmAddWMProtocols()`, so the event-handling mechanism can recognize it if it should arrive.
3. Install a callback routine that is invoked when the protocol is sent to the application using `XmAddWMProtocolCallback()`.

For the case of `WM_DELETE_WINDOW`, the second step has already been taken care of by the VendorShell, since it is an established and standardized ICCCM protocol. The VendorShell has already registered interest in the protocol so it can react to it in the method described by its `XmNdeleteResponse` resource. However, other protocols (customized or not) may not be registered. Since it doesn't hurt to register a protocol with a window more than once,

* A shell can actually have any number of widget children, as long as only one of them is managed at a time. In the case of the Motif VendorShell, these other widgets are not managed but are used to process and manage protocols that are exchanged between the window manager and the application.

it's always a good practice to register the protocol using XmAddWMProtocols(), which takes the following form:

```
void XmAddWMProtocols (Widget shell, Atom *protocols, int num_protocols)
```

This function takes a list of protocols, so you can use it to add as many protocols as you like at one time.

Session Management

A *session manager* is an application that acts something like a window manager. However, rather than controlling only the windows on a screen, it monitors the actual applications running on that screen. Frequently, session managers allow the user to start, terminate, or even restart any program automatically, through a variety of interface controls. Session managers may even cause a program to "sleep" by terminating all its keyboard and mouse input, so as far as the program is concerned, the user is just not interacting with it.

This section discusses one aspect of session manager behavior and how it might be implemented. This behavior concerns the ability of an application running under the session manager to restart itself at the point where it left off in a previous session. The implementation focuses initially on the functionality inherent in the protocols defined by the ICCCM, and proceeds to a discussion of the new features provided by the X11R6 SessionShell.

Session Management in X11R5

Under the scheme drafted prior to X11R6, if the session manager decides that it should terminate (which might result in the entire X connection terminating), it may send a request to all its applications to save their internal state so they can be restarted later. In this case, the session manager sends a WM_SAVE_YOURSELF protocol message. According to the ICCCM, client applications that can save their current state and restart from that state should register the atom WM_SAVE_YOURSELF on the WM_PROTOCOLS property on one of their top-level windows.

The ICCCM further stated that after sending the WM_SAVE_YOURSELF message to the application, the session manager should wait until the program updates its WM_COMMAND property on the same window that received the protocol message. The application was not permitted to interact with the user in any way at this time. You were not supposed to prompt for filenames or ask if the user wants to save state. The callback routine saved its current state somehow, possibly in a predefined file that could be made known to the user through documentation, rather than a run-time message. It then updated the WM_COMMAND property to reflect the parameters that started the program, as well as any additional parameters that might be required to restart it.

For example, say your application is called *wm_save* and you want to be able to restart it from a previously-saved file. In this case, your application might parse the following command-line option:

```
% wm_save -restart filename
```

Example 20-6 contains a code fragment that demonstrates how you would implement this functionality which is compatible with the X11R5 model.*

Example 20-6 The wm_save.c program

```
/* wm_save.c -- demonstrate how to save the state of an application
** from a WM_SAVE_YOURSELF session manager protocol. This is not a
** real program -- just a template.
*/

#include <Xm/Xm.h>
#include <Xm/Protocols.h>
#include <stdio.h>

/* save the original argc and argv for possible WM_SAVE_YOURSELF messages */
int save_argc;
char **save_argv;

main (int argc, char *argv[])
{
    Widget          toplevel;
    XtAppContext    app;
    Atom            WM_SAVE_YOURSELF;
    void            save_state();
    char            *restart_file;
    int             i;

    /* save argc and argv values */
    save_argv = (char **) XtMalloc (argc * sizeof (char *));
    for (i = save_argc = 0; i < argc; i++) {
        /* we don't need to save old -restart options */
        if (!strcmp (argv[i], "-restart"))
            i++; /* next arg is filename */
        else {
            char *copy = XtMalloc (strlen (argv[i]) + 1);
            save_argv[save_argc++] = strcpy (copy, argv[i]);
        }
    }

    XtSetLanguageProc (NULL, NULL, NULL);
    /* initialize toolkit: argv has its Xt-specific args stripped */
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv, NULL,
                sessionShellWidgetClass, XmNwidth, 100, XmNheight, 100, NULL);
```

* `XtVaAppInitialize()` is considered deprecated in X11R6. `XInternAtom()` is marked for deprecation from Motif 2.0.

```
    /* get the WM_SAVE_YOURSELF protocol atom and register it with the
    ** toplevel window's WM_PROTOCOLS property. Also add a callback.
    */
    WM_SAVE_YOURSELF = XInternAtom (XtDisplay (toplevel),
                                    "WM_SAVE_YOURSELF", False);
    XmAddWMProtocols (toplevel, &WM_SAVE_YOURSELF, 1);
    XmAddWMProtocolCallback (toplevel, WM_SAVE_YOURSELF, save_state,
                             (XtPointer) toplevel);

    /* create widgets... */
    ...
    /* now check to see if we are restarting from a previously run state */
    for (i = 0; i < argc; i++) {
        if (!strcmp (argv[i], "-restart")) {
        /* restarting from a previously saved state */
            restart_file = argv[++i];
        }
        /* possibly process other args here, too */
    }

    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

/* called if WM_SAVE_YOURSELF client message was sent...
*/

void save_state (Widget widget, XtPointer client_data, XtPointer call_data)
{
    Widget         toplevel = (Widget) client_data;
    /* hypothetical function */
    extern char    *SaveStateAndReturnFileName();
    char           *filename = SaveStateAndReturnFileName ();

    puts("save_state()”);
    save_argv = (char **) XtRealloc ((char *) save_argv,
                   (save_argc+2) * sizeof (char *));
    save_argv[save_argc++] = "-restart";
    save_argv[save_argc++] = filename;
    /* notice the order of XSetCommand() args! */
    XSetCommand (XtDisplay (toplevel), XtWindow (toplevel),
                             save_argv, save_argc);
}
```

This program registers the WM_SAVE_YOURSELF protocol using XmAddWMProtocols() before it specifies the callback routine. If the session manager sends a WM_SAVE_YOURSELF message to this program then the save_state() function is called, which causes the program to save its internal state using the function SaveStateAndReturnFileName(). This is a hypothetical function that you would write yourself to save the state of the program and return the filename that contains the state information. The callback routine also adds the -restart flag and the new filename to the saved argv from the beginning of the program. The function XSetCommand() is used to

set the WM_COMMAND property on the window associated with the top-level shell, which fulfills the program's obligation to the session manager.

For more information about session managers and the save-yourself communication protocol, see Volume 0, *X Protocol Reference Manual*. For more details on XSetCommand() and other Xlib-based functions that set and get window manager properties on top-level windows, see Volume 1, *Xlib Programming Manual*, and Volume 2, *Xlib Reference Manual*.

Session Management in X11R6

X11R6 introduces the SessionShell widget class, which is specifically designed to encapsulate the interaction between an application and the session manager. The SessionShell is a subclass of the ApplicationShell; the ApplicationShell is now considered obsolete.

Using the SessionShell, handling the interaction between the session manager and the application no longer requires direct programming using the lower level window manager protocols. It is simply a matter of setting some new resources, and providing new callbacks where appropriate.

The SessionShell widget class is fully described in the *Programmer's Supplement for Release 6 of the X Window System*. Only the basics will be described here, in order to give sufficient information to describe the X11R6 equivalent of the techniques described in the previous X11R5 section. You are referred to the *Supplement* for further details.

Connecting to the Session Manager

The connection between the application and the session manager is established automatically when you create a SessionShell. The internals of the communicating process, and the messages passed backwards and forwards between the application and the session manager in performing the initial handshake are beyond the scope of this chapter. For the purposes of this chapter, it is simply sufficient to know that calling XtVaOpenApplication() or XtOpenApplication(), passing the sessionShellWidgetClass as a parameter, will establish the connection for us as a side effect of creating our first toplevel shell.

SessionShell Resources

The most important information which is passed between the session manager and the application is the name of the command, and arguments, which are to be used by the session manager to restart the application. This is specified using the XtNrestartCommand resource. The resource is represented internally by an array of strings, and is initialized by default from the argv, and argc parameters of the application which are passed to the XtOpenApplication() or XtVaOpenApplication() call when creating the

SessionShell. For many applications, this default behavior may be considered sufficient to recover the current state of the application. However, for many applications, the internal state which changes due to various options or actions taken by the user may well result in the need to modify the notional parameters passed to the application if that internal state is to be recovered. Consider an editor, which edits files either passed to it on the command line, and which dynamically opens files as a result of menu or other actions. We may need to update the XtNrestartCommand resource as each new file is opened in the application.

Recovering application state often involves considerably more than simply constructing an array of command line arguments. We may need to take into account the current directory in which the application was running at the time, or consider the state of environment variables which affect application behavior, or even explicitly supply the path to the application program. The SessionShell also supports these aspects of application recovery through resources.

The application environment can be explicitly set using the XtNenvironment resource; this is specified through a NULL-terminated array of strings. The default value is NULL, and is not initialized in any way from the current environment settings: you must explicitly program into the resource any environment which the application requires.

The current working directory can be specified through the XtNcurrentDirectory resource. Again, this resource has a default value of NULL, and must be explicitly set if required.

An explicit path to the application command can be specified through the XtNprogramPath resource. Unlike XtNrestartCommand, this resource is not initialized from the parameters passed to XtOpenApplication(), and the default value is NULL.

As well as specifying the restart behavior of our application, we can also inform the session manager of any commands we may care to execute in order to tidy up the current application before it exits. The XtNshutdownCommand resource specifies a command and arguments to be called by the session manager after our application terminates.

A unique handle on the interaction between the session manager and the application is available through the XtNsessionID resource. The exact syntax of the resource will not be covered here - you are referred to the *Supplement* for more details of this. As far as we are concerned in the examples which follow, it will be used simply to inform the session manager that we wish to restart our application in the same logical session in which it is currently running. We do this simply by copying the value from the SessionShell (where it was set up by the session manager) into part of the XtNrestartCommand array.

There is one other interesting resource which may be of use. This is the XtNcloneCommand, which can be used to inform the session manager how the application should be started in the general case. By default, if no XtNcloneCommand is specified, the session manager will clone a new application using the XtNrestartCommand value.

Think of the difference between restart and clone as this: a restart command informs the session manager how to recover as near as possible the current application state, but the clone command just starts the application in the normal initial state.

Example 20-7 shows a specimen routine which resets the application restart parameters. It can be considered as a logical equivalent of the save_state() routine from Example 20-6. We ignore for the moment the problem of how this routine gets to be called in the new scheme of things: this is covered in the following section on SessionShell callbacks.

Example 20-7 The set_session_restart() routine.

```
void set_session_restart (Widget w, XtPointer client_data, XtPointer call_data)
{
    Widget         toplevel = (Widget) client_data;
    /* hypothetical function */
    extern char    *SaveStateAndReturnFileName();
    char           *filename = SaveStateAndReturnFileName ();

    puts("set_session_restart()");
    save_argv = (char **) XtRealloc ((char *) save_argv,
                        (save_argc+3) * sizeof (char *));
    save_argv[save_argc++] = "-restart";
    save_argv[save_argc++] = filename;
    save_argv[save_argc] = (char *) 0; /* NULL terminated */

    XtVaSetValues (toplevel, XtNrestartCommand, save_argv, NULL);
}
```

SessionShell Callbacks

In the X11R5 model, we have to program the interchange between the session manager and the application using protocols. In X11R6, we use SessionShell callbacks. Furthermore, unlike the X11R5 model, we are allowed to interact with the user for whatever confirmation or information we require. User interaction is however strictly controlled in the sense that it should occur only at specific points in the interactions between the session manager and the application.

There are six session management callbacks which can be used to program the various stages of the interaction between the session manager and the application. Not all need to be programmed for the interactions to work - it all depends on the degree of sophistication and error recovery required by the application to hand.

The most important callback is the XtNsaveCallback. This is used to perform actual application state save. It should also initialise the XtNrestartCommand resource if the application state has changed since the last time the session manager issued a request. The code in Example 20-7 is entirely typical: it simply resets the XtNrestartCommand value to reflect the current save file name. Note that the XtNsaveCallback is not supposed to interact with the user. We would ensure that the XtNsaveCallback is active simply by registering the routine using normal Xt means:

```
extern Widget sessionShell; /* The application top level */

XtAddCallback (sessionShell, XtNsaveCallback, set_session_restart, NULL);
```

Once an application has saved its state, it may or may not require notification from the session manager that the message has been received and understood - that is, the session manager has managed to process all changes to the `XtNrestartCommand` resources for all the participating applications in the current session. The `XtNsaveCompleteCallback` can be used if this part of the interaction is important. For a typical application, it is not.

If the session manager or the application decides to terminate the save state request, any handling of clean-up operations required should be programmed using an `XtNcancelCallback`. Again, a typical application would not be over-concerned: it is unlikely that you would want to unwind any save operations.

The session manager can request that the application kills itself: it would do this when the session is closing down. The application can catch this request using an `XtNdieCalback`. It should not attempt to interact with the user or save state in this callback, but simply exit as cleanly as possible.

When the program does want to interact with the user, it should register an `XtNinteractCallback`. Typically, this would be used to prompt the user for the name of a file into which the application state is to be saved, or indeed to request user confirmation as to whether she really does want the current application state saved at all. The `XtNinteractCallback` does not create any graphical interface for the user interaction - the programmer should create the message dialogs as appropriate inside the callback.

The last SessionShell callback available to the programmer is the `XtNerrorCallback`, which would be used by mission-critical applications that need to be exactly informed of errors in the session manager interaction.

All of the SessionShell callbacks receive as callback data an `XtCheckPointToken`. This is a pointer to a data structure, the `XtCheckPointTokenRec`, defined as follows:

```
typedef struct _XtCheckpointTokenRec {
    int          save_type;
    int          interact_style;
    Boolean      shutdown;
    Boolean      fast;
    Boolean      cancel_shutdown;
    int          phase;
    int          interact_dialog_type;
    Boolean      request_cancel;
    Boolean      request_next_phase;
    Boolean      save_success;
    int          type;
    Widget       widget;
} XtCheckpointTokenRec, *XtCheckpointToken;
```

The exact meaning of each of the elements is fully described in the *Supplement*. For our purposes, we will confine ourselves to the following elements: `save_success`, `request_cancel`, `cancel_shutdown`, `interact_style`, and `interact_dialog_type`.

The `save_success` element indicates whether the application was able to successfully save its state. This should be set to `TRUE` or `FALSE` during the `XtNsaveCallback` depending on circumstances.

The `request_cancel` element should be set to `TRUE` if the application wants to abort the current save operation for any reason.

The `cancel_shutdown` element should be set to `TRUE` if the application wants to abort the current shutdown operation for any reason.

The `interact_style` element is set by the session manager to inform the program whether or not it is allowed to interact with the user. The possible values are:

```
SmInteractStyleNone
SmInteractStyleAny
SmInteractStyleErrors
```

The program should not attempt to interact with the user if the value is `SmInteractStyleNone`, and should only interact with the user in the case of an internal error if the style is `SmInteractStyleErrors`.

The `interact_dialog_type` is set by the programmer, and indicates back to the session manager whether any popup dialogs which will be created by the program are for the purposes of warning the user of an error, or if the dialog is an ordinary one for collecting user information or confirmation. Possible values are:

```
SmDialogError                SmDialogNormal
```

Tokens

The session management system works by passing a logical token - an identifier represented in the client by an XtCheckpointToken - between the manager and the various applications participating in the session. Each application in turn holds the token as it attempts to save its state. This token must be returned to the session manager at the termination of each deferred save callback. That is, if you pop up a dialog to request information from the user using an interact callback, the callbacks associated with this dialog should return the token, not the interact callback itself. The routine `XtSessionReturnToken`() is used to perform this task, and has the following functional signature:

```
void XtSessionReturnToken (XtCheckpointToken token)
```

The *`token`* parameter is simply the data passed through to the given session callback. It is also possible to fetch a token. What this means in the context of a session management

interaction is simply whether or not the session manager is currently in the process of talking to (*Checkpointing*) the application. The routine XtSessionGetToken() returns a token depending upon whether a current checkpoint operation is in force. It is formally defined as follows:

```
XtCheckpointToken XtSessionGetToken (Widget sessionShell)*
```

The routine returns NULL if the session manager does not have a checkpoint operation in force.

It is very important that you remember to return the token in your session management deferred callbacks, otherwise the session manager can hang awaiting a non-existent reply.

An Example

The code in Example 20-8 is a simple application which sets up various session callbacks in order to save its state. The application does nothing more than display a spinbox: the state to be saved is the current value of the spinbox.

Example 20-8 The session.c program

```
/* session.c - outlines the interactions with the session manager
*/

#include <stdio.h>
#include <Xm/Xm.h>
#include <Xm/RowColumn.h>
#include <Xm/SSpinB.h>
#include <Xm/MessageB.h>

Widget toplevel, spin;

/* The command by which the session manager will restart this application */
char *restart_command[6] = {
    NULL,
    "-xtsessionID",
    NULL,
    "-value",
    NULL,
    NULL
};

/* The "OK" button is pressed in the popup interaction dialog */
/* This does not perform save-yourself actions - */
/* it informs the session manager that we need to do so */
static void msg_ok_callback( Widget         w,
                             XtPointer      client_data,
```

* The Solaris (and other) manual pages for this routine list the function in the form:

```
XtCheckpointToken XtSessionGetToken (Widget sessionShell, int type)
```

This is a bug: there is no type parameter.

```
                              XtPointer      call_data)
{
    XtCheckpointToken token = (XtCheckpointToken) client_data;

    /* Asks the session manager to call the save_callback */
    token->save_success = True;
    /* Return the token */
    XtSessionReturnToken (token);
}

/* The "Cancel" button is pressed in the popup interaction dialog */
static void msg_cancel_callback(  Widget         w,
                                  XtPointer      client_data,
                                  XtPointer      call_data)
{
    XtCheckpointToken token = (XtCheckpointToken) client_data;

    /* Tells the session manager not to call the save_callback */
    token->request_cancel = True;
    token->save_success = False;
    /* Return the token */
    XtSessionReturnToken (token);
}

/* Interacts with the user during session shell operations */
static void interact_callback( Widget       w,
                               XtPointer   client_data,
                               XtPointer   call_data)
{
    static Widget        message = (Widget) 0;
    XtCheckpointToken    token = (XtCheckpointToken) call_data;
    XmString             xms;
    Arg                  args[8];
    int                  n;

    if (token->cancel_shutdown || token->interact_style == None) {
        token->save_success = False;
        return;
    }

    if (message == (Widget) 0) {
        n = 0;
        xms = XmStringCreateLocalized ("Save Changes Before Quitting?");
        XtSetArg (args[n], XmNmessageString, xms); n++;

        message = XmCreateQuestionDialog (toplevel, "message", args, n);

        XtUnmanageChild (XtNameToWidget (message, "Help"));

        XtAddCallback (XtNameToWidget (message, "OK"), XmNactivateCallback,
                       msg_ok_callback, call_data);
        XtAddCallback (XtNameToWidget (message, "Cancel"),
                       XmNactivateCallback, msg_cancel_callback, call_data);
    }
```

```
    XtManageChild (message);

    /* Don't return the token: we are still interacting with the user */
    /* The token is returned at a deferred time in the message dialog */
    /* callbacks */
}

/* Performs the session manager save-yourself actions */
/* That is, it sets up the XtNrestartCommand array as appropriate */
static void save_callback( Widget       w,
                           XtPointer    client_data,
                           XtPointer    call_data)
{
    XtCheckpointToken    token = (XtCheckpointToken) call_data;
    int                  spin_value;
    char                 buf[20];

    XtVaGetValues (spin, XmNposition, &spin_value, NULL);
    (void) sprintf (buf, "%d", spin_value);
    restart_command[4] = buf;
    XtVaSetValues (toplevel, XtNrestartCommand, restart_command, NULL);

    if (token->interact_style != SmInteractStyleNone) {
        if (token->interact_style == SmInteractStyleAny)
            token->interact_dialog_type = SmDialogNormal;
        else
            token->interact_dialog_type = SmDialogError;

        XtAddCallback (toplevel, XtNinteractCallback,
        interact_callback, NULL);
    }
}

/* Signals to the session manager that save completed successfully */
/* In this example, there is nothing to do */
static void save_complete_callback(   Widget       w,
                                      XtPointer    client_data,
                                      XtPointer    call_data)
{
    XtCheckpointToken token = (XtCheckpointToken) call_data;
}

/* Kills this application in response to session manager request */
static void die_callback(   Widget         w,
                            XtPointer      client_data,
                            XtPointer      call_data)
{
    XtDestroyWidget (toplevel);
    exit (0);
}

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

```
    XtAppContext     app;
    Arg              args[16];
    int              i, n;
    String           smcid;
    int              spin_value = 0;

    /* Parse the command-line arguments for -value nnn. */
    for (i = 1; i < argc; i++) {
        if ((strcmp (argv[i], "-value") == 0) && (i < argc - 1)) {
            spin_value = atoi (argv[++i]);
        }
    }

    XtSetLanguageProc (NULL, NULL, NULL);

    toplevel = XtOpenApplication (&app, "Demos", NULL, 0, &argc, argv, NULL,
                                  sessionShellWidgetClass, args, n);

    /* Set up the restart command */
    XtVaGetValues (toplevel, XtNsessionID, &smcid, NULL);

    restart_command[0] = argv[0];
    restart_command[2] = XtNewString (smcid);
    restart_command[4] = "0";
    XtVaSetValues (toplevel, XtNrestartCommand, restart_command, NULL);

    /* Set up the session manager callbacks */
    XtAddCallback (toplevel, XtNsaveCallback, save_callback, NULL);
    XtAddCallback (toplevel, XtNcancelCallback,
    save_complete_callback, NULL);
    XtAddCallback (toplevel, XtNsaveCompleteCallback,
    save_complete_callback, NULL);
    XtAddCallback (toplevel, XtNdieCallback, die_callback, NULL);

    n = 0;
    XtSetArg (args[n], XmNspinBoxChildType, XmNUMERIC); n++;
    XtSetArg (args[n], XmNcolumns, 2);                  n++;
    XtSetArg (args[n], XmNeditable, FALSE);             n++;
    XtSetArg (args[n], XmNposition, spin_value);        n++;
    XtSetArg (args[n], XmNminimumValue, 0);             n++;
    XtSetArg (args[n], XmNmaximumValue, 99);            n++;
    XtSetArg (args[n], XmNwrap, TRUE);                  n++;

spin = XmCreateSimpleSpinBox (toplevel, "spin", args, n);

    XtManageChild (spin);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}
```

The output of this program is given in Figure 20-2.

Figure 20-2 Output of session program

Customized Protocols

The previous section demonstrated how similar one protocol message is to the next in the way they are added to a program. Adding a completely new protocol is not difficult either. The only changes we have to make are those that would otherwise interfere with the standard protocols and properties that are registered with the X protocol and ICCCM. To avoid conflicts, the convention is to begin the name of non-standard atoms and window properties with at least an underscore, and possibly a more detailed prefix that identifies the atom as a private protocol or property. Accordingly, Motif provides the property `_MOTIF_WM_MESSAGES` as a private atom specifically for Motif-based applications that wish to send private messages to themselves or one another. Private does not mean that no one else can see the messages; it just implies that the protocol is not publicly available for other third-party applications to use, so don't expect other programs on the desktop to participate in the protocol.

Example 20-9 demonstrates how to register your own protocol with the shell and set up a callback routine that is invoked when that protocol is delivered. Like Example 20-6, this program is a skeletal frame only; it does not have any real functionality.*

Example 20-9 The wm_protocols.c program

```
/* wm_protocol.c -- demonstrate how to add your own protocol to a
** shell. The nature of the protocol isn't important; however, it
** must be registered with the _MOTIF_WM_MESSAGES property on the
** shell. We also add a menu item to the window manager frame's
** window menu to allow the user to activate the protocol, if desired.
*/

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

* `XtVaAppInitialize()` is considered deprecated in X11R6. `XmInternAtom()` is marked for deprecation in Motif 2.0.

```
#include <stdio.h>

main (int argc, char *argv[])
{
    Widget           toplevel;
    XtAppContext     app;
    Atom             MOTIF_MSGS, MY_PROTOCOL;
    void             my_proto_callback(Widget, XtPointer, XtPointer);
    char             buf[64];

    XtSetLanguageProc (NULL, NULL, NULL);
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv, NULL,
                                    sessionShellWidgetClass,
                                    XmNwidth, 100,
                                    XmNheight, 100,
                                    NULL);

    /* get the MOTIF_MSGS and MY_PROTOCOL atoms */
    MY_PROTOCOL = XInternAtom (XtDisplay (toplevel), "_MY_PROTOCOL", False);
    MOTIF_MSGS = XInternAtom (XtDisplay (toplevel),
                              "_MOTIF_WM_MESSAGES", False);
    /* Add MY_PROTOCOL to the _MOTIF_WM_MESSAGES VendorShell-defined
    ** property on the shell. Add a callback for this protocol.
    */
    XmAddProtocols (toplevel, MOTIF_MSGS, &MY_PROTOCOL, 1);
    XmAddProtocolCallback (toplevel, MOTIF_MSGS, MY_PROTOCOL,
                           my_proto_callback, NULL);
    /* allow the user to activate the protocol through the window manager's
    ** window menu on the shell.
    */
    sprintf (buf, "MyProtocol _P Ctrl<Key>P f.send_msg %d", MY_PROTOCOL);
    XtVaSetValues (toplevel, XmNmwmMenu, buf, NULL);

    /* create widgets... */
    ...

    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

/* called if _MY_PROTOCOL was activated, a client message was sent...
*/
void my_proto_callback (Widget widget, XtPointer client_data,
                        XtPointer call_data)
{
    puts ("My protocol got activated!");
}
```

This program is set up to receive the protocol `_MY_PROTOCOL`. If the message is sent, the function `my_proto_callback()` is called, passing the appropriate client data and callback structure as before. However, since we just made up the protocol, the only way it

can be delivered is by the window manager if (and only if) the user selects the new menu item that we attached to the window menu, as shown in Figure 20-3.

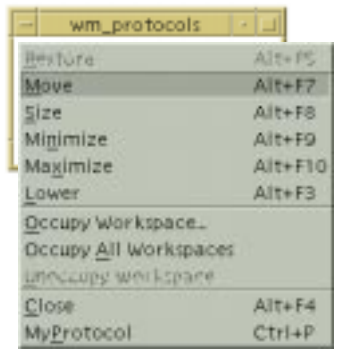


Figure 20-3 Output of wm_protocol program

The menu item is added using the `XmNmwmMenu` resource in the call to `XtVaSetValues()`.The syntax of the value for the string used by the `XmNmwmMenu` resource is described completely in the *mwm* documentation in Volume 6B, *Motif Reference Manual*. Briefly, each of the arguments refers to a single entry in the menu that is always added after the last standard protocol in the menu, which is usually the *Close* button. The syntax for the resource is:

```
label [mnemonic] [accelerator] function
```

Only the label and the window manager function (*mwm*-specific) are required. The label is always first; if a space needs to be embedded in the label, precede it by two backslashes. The next token is parsed as a mnemonic if it starts with an underscore. If an accelerator is given, the Motif toolkit parses this string and creates a corresponding accelerator text string for the menu. Finally, the parser looks for a window manager function as described by the *mwm* documentation. These include `f.move`, `f.raise` and `f.send_msg`, for example. We use `f.send_msg` to tell *mwm* to send the specified client message to the application.

It is possible to deactivate a protocol on the window menu using `XmDeactivateWMProtocol()`. Deactivation makes a protocol insensitive (unselectable). Protocols may be reactivated by `XmActivateWMProtocol()`; new protocols are automatically activated when they are added `XmActivateProtocol()` and `XmDeactivateProtocol()` perform an analogous function for non-window manager protocols.

But what can you do with your own private protocol? These protocols can come in handy if you want to attach any application-specific functionality to a window so that it can communicate with similar applications on the desktop. For example, larger application suites that contain multiple programs might need to communicate with one another through this protocol. If a suite of painting, drawing, and desktop publishing products wanted to

pass document information to one another, they could pass messages using their own protocol.Whether or not you allow the window manager (and thus the user) to participate in the protocol can be controlled by whether you make the protocol handle available in the window menu, as shown in Figure 20-2.

Advanced work with protocols is getting beyond the scope of this book. Further progress requires Xlib-level code that you can research on your own by reading portions of Volume 1, *Xlib Programming Manual*. However, if you are interested in providing this kind of functionality, you might consider the following design approach:

- When an application is interested in communicating via a private protocol, it should place a property on its top-level windows that express this interest. For example, let's call this atom `_MYAPP_CLIENT_PROP`. The atom can be added to the `WM_PROTOCOLS` property already on the window using `XmAddWMProtocol()`, just as we did earlier. An application can also choose to use `XChangeProperty()` to actually use the atom as the property itself; `XChangeProperty()` adds a new property to a window's list of existing properties.
- An application interested in seeking out other windows that have expressed interest in `_MYAPP_CLIENT_PROP` can call `XQueryTree()` to start at the root window and search all of its immediate children for those windows that have that property. The function `XGetWindowProperty()` can be used to test for the existence of the property itself.
- When an application finds a window that contains the property, it can use `XSendEvent()` to send an `XClientMessageEvent` to that window. When sending a client message, the application can either do what the Motif toolkit does and send a `WM_PROTOCOLS` message, or it can just send the `_MYAPP_CLIENT_PROP` atom itself. If the program uses the first technique, the `data.l[0]` field of the `XClientMessageEvent` data structure contains the value `WM_PROTOCOLS`, and the `data.l[1]` field contains `_MYAPP_CLIENT_PROP`.If the receiving window is part of a Motif application that has registered a callback function for this protocol, the function is invoked.*
- If the sending application wishes to send any additional data to the receiving application, it should either add or replace the receiving window's `_MYAPP_CLIENT_PROP` property and upgrade or change its value.

Remember, since this is your own private protocol, you can do whatever you like in the correspondence process. If you wanted, you could specify that the receiving window would always test for a newly-defined property on its window, and if that property is set, obtain further information from the primary selection. Using this process, you could write your

* Whether or not the receiving application is a Motif application, it can set up its own event handler to trap for the client message.

own data transfer methods. However, whatever you come up with is strictly private, so no other application can participate in your protocol unless you tell the developer of the other application what to do.

You can place whatever information you like in properties: a string, an integer, or a data structure. Just make sure that it's not per-process information like a file descriptor. This type of data cannot be shared among separate processes. You should also try not to make the information host-specific because you are not guaranteed that both clients are going to be running on the same computer, although they will be running on the same server. It is also a good idea to avoid protocols that involve continuous chatting between programs. Protocols are not a good method for doing interactive talk programs because the network can't handle that kind of traffic. To do this kind of communication, it is typically better to establish your own TCP or STREAM connection between the two applications. You should attempt to be as network-portable as possible, but this is your own personal protocol, so you can do anything you like.

Summary

The best applications can still function adequately without a window manager. For portability reasons, you should not assume that the user is running *mwm*. Except for dealing with `WM_DELETE_WINDOW` protocol messages to handle the window menu's *Close* button, you should avoid interfering with the interaction between your applications and the window manager. Despite this advice, many developers believe they know better and attempt to redesign Motif on a per-application basis. If you attempt to go this route, be aware of the guidelines provided by the *Motif Style Guide* and the ICCCM.

Client messages can be an extremely powerful tool for a large application with many top-level windows that need to interact with each other. They can also be useful for larger groups of similar applications by the same vendor that need to talk to one another. The secret to making a private protocol work is establishing a good communication channel and being able to transfer a lot of information without having to transfer a lot of data.

Exercises

These exercises are designed to help you understand the material that was presented in this chapter.

1. Write a program that always places its error dialogs in the center of the screen.
2. Whenever a shell changes from normal state to iconic state, the window manager changes the shell's `WM_STATE` property. Write a program that gets the `PropertyNotify` event generated from this state change so that you can track when a shell is iconified and de-iconified. Use `XtAddEventHandler()` to register a routine that tracks

for the event in the same way we tracked for `ConfigureNotify` events in *set_minimum.c*

3. Write a program so that when the user selects the *Close* button from a window menu, the shell iconifies itself if it is a TopLevelShell, and destroys itself if it is a DialogShell.

In this chapter:

- *Simple Clipboard Copy and Retrieval*
- *Copy by Name*
- *Clipboard Data Formats*
- *The Primary Selection and the Clipboard*
- *Implementation Issues*
- *Summary*

21

The Clipboard

This chapter describes a way for the application to interact with other applications. Data is placed on the clipboard, where it can be accessed by other windows on the desktop regardless of the applications with which they are associated. From Motif 2.0 and onwards, many of the functions and methods which are described here have been subsumed into the Uniform Transfer Model, described in Chapter 23. Although not officially marked as deprecated, the Motif Toolkit performs some of the housekeeping operations described in this chapter internally to the model on behalf of the programmer.

Imagine a group of people in a room; the only way for them to communicate is by writing messages on paper, placing the paper on a clipboard, and passing the clipboard around. A single person acts as the moderator and holds the clipboard at all times. If someone wants to post a note, she writes the message on a slip of paper and hands the message to the moderator. The note is now available for anyone to read. However, those who read the message do not remove the message from the clipboard; rather, they copy what was written. There is no guarantee that anyone will want to look at any particular message, but it is there nonetheless and will remain there until someone writes a new one.

This scenario is the concept behind the Motif clipboard: a data transfer mechanism that enables widgets to make data available for other widgets, including those in separate applications. Information of any size or type can be passed using the clipboard interface. The most common example of this data transfer model is *cut and paste*, a method by which the user can move or copy text between windows. Here, the user interacts with a Text widget that contains some text that she wishes to transfer to another Text widget. The user first *selects* the text she wants to transfer by clicking the left mouse button and dragging it across the entire area to be copied. Then, she moves the pointer to the target widget and *pastes* the text by clicking the middle mouse button.*

This action causes the text to appear to be copied to the new window. However, the text does not actually move; it is copied to the clipboard, from which the second widget then

* This is the default cut and paste user model; the user may override it using resources or keyboard equivalents. The actual method for performing this task is not the point of discussion here.

copies it into its own window. The original data may have been changed or destroyed since it was sent to the clipboard, but that is of no concern to the second widget.

An object that wishes to place data on the clipboard or read data from it is called a *client* of the clipboard (one of the people in our imaginary room). Since only one client may access the clipboard at a time, whether it is storing or retrieving data, requesting access to the clipboard implies "locking" it. If another widget already has locked the clipboard, the client must wait and ask for it again later (after the current holder has "unlocked" it).

Now, imagine that the people in the room have all sorts of items besides text messages they wish to make available for copy. Some may have pictures, records, tapes - anything. Their "cargo" must be deliverable by the moderator to anyone who requests it. To deal with this situation, the moderator must know what type of cargo she will be handling. Therefore, certain information must be registered with the moderator before cargo may be sent or received through the clipboard mechanism. Once a particular cargo type is registered, anyone may post or request such cargo to or from the moderator.

In the Motif toolkit, different types of cargo are referred to as *formats*. With respect to the X server and client applications, text messages are the most commonly used format of clipboard messages and are therefore registered by default.* Application-specific data structures must be registered separately, perhaps on a per-application basis. Once a new data type is registered, even clients that exist on other computer architectures where data is not represented identically (e.g., due to byte swapping) can use that data type, since the clipboard registration handles the proper data conversion.

There are some situations where it is impractical to place complete information on the clipboard. Some people's cargo may be "too heavy" for the clipboard to hold indefinitely. Other people may have perishables that don't last very long. Still others may have information that varies with the state of the world. For these cases, the person with the special cargo may choose to leave only some information about their cargo rather than the cargo itself. This information might include its weight, type, name and/or reference number, for example. Potential recipients may then examine the clipboard and inquire about the cargo without having to get it or even look at it. Only in the event that someone else wishes to obtain the cargo is the original owner called upon to provide it.

In the Motif world, this scenario describes clipboard data that is available *by name*. For example, if a client wishes to place an entire file on the clipboard, it might choose to register the file by name without providing the actual contents unless someone requests it. This may save a lot of time and resources, since it is possible that no one will request it. Referencing data this way is very cheap and is not subject to expiration or obsolescence.

* There are also other types that are automatically registered, such as integers. A complete list is given in the *Clipboard Data Formats* Section.

When posting messages by name, the client must provide the clipboard with a callback function that returns the actual data. This callback function may be called by the Motif toolkit at any time, provided another client requests the data. If the data is time-dependent or subject to other criteria (someone removed or changed the file), the callback routine may respond accordingly.

The Motif clipboard functions are based on X's Inter-Client Communications Conventions Manual (ICCCM). Knowledge of these conventions will aid greatly in your understanding of how these functions are implemented. However, knowledge of the implementation is not required in order to understand the concepts involved here or to be able to use the clipboard effectively through Motif's application interface. This chapter does not address many of the issues involved with the ICCCM and the lower-level Xlib properties that implement them. Rather, it only addresses the highest level of interaction provided by the Motif toolkit.

Also note that the clipboard is one of three commonly used mechanisms to support interclient communication. There are also the primary and *secondary* selections, which are similar in nature, but are handled differently at the application and user level.

The Motif 1.2 toolkit supports convenience routines that interact with clipboard selections only. To use the other selection mechanisms, you have to use X Toolkit Intrinsics functions discussed in Volume 4, *X Toolkit Intrinsics Programming Manual*. Note, however, that the Text widget supports both mechanisms.

The Motif 2.1 toolkit handles primary and secondary selections in a more consistent manner through the Uniform Transfer Model mechanisms. See Chapter 23 for more details.

Simple Clipboard Copy and Retrieval

To introduce the application programmer's interface (API) for the clipboard functions, we demonstrate how to handle simple copy and retrieval of text. The cut and paste functions provided by the Text widgets handle copy and retrieval from the clipboard in the manner we are about to describe; they also support interaction with the primary and secondary selection mechanisms. However, as pointed out in Chapter 18, *Text Widgets*, these functions are usually reserved for interactive actions taken by the user. Fortunately, Motif provides many convenience functions that facilitate the task of dealing with the clipboard for Text widgets. This section discusses the techniques used by the Text widget when it interacts with the clipboard.

Let's begin with the short program in Example 21-1. This program creates two PushButtons that have complementary callback routines: `to_clipbd()` copies text to the clipboard and `from_clipbd()` retrieves text from the clipboard. For this example, the text copied to the clipboard is arbitrary; we happen to use a string that represents the number of times the *Copy to Clipboard* button is pressed.*

Example 21-1 The copy_retrieve.c program

```
/* copy_retrieve.c -- simple copy and retrieve program. Two
** pushbuttons: the first places text in the clipboard, the other
** receives text from the clipboard. This just demonstrates the
** API involved.
*/

#include <Xm/CutPaste.h>
#include <Xm/RowColumn.h>
#include <Xm/PushB.h>

static void to_clipbd(Widget, XtPointer, XtPointer);
static void from_clipbd(Widget, XtPointer, XtPointer);

main (int argc, char *argv[])
{
    Widget         toplevel, rowcol, button;
    XtAppContext   app;

    XtSetLanguageProc (NULL, NULL, NULL);
    /* Initialize toolkit, application context and session shell */
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv, NULL,
                                    sessionShellWidgetClass, NULL);
    /* manage two buttons in a RowColumn widget */
    rowcol = XmCreateRowColumn (toplevel, "rowcol", NULL, 0);
    /* button1 copies to the clipboard */
    button = XmCreatePushButton (rowcol, "Copy To Clipboard", NULL, 0);
    XtAddCallback (button, XmNactivateCallback, to_clipbd, "text");
    XtManageChild (button);
    /* button2 retrieves text stored in the clipboard */
    button = XmCreatePushButton (rowcol, "Retrieve From Clipboard", NULL, 0);
    XtAddCallback (button, XmNactivateCallback, from_clipbd, NULL);
    XtManageChild (button);
    /* manage RowColumn, realize toplevel shell and start main loop */
    XtManageChild (rowcol);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

/* copy data to clipboard. */
static void to_clipbd (Widget widget, XtPointer client_data,
                                XtPointer call_data)
{
    long             item_id = 0; /* clipboard item id */
    int              status;
    XmString         clip_label;
    char             buf[32];
    static int       cnt;
    Display          *dpy = XtDisplayOfObject (widget);
```

* `XtVaAppInitialize()` is considered deprecated in X11R6. The SessionShell is only available from X11R6 onwards.

```
    Window          window = XtWindowOfObject (widget);
    char            *data = (char *) client_data;

    sprintf (buf, "%s-%d", data, ++cnt); /* make each copy unique */
    clip_label = XmStringCreateLocalized ("to_clipbd");

    /* start a copy -- retry till unlocked */
    do
        status = XmClipboardStartCopy (dpy, window, clip_label, CurrentTime,
                                       NULL, NULL, &item_id);
    while (status == ClipboardLocked);
    XmStringFree (clip_label);

    /* copy the data (buf) -- pass "cnt" as private id for kicks */
    do
        status = XmClipboardCopy (dpy, window, item_id, "STRING", buf,
                                  (long) strlen (buf)+1, cnt, NULL);
    while (status == ClipboardLocked);

    /* end the copy */
    do
        status = XmClipboardEndCopy (dpy, window, item_id);
    while (status == ClipboardLocked);

    printf ("Copied \"%s\" to clipboard.\n", buf);
}

static void from_clipbd (Widget widget, XtPointer client_data,
                                  XtPointer call_data)
{
    int       status;
    long      private_id;
    char      buf[32];
    Display   *dpy = XtDisplayOfObject (widget);
    Window    window = XtWindowOfObject (widget);

    do
        status = XmClipboardRetrieve (dpy, window, "STRING", buf,
                                      sizeof (buf), NULL, &private_id);
    while (status == ClipboardLocked);

    if (status == ClipboardSuccess)
        printf ("Retrieved \"%s\" (private id = %d).\n", buf, private_id);
}
```

The program uses the header file *<Xm/CutPaste.h>* to include the appropriate function declarations and various constants.* The `to_clipbd()` callback routine uses the following clipboard functions to copy data to the clipboard:

```
XmClipboardStartCopy ()
```

* *CutPaste.h* is derived from the phrase “cut and paste,” which historically has been used to describe clipboard-type operations.

```
XmClipboardCopy ()
XmClipboardEndCopy ()
```

Copying data to the clipboard is a three-phase process. Each of the functions locks the clipboard so that other clients cannot access it. Since locking the clipboard is done on a per-window basis, the object that locks the clipboard should have an associated window, which means that gadgets may not work.* When the clipboard is locked, only requests from objects with the same window ID can access the clipboard. Each time an object requests a lock on the clipboard, a counter is incremented so that matching unlock requests can be honored.

`XmClipboardStartCopy()` sets up internal storage for the copy to take place, `XmClipboardCopy()` sends the data to the clipboard, and `XmClipboardEndCopy()` frees the internal supporting structures. When copying data to the clipboard, including copies by name, all three functions must be used.

The `from_clipbd()` callback routine uses `XmClipboardRetrieveCopy()` to retrieve data from the clipboard. Only a single call is needed for the retrieval of short items, as in this example. However, a three-step process similar to that for copying data to the clipboard is required for the incremental retrieval of large amounts of data. We will cover these functions shortly.

Copying Data

The syntax of the functions that copy data to the clipboard is outlined below. Due to the intricacies involved in providing data to the clipboard, these functions take a larger number of parameters than you might expect from the simple examples given so far. Later examples should clarify the intended usage of these functions and their corresponding parameters. Each of the routines takes a pointer to the `Display` and the `Window` associated with the object making the clipboard request. These parameters may be derived from any widget or gadget using `XtDisplayOfObject()` and `XtWindowOfObject()`.

`XmClipboardStartCopy()` takes the following form:

```
int XmClipboardStartCopy ( Display          *display,
                           Window           window,
                           XmString         label,
                           Time             timestamp,
                           Widget           widget,
                           XmCutPasteProc   callback,
                           long             *item_id)
```

The *widget* and *callback* parameters are only used when registering data by name (see the *Copy by Name* Section). Although the *label* parameter is currently unused, its purpose

* Gadgets happen to work in some cases because of their window-based widget parents. However, some of the clipboard functions use `XtWindow()` rather than `XtWindowOfObject()` to get the window of an object. These functions do not work for gadgets.

is to label the data so that certain applications can view the contents of the clipboard. The *timestamp* identifies the server time when the cut took place (CurrentTime is the typical value). The *item_id* parameter is filled in by the toolkit and is returned to the client for use in subsequent clipboard function calls. This value identifies the item's entry in the clipboard.

XmClipboardCopy() has the following form:

```
int XmClipboardCopy ( Display           *display,
                      Window            window,
                      long              item_id,
                      char              *format_name,
                      XtPointer         buffer,
                      unsigned long     length,
                      long              private_id,
                      long              *data_id)
```

XmClipboardCopy() copies the data in *buffer* to the clipboard. The format of the data is described by the *format_name* parameter. This value is not a type, but a string describing the type. For example, "STRING" indicates that the data is a text string. The *length* parameter is the size of the data. Text strings can use strlen (data).

The *item_id* parameter is the ID returned by XmClipboardStartCopy(). The *data_id* parameter returns the format ID. You may pass NULL for this parameter if you are not interested in the value, however you may need it for other functions. For example, you will need it if you wish to withdraw an item from the clipboard. We will discuss this issue later when we talk about registration by name. The *private_id* parameter is an arbitrary number that is application-defined. The value is passed back to various functions, including those that handle calling by name, so we will address it further in the *Copy by Name* Section.

When copying is done, XmClipboardEndCopy() is called to free the internal data structures associated with the clipboard item. The routine takes the following form:

```
int XmClipboardEndCopy ( Display      *display,
                         Window       window,
                         long         item_id)
```

The *item_id* parameter is the ID returned by the call to XmClipboardStartCopy().

The clipboard copy functions return one of three status values: ClipboardSuccess, ClipboardLocked, or ClipboardFail. If the client is successful in gaining access to the clipboard, the routine returns ClipboardSuccess. If another client is already accessing the clipboard, the clipboard is locked and the client can loop repeatedly to attempt to gain access.

Once a copy to the clipboard is complete, you can undo it using XmClipboardUndoCopy(), which takes the following form:

```
int XmClipboardUndoCopy (Display *display, Window window)
```

You can remove an item that you have placed on the clipboard using XmClipboardWithdrawFormat(). This routine is discussed in the *Copy by Name* Section.

Retrieving Data

In Example 21-1, we retrieved the data stored on the clipboard using the function XmClipboardRetrieve(). This function takes the following form:

```
int XmClipboardRetrieve ( Display          *display,
                          Window           window,
                          char             *format_name,
                          char             *buffer,
                          unsigned long    length,
                          unsigned long    *num_bytes,
                          long             *private_id)
```

When using XmClipboardRetrieve(), you must provide buffer space to retrieve the data. In our example, we know that the data is not very large, so we declared *buffer* to have 32 bytes, which is more than adequate. The *length* parameter tells the clipboard how much space is available in *buffer*. The *num_bytes* parameter is the address of an unsigned long variable. This value is filled in by XmClipboardRetrieve() to indicate how much data it gave us. The *private_id* parameter is the address of a long; its value is the same as the *private_id* parameter passed to XmClipboardCopy(). You can pass NULL as this parameter if you are not interested in it.

If the routine is successful in retrieving the data, it returns ClipboardSuccess. If the clipboard is locked, the function returns ClipboardLocked. A rare internal error may cause the function to return ClipboardFail. If the routine does not succeed, you can choose to loop repeatedly to attempt to retrieve data.

One problem with XmClipboardRetrieve() occurs when there is more data in the clipboard than buffer space to contain it. In this case, the function copies only *length* bytes into *buffer* and sets *num_bytes* to the number of bytes it copied, which should be the same value as *length* if not enough space is available. If this situation arises, the function returns ClipboardTruncate to indicate that it did not copy everything that is available. Since we cannot just arbitrarily specify a larger data space without knowing how much data there is, we have two choices: query the clipboard to find out how much data there is or copy the data incrementally. There are advantages and disadvantages to each method. Let's start by discussing incremental retrieval.

To do an incremental retrieval, we need to introduce two functions: XmClipboardStartRetrieve() and XmClipboardEndRetrieve(). These functions are similar to the start and end copy functions discussed earlier. XmClipboardStartRetrieve() takes the following form:

```
int XmClipboardStartRetrieve ( Display      *display,
                               Window       window,
                               Time         timestamp)
```

This function locks the clipboard and notes the *timestamp*. Data placed on the clipboard after this time is considered invalid and the function returns ClipboardFail. The constant CurrentTime is typically used as this value.* XmClipboardStartRetrieve() also allocates internal data structures to support the incremental retrieval operation. Once the function is called, multiple calls to XmClipboardRetrieve() can be made until it returns ClipboardSuccess. While the routine returns ClipboardTruncate, more data needs to be read and you should continue to call the function. Be careful to save the data that has already been retrieved before the next call to the function, or you may overwrite the old data and lose information.

Once all of the data has been retrieved, call XmClipboardEndRetrieve(), which takes the following form:

```
int XmClipboardEndRetrieve (Display *display, Window window)
```

This function unlocks the clipboard and frees the internal data structures. Example 21-2 shows a callback routine that retrieves data from the clipboard incrementally. The from_clipbd_incr() routine could replace the from_clipbd() callback routine in Example 21-1.

Example 21-2 Incrementally retrieving data from the clipboard

```
static void from_clipbd_incr (Widget widget, XtPointer client_data,
                              XtPointer call_data)
{
    int              status;
    unsigned         total_bytes;
    unsigned long    received;
    char             *data = NULL, buf[32];
    Display          *dpy = XtDisplayOfObject (widget);
    Window           window = XtWindowOfObject (widget);

    do
        status = XmClipboardStartRetrieve (dpy, window, CurrentTime);
    while (status == ClipboardLocked);

    /* initialize data to contain at least one byte. */
    data = XtMalloc (1);
    total_bytes = 1;
    do {
        /* retrieve data from clipboard -- if locked, try again */
        status = XmClipboardRetrieve (dpy, window, "STRING", buf, sizeof (buf),
                                      &received, NULL);
```

* It is also common to provide the timestamp found in an event structure when available. This technique is typically used when the clipboard retrieval is initiated as a result of an action or callback routine where an event structure is available.

```
            /* reallocate data to contain enough space for everything */
            if (!(data = XtRealloc (data, total_bytes + received))) {
                XtError ("Can't allocate space for data");
                break; /* XtError may or may not return */
            }
            /* copy buf into data. strncpy() does not NULL terminate */
            strncpy (&data[total_bytes-1], buf, received);
            total_bytes += received;
        } while (status == ClipboardTruncate);

        if (data)
            data[total_bytes] = 0; /* NULL terminate */

        if (status == ClipboardSuccess)
            printf ("Retrieved \"%s\" from clipboard.\n", data);

        status = XmClipboardEndRetrieve (dpy, window);
    }
```

The callback routine works regardless of the amount of data held by the clipboard. If the client placed an entire file on the clipboard, the routine would read all of it in 32-byte increments. It is probably wise to use a larger block size when retrieving data incrementally; the constant `BUFSIZ`* is a good default choice.

The primary advantage of using the incremental retrieval method is that you do not need to allocate a potentially large amount of memory at one time. By segmenting memory, you can reuse some of it, or even discard it as each increment is read. This technique is especially useful if you are scanning for specific data and you have no intention of actually saving everything that you retrieve.

Querying the Clipboard for Data Size

The problem with incremental retrieval is that numerous round trips to the server may be necessary in order to obtain the entire contents of the clipboard. If you intend to save every bit of information you retrieve, the most economical way to handle the retrieval is by reading everything in one fell swoop. A single call to `XmClipboardRetrieve()` is more convenient than the three-step process involving locking the clipboard.

However, as pointed out earlier, we have a problem since we do not know how much data there is to read. The solution to the problem is to determine exactly how much data there is by using `XmClipboardInquireLength()`. This routine has the following form:

```
int XmClipboardInquireLength ( Display          *display,
                               Window           window,
                               char             *format,
                               unsigned long    *length)
```

* `BUFSIZ` is defined in <*stdio.h*>.

The function returns the amount of data being held by the clipboard under the specified *format_name*. In Example 21-3, we are looking for data in the "STRING" format. If any data on the clipboard is in this format, the function returns ClipboardSuccess and the length parameter is set to the number of bytes being held. If there is no data on the clipboard in the specified format, the function returns ClipboardNoData. If *length* is not set to a value other than 0, the data cannot be read from the clipboard.

If XmClipboardInquireLength() is successful, then the number of bytes specified by *length* can be allocated and the data can be retrieved in one call to XmClipboardRetrieve(). Example 21-3 shows a callback routine that retrieves data from the clipboard after querying the size of the data. The from_clipbd_query() routine could replace the from_clipbd() callback routine in Example 21-1.

Example 21-3 The from_clipbd_query() routine

```
static void from_clipbd_query (Widget widget, XtPointer client_data,
                               XtPointer call_data)
{
    int            status;
    unsigned long  recvd, length;
    char           *data;
    Display        *dpy = XtDisplayOfObject (widget);
    Window         window = XtWindowOfObject (widget);

    do
        status = XmClipboardInquireLength (dpy, window, "STRING", &length);
    while (status == ClipboardLocked);

    if (length == 0)
        printf ("No data on clipboard in specified format.\n");

    data = XtMalloc (length+1);

    do
        status = XmClipboardRetrieve (dpy, window, "STRING", data, length+1,
                                      &recvd, NULL);
    while (status == ClipboardLocked);

    if (status != ClipboardSuccess || recvd != length) {
        printf ("Failed to receive all clipboard data\n");
        XtFree (data);
    }
    else
        printf ("Retrieved \"%s\" from clipboard.\n", data);
}
```

Copy by Name

As discussed earlier, there are cases where data should not be copied to the clipboard until it is requested. It is possible to copy data by name, so that the owner of the data is notified

through a callback function when the data is needed by the clipboard. Since copying large amounts of data may be expensive, time-consuming, or even impossible due to other constraints in an application, copying data by name may be the only option available. The technique is especially advantageous if the data is never requested, since time and resources are saved.

The procedure for copying data by name is quite similar to the procedure for normal copying. The application first calls XmClipboardStartCopy(), but unlike a normal copy operation, the *callback* and *widget* parameters are specified. These values indicate that the data is to be copied by name. The *callback* parameter specifies the routine that is called when the data is requested by another client. The *widget* parameter specifies the widget that receives the messages requesting the data. Since the toolkit handles the messages, any valid widget ID can be used.

XmClipboardCopy() is then called with a *buffer* value of NULL. XmClipboardEndCopy() is called as usual. When a client requests the data from the clipboard, the callback routine provided to XmClipboardStartCopy() is called and the application provides the actual data using XmClipboardCopyByName().

You can use the convenience function XmClipboardBeginCopy() instead of XmClipboardStartCopy(). The only difference between the two routines is that the convenience function does not take a timestamp parameter; it simply uses CurrentTime as the timestamp value.

The program shown in Example 21-4 demonstrates copying data to the clipboard by name.*

Example 21-4 The copy_by_name.c program

```
/* copy_by_name.c -- demonstrate clipboard copies "by-name".
** Copying by name requires that the copy *to* clipboard
** functions use the same window as the copy *from* clipboard
** functions. This is a restriction placed on the API by the
** toolkit, not by the ICCCM.
*/

#include <Xm/CutPaste.h>
#include <Xm/RowColumn.h>
#include <Xm/PushB.h>

static void to_clipbd(), from_clipbd();
Widget toplevel;

main (int argc, char *argv[])
{
    Widget           rowcol, button;
    XtAppContext     app;
```

* XtVaAppInitialize() is considered deprecated in X11R6.

```
    XtSetLanguageProc (NULL, NULL, NULL);
    /* Initialize toolkit, application context and toplevel shell */
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv, NULL,
                                    sessionShellWidgetClass, NULL);
    /* manage two buttons in a RowColumn widget */
    rowcol = XmCreateRowColumn (toplevel, "rowcol", NULL, 0);
    /* button1 copies to the clipboard */
    button = XmCreatePushButton (rowcol, "Copy To Clipboard", NULL, 0);
    XtAddCallback (button, XmNactivateCallback, to_clipbd, NULL);
    XtManageChild (button);
    /* button2 retrieves text stored in the clipboard */
    button = XmCreatePushButton (rowcol, "Retrieve From Clipboard", NULL, 0);
    XtAddCallback (button, XmNactivateCallback, from_clipbd, NULL);
    XtManageChild (button);
    /* manage RowColumn, realize toplevel shell and start main loop */
    XtManageChild (rowcol);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

static void copy_by_name (Widget widget, int *data_id, int *private_id,
                          int *reason)
{
    Display      *dpy = XtDisplay (toplevel);
    Window       window = XtWindow (toplevel);
    static int   cnt;
    int          status;
    char         buf[32];

    printf ("Copy by name:\n\treason: %s, id: %d, data_id: %d\n",
                *reason == XmCR_CLIPBOARD_DATA_REQUEST? "request" : "delete",
                *private_id, *data_id);

    if (*reason == XmCR_CLIPBOARD_DATA_REQUEST) {
        sprintf (buf, "stuff-%d", ++cnt); /* make each copy unique */

        do
            status = XmClipboardCopyByName (dpy, window, *data_id, buf,
                                    strlen (buf)+1, *private_id = cnt);
        while (status != ClipboardSuccess);
    }
}

/* copy data to clipboard */
static void to_clipbd (Widget widget, XtPointer client_data,
                       XtPointer call_data)
{
    long         item_id = 0; /* clipboard item id */
    int          status;
    XmString     clip_label;
    Display      *dpy = XtDisplay (toplevel);
    Window       window = XtWindow (toplevel);

    clip_label = XmStringCreateLocalized ("to_clipbd");
```

```
    /* start a copy. retry till unlocked */
    do
        status = XmClipboardBeginCopy (dpy, window, clip_label, widget,
                                       copy_by_name, &item_id);
    while (status == ClipboardLocked);

    /* copy by name by passing NULL as the "data", copy_by_name() as
    ** the callback and "widget" as the widget.
    */
    do
        status = XmClipboardCopy (dpy, window, item_id, "STRING", NULL, 8L, 0,
                                  NULL);
    while (status == ClipboardLocked);

    /* end the copy */
    do
        status = XmClipboardEndCopy (dpy, window, item_id);
    while (status == ClipboardLocked);
}

static void from_clipbd (Widget widget, XtPointer client_data,
                         XtPointer call_data)
{
    int             status;
    unsigned        total_bytes;
    unsigned long   received;
    char            *data = NULL, buf[32];
    Display         *dpy = XtDisplay (toplevel);
    Window          window = XtWindow (toplevel);

    do
        status = XmClipboardStartRetrieve (dpy, window, CurrentTime);
    while (status == ClipboardLocked);

    /* initialize data to contain at least one byte. */
    data = XtMalloc (1);
    total_bytes = 1;

    do {
        buf[0] = 0;
        /* retrieve data from clipboard -- if locked, try again */
        status = XmClipboardRetrieve (dpy, window, "STRING", buf, sizeof (buf),
                                      &received, NULL);
        if (status == ClipboardNoData) {
            puts ("No data on the clipboard");
            break;
        }
        /* reallocate data to contain enough space for everything */
        if (!(data = XtRealloc (data, total_bytes + received))) {
            XtError ("Can't allocate space for data");
            break; /* XtError may or may not return */
        }
        /* copy buf into data. strncpy() does not NULL terminate */
```

```
            strncpy (&data[total_bytes-1], buf, received);
            total_bytes += received;
        } while (status == ClipboardTruncate);

        data[total_bytes-1] = 0; /* NULL terminate */

        if (status == ClipboardSuccess)
            printf ("Retrieved \"%s\" from clipboard (%d bytes)\n", data,
                        total_bytes);

        status = XmClipboardEndRetrieve (dpy, window);
    }
```

Just as in Example 21-1, the function `to_clipbd()` is used to initiate copying data to the clipboard. However, rather than passing actual data, we use:

```
status = XmClipboardBeginCopy ( dpy, window, clip_label, widget,
                                copy_by_name, &item_id);
```

Passing a valid widget and a callback routine indicates that the copy-by-name method is being used. Here, the data is provided through the given callback routine when it is requested, rather than being provided immediately. The `item_id` parameter is filled in by the clipboard function to identify the particular data element. The parameter is then used in the call to copy data:

```
status = XmClipboardCopy (dpy, window, item_id, "STRING", NULL, 8L, 0, NULL);
```

Passing `NULL` as the data also indicates that the data is passed by name. The value `8L` is passed as the *`size`* parameter to indicate how much data will be sent if the data is requested. This value is important in case other clients query the clipboard to find out how much data is available to copy.

The callback function `copy_by_name()` is called either when someone requests the data from the clipboard or when another client copies new data (by name or with actual data) to the clipboard. In the first case, the data must be copied to the clipboard; in the second case, the clipboard is telling the client that it can now free its data. The callback function is an `XmCutPasteProc`, which takes the following form:

```
typedef void (*XmCutPasteProc) ( Widget  widget,
                                 int     *data_id,
                                 int    *private_id,
                                 int    *reason)
```

The *`widget`* parameter is the same as that passed to `XmClipboardStartCopy()`. The *`data_id`* argument is the ID of the data item that is returned by `XmClipboardCopy()`, and *`private_id`* is the private data passed to `XmClipboardCopy()`. The *`reason`* parameter takes the value `XmCR_CLIPBOARD_DATA_REQUEST`, which indicates that the data must be copied to the clipboard, or `XmCR_CLIPBOARD_DATA_DELETE`, which indicates that the client can delete the data from the clipboard. Although the last three

parameters are pointers to integer values, the values are read-only and changing them has no effect.

The purpose of the function is either to send the appropriate data to the clipboard or to free the data. The value of `reason` determines which action is taken. Since no data is passed to the clipboard until this callback function is called, either the data must be stored locally (in the application) or the function must be able to generate it dynamically. The example makes no assumptions or suggestions about how to create the data, since it is entirely subject to the nature of the data and/or the application.

Once the data is obtained, it is sent to the clipboard using `XmClipboardCopyByName()`. This function does not need to lock the clipboard since the clipboard is already being locked by the window that called `XmClipboardRetrieve()`. At this point in time, both routines are accessing the clipboard. If the same application is both retrieving the data and copying the data, the `XmClipboardRetrieve()` and `XmClipboardCopyByName()` routines must use the same window for their respective *`window`* parameters because otherwise deadlock will occur and the application will hang. There may be cases where you should copy data to the clipboard incrementally. The data may be large enough that allocating one large data space to handle the entire copy is unreasonable; its size may warrant sending it in smaller chunks. Moreover, data may be generated by a slow mechanism such as a database library. If the database only returns data in specific block sizes, then you need not buffer them all up and send to the clipboard with one call; you can send each block as it comes through.

Incremental copying requires multiple calls to `XmClipboardCopyByName()`. Since `XmClipboardCopyByName()` does not lock the clipboard, you need to do that yourself by calling `XmClipboardLock()`. However, you only need to call it once no matter how much data is transferred. When you are through copying the data, you need to call `XmClipboardUnlock()`. In some cases, you may need to stop sending data before the copy is complete. For example, if the database is not responding to your application or there are other extenuating circumstances, you may want to terminate the copy operation using `XmClipboardCancelCopy()`, which has the following form:

```
int XmClipboardCancelCopy ( Display    *display,
                            Window     window,
                            long       item_id)
```

When using `XmClipboardCancelCopy()`, you should not unlock the clipboard using `XmClipboardUnlock()`.

If you have copied data by name to the clipboard under a specific data format, you may withdraw it by calling `XmClipboardWithdrawFormat()`. The function takes the following form:

```
int XmClipboardWithdrawFormat ( Display     *display,
                                Window       window,
                                int          data_id)
```

Despite the name of the procedure, its main purpose is not to remove a format specification, but to remove a data element in that format from the clipboard. The *data_id* parameter is the same value that is returned by XmClipboardCopy() when the data is initially copied by name. If the specified window holds the clipboard data but it is in a different format than that specified by *data_id*, then the data is not removed from the clipboard.

Clipboard Data Formats

As discussed in the introduction, the clipboard can contain data in arbitrary formats. While the most commonly used format is text, other formats include integers, pixmaps, and arbitrary data structures. Since all applications on the desktop have access to the clipboard, any of them may register a new format and place items of that type on the clipboard.

When registering a new format, you must also register a corresponding format name and the format length in bits (8, 16, and 32). Determining the type of data on the clipboard is much easier when there is a descriptive name associated with it. The length allows applications to send and receive data without suffering from byte-swapping problems due to differing computer architectures.

To register a new format, use XmClipboardRegisterFormat(), which takes the following form:

```
int XmClipboardRegisterFormat ( Display          *display,
                                char             *format_name,
                                unsigned long    format_length)
```

The function may return ClipboardBadFormat if the format name is NULL or the format length is other than 8, 16, or 32. The format length may be specified as 0, in which case Motif will attempt to look up the default length for the given name. Table 21-1 shows the format lengths for some predefined format names.

Table 21-1 Predefined Format Names and Lengths

Format Name	Format Length
“TARGETS”	32
“MULTIPLE”	32
“TIMESTAMP”	32
“STRING”	8
“LIST_LENGTH”	32
“PIXMAP”	32
“DRAWABLE”	32
“BITMAP”	32
“FOREGROUND”	32
“BACKGROUND”	32

Table 21-1 Predefined Format Names and Lengths (continued)

Format Name	Format Length
“COLORMAP”	32
“ODIF”	8
“OWNER_OS”	8
“FILE_NAME”	8
“HOST_NAME”	8
“CHARACTER_POSITION”	32
“LINE_NUMBER”	32
“COLUMN_NUMBER”	32
“LENGTH”	32
“USER”	8
“PROCEDURE”	8
“MODULE”	8
“PROCESS”	32
“TASK”	32
“CLASS”	8
“NAME”	8
“CLIENT_WINDOW”	32
“COMPOUND_TEXT”	8

Although these format names are known, they are not necessarily registered automatically with the server; you may still need to register the one(s) you want to use. If you are specifying your own data structure as a format, you should choose an appropriate name for it and use 32 as the format size.

The following code fragment shows how you can register a data format and then copy data in that format to the clipboard:

```
long         item_id;
long         data_id;
int          status;
void         my_data_callback(Widget, long *, long *, int *);
Display      *dpy = XtDisplay (widget);
Window       window = XtWindow (widget);
XmString     label = XmStringCreateLocalized ("my data");

/* register our own data structure with clipboard. */
XmClipboardRegisterFormat (dpy, "MY_DATA_STRUCT", 32);

/* use the copy-by-name method to transfer data to clipboard */
do
    status = XmClipboardStartCopy (dpy, window, label, CurrentTime,
                                   my_data_callback, widget, &item_id);
while (status == ClipboardLocked);
```

```
XmStringFree (label); /* don't need this any more */
/* MY_DATA_SIZE is presumed to be defined as the amount of data to transfer */
do
    status = XmClipboardCopy (dpy, window, item_id, "MY_DATA_STRUCT",
                              NULL, MY_DATA_SIZE, 0, &data_id);
    /* save the data_id! */
while (status == ClipboardLocked);

do
    status = XmClipboardEndCopy (dpy, window, item_id);
while (status == ClipboardLocked);
```

Once the "MY_DATA_STRUCT" format has been registered with the server, we follow the standard procedure for copying data to the clipboard. Here, we chose to use the copy-by-name method discussed earlier. Note that we save the value of the data_id returned by XmClipboardCopy(). This value is used so that we may withdraw the data later using XmClipboardWithdrawFormat() if necessary. Note that formats are never removed from the clipboard; only data can be removed from the clipboard. Once a particular format is registered with the clipboard, it is there until the server goes down. If you plan on retrieving data held by the clipboard, you may wish to inquire about the format of the data it is holding. To do so, you must use two functions together: XmClipboardInquireCount() and XmClipboardInquireFormat(). They take the following form:

```
int XmClipboardInquireCount ( Display      *display,
                              Window        window,
                              int           *count,
                              int           *max_length)

int XmClipboardInquireFormat ( Display        *display,
                               Window           window,
                               int              index,
                               char             *format_name_buf,
                               unsigned long    buffer_len,
                               unsigned long    *copied_len)
```

XmClipboardInquireCount() returns the number of formats the clipboard knows about for the data item it is currently holding. Also returned is the string length of the longest format name. You can iterate through the formats starting from 1 (one) through *count* by calling XmClipboardInquireFormat(). The iteration number is passed as the *index* parameter. You should use this value to ensure that you can read all the format types in your search for the desired format.

Although there is only one data item stored on the clipboard at anyone time, that item may have multiple formats associated with it. While this is unusual, it is possible to handle this case by providing different formats to successive calls to XmClipboardCopy() or XmClipboardCopyByName().

The Primary Selection and the Clipboard

Since text is the most commonly used format in the clipboard, there is a natural interaction between the clipboard and windows that contain text. In most situations, it is usual (even expected) that when the user selects text, the selection should be placed on the clipboard, which is known as a copy operation. Retrieving text from the clipboard and placing it in another window is known as a paste operation. In some cases, after the data is pasted from the clipboard, the original window deletes the data it copied, which is classified as a cut operation. The clipboard uses what is commonly referred to as the cut and paste model.

The low-level implementation of the clipboard mechanism uses the X Toolkit selection mechanism. This model has additional properties that provide for more detailed communication between the clients involved. For example, cutting text from a Text widget and placing it in another widget involves more communication between the widgets than that of the clipboard copy and retrieval mechanism. When the text that was selected in the first widget is pasted in the other, the first widget may be notified to delete the selected text. This type of communication can be handled either automatically by the Text widgets or through low-level X calls where the corresponding windows of the widgets send real events called client messages to one another.

Clipboard Functions With Text Widgets

In most cases, you should not need to access the clipboard functions to perform simple text copy and retrieval (cut and paste) for Text widgets. If you need to access the clipboard above and beyond the normal selection mechanisms provided by the Text widgets, there are a number of convenience routines that deal with selections automatically. We present a brief overview of these functions here; see Chapter 18, *Text Widgets*, for detailed information.

The `XmTextCut()`, `XmTextCopy()`, and `XmTextPaste()` routines handles cutting, copying, and pasting operations for the Text widget. There are also corresponding functions for the TextField widget. `XmTextCut()` and `XmTextCopy()` take the following form:

```
Boolean XmTextCut (Widget widget, Time time)
Boolean XmTextCopy (Widget widget, Time time)
```

If there is text selected in the Text widget referred to by the *`widget`* parameter, the selected text is placed on the clipboard. For `XmTextCut()`, the selected text is also deleted from the Text widget, while for `XmTextCopy()` it is not. The functions return `True` if all of these things happen successfully. If `False` is returned, it is usually because the Text widget does not have any selected text.

The *`time`* parameter controls when the operation takes place and may be set to any server timestamp value. For example, if you are calling this function from a callback routine, you

may wish to use the time field from the event pointer in the callback structure provided by the Motif toolkit. The value CurrentTime can also be used, but there is no guarantee that this value will prevent any race conditions between other clients wanting to use the clipboard. Although race conditions are not likely, the possibility does exist. The result of the race condition is that one widget may appear to have cut or copied selected text to the clipboard when in fact another Text widget got there first.

XmTextPaste() takes the following form:

```
Boolean XmTextPaste (Widget widget)
```

XmTextPaste() gets the current data from the clipboard and places it in the Text widget. The routine returns False if there is no data on the clipboard.

XmTextCut() and XmTextCopy() only work if there is a current selection in the specified Text widget, which may be dependent on whether or not the user has made a selection. However, you can force a selection in a Text widget using XmTextSetSelection(). This routine takes the following form:

```
void XmTextSetSelection ( Widget            widget,
                          XmTextPosition    first,
                          XmTextPosition    last,
                          Time              time)
```

XmTextSetSelection() selects the text between the specified positions in the Text widget. Once the text has been selected, either XmTextCut() or XmTextCopy() may be called to place the selection on the clipboard.

Although XmTextGetSelection() does not deal with the clipboard directly, it provides a convenient way to get the current selection from the corresponding Text widget. This routine takes the following form:

```
char *XmTextGetSelection (Widget widget)
```

Note that the text returned by the routine is allocated data and must be freed by the caller using XtFree(). The function returns NULL if the specified widget does not own the text selection.

To deselect the current selection in a Text widget, you can use XmTextClearSelection(), which takes the following form:

```
void XmTextClearSelection (Widget widget, Time time)
```

The Owner of the Selection

Sometimes, if you have a large number of Text widgets, you may need to know which of the widgets has the text selection. You can determine this by using the Xlib function XGetSelectionOwner():

```
Window XGetSelectionOwner (Display *display, Atom selection)
```

The *display* parameter can be taken from any widget using XtDisplay(). The *selection* argument represents the Atom associated with the kind of selection you are looking for. For example, you can determine the Text widget that has the current clipboard selection with the following calls:*

```
Display   *dpy = XtDisplay (widget);
Atom       clipboard_atom = XInternAtom (dpy, "CLIPBOARD", False);
Window     win = XGetSelectionOwner (dpy, clipboard_atom);
Widget     text_w = XtWindowToWidget (dpy, win);
```

Implementation Issues

The Motif clipboard mechanism relies on an underlying X mechanism referred to as properties. Windows are data structures maintained by the X server; each window can have an arbitrary list of properties associated with it. Each property consists of a name (called an atom), an arbitrary amount of data, and a format. Property formats are not at all the same thing as the higher-level Motif formats - they simply indicate whether the data is a list of 8-bit, 16-bit, or 32-bit quantities, so that the server can perform byte-swapping, if appropriate. Properties are the underlying mechanism for all interclient communication, including interaction between applications and window managers, and inter-application interaction such as the transfer of selections.

In order to simplify communication over the network, property names are not passed as arbitrary-length strings, but as defined integers known as atoms. A number of standard properties (such as those used for communication between applications and window managers) are predefined and *interned*, or made known to, and cached by the server. However, application-defined atoms can also be interned with the server by calling the Xlib function XInternAtom()†. Atoms are not only used to name properties, but to name any string data that may need to be passed back and forth between a client and the server.

We started this chapter with the analogy that the Motif toolkit is the moderator of the clipboard. In reality, the clipboard itself is a property (called CLIPBOARD) that is automatically maintained by the X server. A property is uniquely identified by both an atom and a window, which means that it is possible for there to be multiple copies of a given property. However, there should be only one CLIPBOARD property active at one time, based on conventions about the use of properties set forth in a document called the *Inter-Client Communication Conventions Manual* and followed by the deeper layers of X software. ‡Among these conventions are that certain properties should only be set by application top-level windows and that only one window should own the CLIPBOARD property at any one time. When an application makes a call to XmClipboardCopy(), the

* XmInternAtom() is marked as deprecated from Motif 2.0 onwards.

† XmInternAtom() is marked as deprecated from Motif 2.0 onwards.

‡ Reprinted as *Appendix L of Volume 0, X Protocol Reference Manual*.

data is actually stored in the CLIPBOARD property of the window that was identified in the call to `XmClipboardCopy()`.

The format of the data stored in a property is defined by another property. The standard formats are based on those recommended by the ICCCM. For example, the FONT format might suggest that an application wants the font that the data string is rendered in, rather than the data string itself. At present, Motif does not support this functionality. You have to remember that formats (or targets, as they are referred to in the ICCCM) are not really things that have any functionality. They are simply names that are translated into integer atoms. The meaning of the formats to an application depends entirely on convention. At present, most applications only support the STRING format. But eventually, conventions will doubtless be articulated for doing far more with the selection mechanism.

A further complication that needs some mention is how the Motif clipboard implementation relates to the underlying X Toolkit implementation of selections. The ICCCM actually defines three separate properties that can be used for selections: PRIMARY, SECONDARY, and CLIPBOARD. Standard Xt applications, including all of the clients distributed by MIT, use the PRIMARY property for storing selections.

The SECONDARY property is designed for quicker, more transient selections. An application that makes use of this property usually copies data directly to another window instantly when the owner finishes copying data to the property. The Motif Text widget uses the SECONDARY property when the META key is down while the middle button is clicked and dragged. As soon as the selection is complete, the selected data is immediately sent to the window that has the input focus, which may be the same window.

In the standard MIT implementation, the CLIPBOARD property is used by an independent client called *xclipboard*. Keep in mind that a property stays around only as long as the window with which it is associated. When you terminate a client and close its windows, any data stored in a property on one of the client's windows is lost. If the CLIPBOARD property is associated with a client that is kept around between invocations of other applications, it embodies a consistent repository for information to be passed between applications.

The ICCCM blesses the use of both the PRIMARY and CLIPBOARD selection properties. However, you should be aware that the difference between the Motif use of the CLIPBOARD property and the use of the PRIMARY selection property by other Xt applications makes inter-operability questionable, unless you take care to handle the PRIMARY selection in your application. The X Toolkit mechanisms for handling selections are described in Volume 4, *X Toolkit Intrinsics Programming Manual*. The Motif Text widgets support both the Xt mechanism, which uses the PRIMARY selection, and the Motif clipboard, depending on the interaction. You should probably do the same for your application.

While you can manipulate properties and atoms directly using Xlib, the higher-level API provided by Motif and Xt should insulate you from many of the details and ensure that your applications inter-operate well with others. Eventually, toolkits and applications will doubtless support numerous extensions of the current clipboard and selection mechanisms.

Summary

The clipboard provides a convenient mechanism that allows applications to interact with one another in a way that is independent of the application, operating system, and system architecture. The clipboard is one of two common mechanisms used to handle data transfer between objects. The primary selection is still regarded as the most common method for data transfer between applications, mostly because it is the standard cut and paste method used to move textual data between terminal emulators like *xterm*. A secondary selection method is also available, but is not very widely used.

The Motif toolkit tries to compensate for the *de facto* standard use of the primary selection method by integrating both the primary and clipboard selections into the same set of functions. Although users seem to be oblivious to the differences, this technique has the unfortunate side effect of confusing programmers. This is partly alleviated in the 2.1 version of the Motif toolkit, where the Uniform Transfer Model provides a single interface to the transference of application data. The Uniform Transfer Model is fully described in Chapter 23.

In this chapter:
- *Using Drag and Drop*
- *The Drag and Drop Model*
- *Customizing Built-in Drag and Drop*
- *Working With Drag Sources*
- *Working With Drop Sites*
- *Summary*

22

Drag and Drop

This chapter describes the drag and drop mechanism provided by the Motif toolkit. Drag and drop can be used to transfer data within and between applications on the desktop. Although not marked as deprecated, from Motif 2.0 onwards many of the routines described within this chapter are subsumed into the Uniform Transfer Model, which is described in Chapter 23. The Uniform Transfer Model is concerned primarily with data transfer; it does not concern itself directly with the visuals associated with a transfer. You should still read this chapter if you are interested in providing customised feedback during mouse-based data transfer operations, and if you want to gain an understanding of the mechanisms which now partly sit underneath the Uniform Transfer Model.

A graphical user interface provides objects that the user can manipulate and actions that can be performed on those objects. The drag and drop mechanism for transferring data is a natural one for a GUI, as drag and drop allows the user to transport data within and between applications by dragging an iconic representation of the data from one location to another.

An important question that a developer needs to consider is whether or not drag and drop is appropriate for a particular application. You need to think about the data that is manipulated by the application, the actions that can be performed on the data, and whether the drag and drop metaphor makes sense in this context. This decision involves figuring out if drag and drop allows you to enhance the usability of your application by making it easier for the user to perform various tasks.

For example, an electronic mail application might allow the user to drag messages that have been received into folders for storage or into a text editor for composing a response. Perhaps the most common use of drag and drop functionality is for desktop-style applications. These programs allow the user to manipulate files in the directory structure and run other applications by dragging objects around on the desktop.

Using Drag and Drop

From the user's perspective, drag and drop involves choosing a data source, dragging the data around on the desktop, and dropping the data on a new location. The mechanism is the same no matter what type of data is being manipulated. In most cases, the data is moved or

copied to the new location. However, an application can also allow the user to drag an object and drop it to invoke an action. For example, dropping a file on a printer icon could cause the file to be printed.

The *Motif Style Guide* specifies that the middle mouse button is used for drag and drop. The user starts a drag and drop transfer by pressing the second button over the data, which is referred to as the *drag source*. While the user is dragging the data, the pointer shape is changed to a *drag icon* which is a picture that represents the type of data being dragged. The drag icon is meant to provide the user with feedback about the current data transfer, so different drag icons can be used to represent textual data and graphical data, for example.

The user can drag the data to another location within the same application or to a location within another application by moving the pointer with the middle button pressed. The data can be dropped in any location that has been registered as a *dropsite*. The drop occurs when the user releases the mouse button. Figure 22-1 shows the conceptual model of drag and drop.

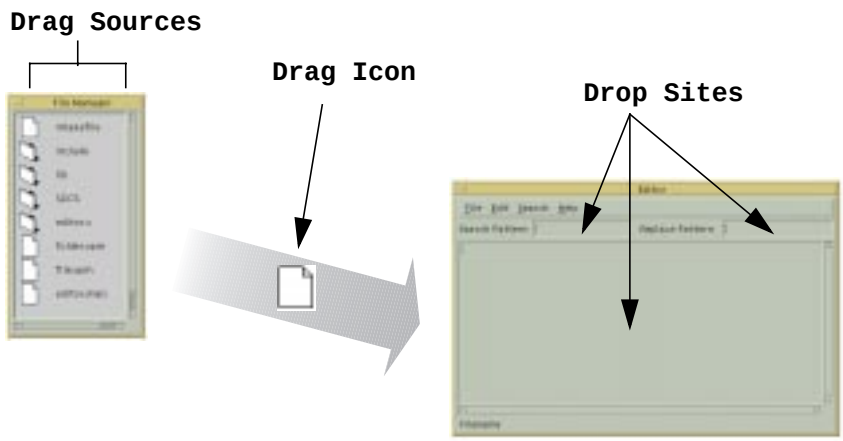


Figure 22-1 Drag and drop conceptual model

A drag and drop transfer can result in the data being moved, copied, or linked. A move operation copies the data to the drop site and then removes it from the drag source, while a copy operation copies the data to the drop site without removing it. A link operation allows the drop site access to the data in the drag source without copying it.

The default operation depends on the type of data that is being manipulated. In an editable text area, the default operation might be a move, while in a read-only area, the default operation should be a copy. A drag source can support multiple operations, in which case the user should be able to select the operation that is used. The *Motif Style Guide* specifies that the SHIFT key selects a move operation, the CTRL key selects a copy operation, and CTRL-SHIFT selects a link operation.

The user can cancel a drag at any time by pressing the ESCAPE key. The user can also request help on a drop site by pressing the HELP or F1 key before dropping the data. The help information should tell the user what will happen if the data is dropped in the drop site.

Besides representing the type of data being manipulated, the drag icon can also indicate the current operation and whether the pointer is over a valid drop site, over an invalid drop site, or not over a drop site at all. For a drop site to be valid, the drag source and the drop site must understand at least one common data format. If a drag source only provides graphical data and a drop site only understands text, the data transfer cannot succeed.

The drag icon may change as it enters and leaves drop sites to provide this state information; these changes are called *drag-over* visuals. For example, the drag icon could be displayed without any modification when it is over a valid drop site, but be superimposed with a do-not-enter symbol when the drop site is invalid. A drop site may also change its appearance when the drag icon is within it; these effects are known as *drag-under* visuals. A "garbage can" drop site might use animation to show the lid opening when a drag icon moves into the drop site. When the user performs a drop, the drag icon melts into the drop site if the data transfer is successful or springs back to the drag source if the transfer fails.

The Drag and Drop Model

The Motif implementation of drag and drop introduces a number of new programming constructs. The interaction between the different components is complex, so it may be difficult to understand just what needs to be done to implement drag and drop functionality. Since you need to understand all of the different components before you can see what your application may need, we've decided to describe all of the components of drag and drop in a somewhat abstract way before we present any examples. Although this material may be a bit dry, we think that this approach works better than presenting an example early and then having to jump around a lot to explain all of its parts. Hopefully, once you see the big picture, it will be easier to understand the different pieces more fully.

From the programmer's perspective, providing drag and drop functionality in an application can be as simple as using the Motif widgets that support drag and drop. In Motif, the Text, TextField, and List widgets are all drag sources, which means that the textual data they contain can be dragged. The Label widget and its subclasses are also drag sources for both textual and pixmap data*. The Text and TextField widgets are registered as drop sites, which means that textual data can be dropped in them. When you use these widgets in an application, you do not have to do any extra programming to provide their drag and drop capabilities since the functionality is built into the widgets.

* Under CDE Motif 2.0 or later, dragging from a Scale, Label, or LabelGadget may be turned off if the XmDisplay resource `XmNenableUnselectableDrag` is `False`.

The drag and drop capabilities provided by the Motif toolkit are highly customizable, so an application can also implement custom drag and drop transfers. Drag source and/or drop site functionality can be added to any widget. An application can provide custom drag icons and implement custom drag-under effects, such as animated drop sites. Drag and drop can be made to handle any type of data. The amount of programming required to implement custom drag and drop features varies depending on the degree of customization that is desired. While it is relatively easy to provide a new drop site for textual information, supporting drag and drop for graphical objects requires quite a bit of work.

The Motif toolkit layers the implementation of drag and drop on top of the selection mechanisms provided by the X Toolkit Intrinsics. If you are simply using the built-in drag and drop functionality, the implementation details are completely invisible. However, if you are customizing drag and drop in any way, you need to understand the underlying selection mechanisms because the drag and drop implementation is not a complete abstraction over the Xt mechanisms. For example, an application that uses custom drag sources and drop sites must provide certain selection conversion and transfer procedures in order for the data transfer to occur.

Since the Xt selection mechanisms are based on X's *Inter-Client Communications Conventions Manual* (ICCCM), the Motif implementation of drag and drop also adheres to the ICCCM. Data is transferred using properties on the server, where properties are referenced using atoms. Drag sources and drop sites also use atoms to specify the data formats, or targets, that they support. The ICCCM suggests a list of possible target types so that applications can understand each other. These targets and their meanings are shown in Table 22-1. You can also define your own targets, but unless you document them, other applications will not necessarily be able to communicate with your application using these targets.

Table 22-1 Target Types Defined by ICCCM

Atom	Type	Meaning
TARGETS	ATOM	List of valid target atoms
MULTIPLE	ATOM_PAIR	Multiple conversion requests
TIMESTAMP	INTEGER	Timestamp used to acquire selection
STRING	STRING	ISO Latin 1 text
COMPOUND_TEXT	COMPOUND_TEXT	Text in compound text encoding
TEXT	TEXT	Text in owner's encoding
LIST_LENGTH	INTEGER	Number of disjoint parts of selection
PIXMAP	DRAWABLE	Pixmap ID
DRAWABLE	DRAWABLE	Drawable ID
BITMAP	BITMAP	Bitmap ID
FOREGROUND	PIXEL	Pixel value
BACKGROUND	PIXEL	Pixel value

Table 22-1 Target Types Defined by ICCCM (continued)

Atom	Type	Meaning
ODIF	TEXT	ISO Office Document Interchange Format
OWNER_OS	TEXT	Operating system of owner
FILE_NAME	TEXT	Full path name of a file
HOST_NAME	TEXT	Host name of machine of owner
CHARACTER_POSITION	SPAN	Start and end of selection in bytes
LINE_NUMBER	SPAN	Start and end line numbers
LENGTH	INTEGER	Number of bytes in selection
USER	TEXT	Name of user running owner
PROCEDURE	TEXT	Name of selected procedure
MODULE	TEXT	Name of selected module
PROCESS	INTEGER, TEXT	Process ID of owner
TASK	INTEGER, TEXT	Task ID of owner
CLASS	TEXT	Class of owner (WM_CLASS)
NAME	TEXT	Name of owner (WM_NAME)
CLIENT_WINDOW	WINDOW	Top-level window of owner
DELETE	NULL	True if owner deleted selection
INSERT_SELECTION	NULL	Insert specified selection
INSERT_PROPERTY	NULL	Insert specified property

Motif uses some new objects to encapsulate information about various aspects of a drag and drop transfer. These objects act like widgets, in that they are created by the programmer, they have resources that can be set and retrieved, and they interact with the application using callbacks. However, they are unlike traditional widgets in that they are not visible components of the user interface. The DragContext object is used to store information during a drag, while the DropTransfer object keeps track of information during a drop. The DragIcon object is used to represent the pointer shape that is used during a drag and drop transfer. The DropSite object maintains information about all of the drop sites in an application. The Display and Screen objects also provide resources that control the behavior of drag and drop, although they are not specifically part of drag and drop.

The following sections describe all the components of a drag and drop transfer and present the Motif objects that are used to implement drag and drop. As we describe the objects, we mention many of their resources, callbacks, and related functions so that you can see how everything fits together. We describe each of the objects in much greater detail later in the chapter when we talk about how they can be used to customize different aspects of drag and drop. However, this chapter does not attempt to describe all of the possible ways in which drag and drop can be customized. We present some common situations and leave you to explore all of the details on your own. For complete information about each Motif object

used to implement drag and drop, see the appropriate reference pages in Volume 6 B, *Motif Reference Manual*.

The Drag Source

The widget that contains the data being manipulated with drag and drop is known as the drag source. When the user starts a drag, the application that contains the drag source is considered the initiator of the transfer. The data provided by a drag source depends on the type of object the source represents. For example, a Text widget provides textual data, while a DrawingArea could provide some form of graphical data.

A drag source can be designed to support and transfer any type of data. There can even be multiple formats for a given piece of data if appropriate. A drag source also specifies the operations (move, copy, or link) that it allows. The type of data, and in some cases the widget that contains the data, affects the operations that are supported. For example, the List widget only supports copy operations because it is a read-only component.

In order for a drag and drop transfer to work, the drag source and the drop site need to understand the same type of data. The drag source announces the data targets it can supply to the drop site. A drag source that supports textual data might offer the data using COMPOUND_TEXT, STRING, and TEXT targets, while a graphical drag source could provide PIXMAP, FOREGROUND, and BACKGROUND targets. When the drop occurs, the drop site can request the data in any of the targets supported by the drag source, so the drag source needs to know how to convert between supported types.

In order for a widget to be a drag source, the widget must be able to recognize a `ButtonPress` event for the second mouse button. Essentially, you need to set up a translation and action or an event handler for this event that invokes a function that starts the drag. The following code fragment shows the definition of a translation and an action for a drag source:

```
static char dragTranslations[] = "#override <Btn2Down>: StartDrag()";
static XtActionsRec dragActions[] = {
    {"StartDrag", (XtActionProc) StartDrag}
};
```

Just as with any translation and action, the application code needs to call `XtParseTranslationTable()` and `XtAppAddActions()`. The parsed translation table can be used to set the `XmNtranslations` resource for the drag source widget.

The Motif toolkit uses the DragContext object to store information about a drag source once a drag has started. This object also keeps track of state information about the transfer as it is happening. The routine that starts a drag calls `XmDragStart()` to create the DragContext and get things rolling. The DragContext object has resources that need to be set at creation time to provide information about the drag source. The `XmNdragOperations` resource specifies the operations supported by the drag source,

while XmNexportTargets and XmNnumExportTargets indicate the data targets that are supported.

The DragContext also has a number of resources that control the visual effects used during the drag. Many of these resources specify various attributes of the drag icon for the transfer. For example, the XmNsourceCursorIcon, XmNoperationCursorIcon, and XmNstateCursorIcon resources indicate the images that are used for different parts of the drag icon.If these resources are not specified, the DragContext uses default icons. There are also resources that allow you to specify different foreground and background colors for the drag icon. We describe the drag icon in more detail later in this section.

The DragContext also provides callback routines that can be used to monitor the drag and provide custom visual effects. All of the routines use special callback structures that provide information about the current state of the drag. The *Working With Drag Sources* Section provides more information about these callbacks.

The XmNconvertProc is a procedure that must be specified when a DragContext is created. This procedure is used to convert the drag source data to the format requested by the drop site when the drop occurs. The convert procedure is either an XtConvertSelectionProc or an XtConvertSelectionIncrProc, depending on whether or not the drag source is using incremental transfer. If the XmNincremental resource is set to True, the data is transferred incrementally. Both of these procedures are part of the underlying Xt selection mechanism that is not completely hidden by the Motif drag and drop abstraction. See Volume 4, *X Toolkit Intrinsics Programming Manual*, for more information on these procedures.

The following code fragment shows the creation of a DragContext object with a minimal set of resources:

```
Atom       exportList[1];
Widget     widget, dc;
Arg        args[5];
int        n;
Boolean    ConvertProc(Widget, Atom *, Atom *, Atom *, XtPointer *,
                       unsigned long *, int *);
XEvent     *event;
...
n = 0;
exportList[0] = COMPOUND_TEXT;
XtSetArg (args[n], XmNexportTargets, exportList);               n++;
XtSetArg (args[n], XmNnumExportTargets, XtNumber (exportList)); n++;
XtSetArg (args[n], XmNdragOperations, XmDROP_COPY);             n++;
XtSetArg (args[n], XmNconvertProc, ConvertProc);                n++;
dc = XmDragStart (widget, event, args, n);
```

In the *Working With Drag Sources* Section, we present an example that creates a custom drag source, and we describe the source code in detail.

Once a DragContext has been created, the Motif toolkit for the initiating application assumes control of the drag, so the application itself doesn't have to do anything during the drag. If any of the DragContext callbacks have been specified, they are called automatically by the Motif toolkit at the appropriate time.

When the drop occurs, the drop site determines whether or not the data transfer can succeed based on the operations and targets supported by the drag source and the drop site. If the transfer can succeed, the drop site initiates the transfer, which causes the `XmNconvertProc` to be called for each data target that the drop site has requested. This routine converts the data into the requested format and passes it back to the drop site, so the drop site can do whatever it needs with the data.

The Drop Site

Once the user starts a drag and drop transfer, the data can be dropped in any location that has been registered as a drop site, and if the drop site understands the data, the transfer will succeed. The application that contains the drop site where data is dropped is the receiving client in a drag and drop transfer. A drop site is always associated with a widget. Like a drag source, a drop site supports particular types of data, depending on the type of object it is.

A drop site can be designed to handle any type of data, or even multiple types if appropriate. A drop site also specifies the operations that it supports. The standard operations are to move, copy, and link data. However, a drop site can instead invoke an action as the result of a drop. For example, a "send message" drop site could send an electronic mail message when text is dropped in it.The type of object that functions as the drop site also has an effect on the supported operations. In most cases, it only makes sense for writable components to act as drop sites, since read-only components like Lists and Labels cannot be modified by the user.

Motif stores information about all of the drop sites in an application using DropSite objects. An application registers a widget as a drop site by calling `XmDropSiteRegister()` for the widget. The DropSite object uses resources to keep track of information about the drop site. This information can be set when the drop site is registered, or it can be specified later using `XmDropSiteUpdate()`; the values of the resources can be retrieved using `XmDropSiteRetrieve()`. Since a widget is being used as the handle to the drop site, you cannot use `XtVaSetValues()` and `XtVaGetValues()` to set and retrieve drop site information, as these routines manipulate the widget's resources.

Just as a drag source specifies the data types that it can process, a drop site also needs to provide this information. The `XmNimportTargets` and `XmNnumImportTargets` resources specify this information, while the `XmNdropSiteOperations` resource specifies the operations supported by the drop site.

A drop site provides visual effects when the drag icon passes through it; these effects are called drag-under effects. By default, the widget is highlighted. Other simple effects, such as a shadow border or a special pixmap, can be specified using the XmNanimationStyle resource. All of these effects are handled automatically by the toolkit on the initiating side once the resource is set. For more sophisticated effects, such as animation, a drop site must register an XmNdragProc. This callback is invoked whenever there is any drag activity in the drop site, so the application can do whatever it likes in terms of drag-under effects.

While the XmNdragProc is optional, every drop site must have a XmNdropProc registered. This routine is called when a drop occurs in the drop site. The procedure is responsible for determining whether the drop is successful or not, based on the targets and operations supported by the drag source and the drop site. The following code fragment shows how a widget that can handle compound text is registered as a drop site:

```
Arg          args[10];
int          n;
Widget       label;
Atom         importList[1];
void         HandleDrop(Widget, XtPointer, XtPointer);
...
n = 0;
importList[0] = COMPOUND_TEXT;
XtSetArg (args[n], XmNimportTargets, importList);               n++;
XtSetArg (args[n], XmNnumImportTargets, XtNumber (importList)); n++;
XtSetArg (args[n], XmNdropSiteOperations, XmDROP_COPY);         n++;
XtSetArg (args[n], XmNdropProc, HandleDrop);                    n++;
XmDropSiteRegister (label, args, n);
```

When a drop occurs in the drop site, the XmNdropProc is called automatically by the Motif toolkit. This routine must call XmDropTransferStart() whether or not the drop is successful. XmDropTransferStart() creates a DropTransfer object that maintains information about the data transfer. When The DropTransfer object is created, the XmNtransferStatus resource must be set to indicate the success or failure of the drop. If the resource is set to XmTRANSFER_FAILURE, XmDropTransferStart() does not transfer any data and merely cleans up after the drag and drop transfer.

If XmNtransferStatus is set to XmTRANSFER_SUCCESS when the DropTransfer object is created, some other resources must also be specified to cause the data to be transferred. The XmNdropTransfers and XmNnumDropTransfers resources specify the data targets to be processed, while XmNtransferProc indicates the procedure that receives the converted data from the drag source. This transfer procedure is of type XtSelectionCallbackProc. Once the data transfer has started, the routine XmDropTransferAdd() can be used to request the processing of additional data targets. In the *Working With Drop Sites* Section, we discuss in detail the tasks involved in creating a drop site.

When a drop takes place, visual effects are used to indicate the status of the transfer. Unlike the different drag effects, these visuals are not customizable. When the drop occurs, the pointer shape is changed back to the standard cursor, while the drag icon sits over the drop site. If the drop succeeds, the icon melts into the drop site. If the transfer fails or is cancelled by the user, the icon snaps back to the drag source.

A drop site is normally the size and shape of the widget with which it is associated. However, a drop site can also be shaped. The `XmNdropRectangles` and `XmNnumDropRectangles` resources control this feature. Drop sites can also be nested, so that a manager widget can be a drop site and can also contain children that are drop sites. The `XmNdropSiteType` resource controls whether the drop site is a simple drop site or a compound drop site. Drop sites have a stacking order, which means that they can overlap. When drop sites overlap, the drop site on the top of the stack obscures the drop sites beneath it, as you would expect. An application can control the stacking order of drop sites using the toolkit convenience routines `XmDropSiteQueryStackingOrder()` and `XmDropSiteConfigureStackingOrder()`.

The Drag Icon

During a drag, the pointer shape is changed into a dragicon that represents the data that is being dragged. One of the purposes of the special icon is to make it clear that a drag and drop transfer is in progress. The drag icon can also change during a drag to indicate the current status of the transfer. These visual effects are called drag-over visuals. Typical effects include changing the shape and changing the color of the icon.

A drag icon can be composed of three distinct parts, each of which is really a separate icon. The *source icon* represents the type of data that is being dragged; this icon is the only necessary component of a drag icon. The source icon for a drag that manipulates files might be an image of a piece of paper, for example. The *state icon* indicates whether the pointer is over a valid drop site, over an invalid drop site, or not over a drop site. The *operation icon* specifies the current operation. The source icon in a drag icon is static, while the state and operation icons can be dynamic. Figure 22-2 shows the components of a drag icon.

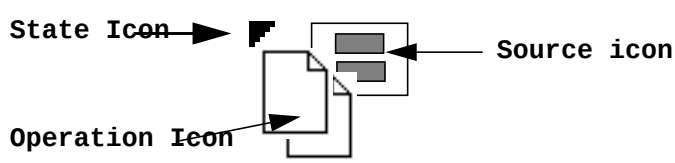


Figure 22-2 A drag icon

The Motif toolkit provides default icons for all of the different drag icon components. In Motif 2.0 and later, there are two sets of default icons; which set is displayed depends upon the value of the XmDisplay object `XmNenableDragIcon` resource. If this is `False`, the

default icons are consistent with Motif 1.2 behavior, otherwise an alternative set is displayed. The default source icons for textual data and for generic data are shown in Figure 22-3.

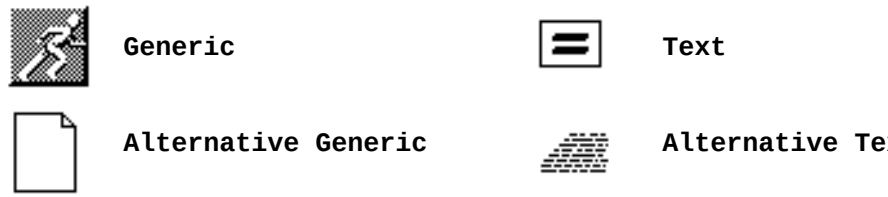


Figure 22-3 Default source icons

The default state icon for all of the different states is an arrow, as shown in Figure 22-2, while the default operation icons for the move, copy, and link operations are shown in Figure 22-4.

Figure 22-4 Default operation icons

Motif uses DragIcon objects to represent the parts of a drag icon. In order to use a custom image, you need to create each part of the icon using `XmCreateDragIcon()`. The `XmNblendModel` resource of the DragContext for a drag and drop transfer specifies the different pieces that are blended together to create the actual drag icon.

A drag icon is essentially a pixmap, and the DragIcon object encapsulates all of the information about the image. When you create a DragIcon, you specify resources that describe the image. The `XmNpixmap` and `XmNmask` resources represent the actual pixmap and its mask if you use one. Other resources include `XmNheight`, `XmNwidth`, and `XmNdepth` for specifying those attributes of the image, as well as `XmNhotX` and `XmNhotY` for indicating the x, y coordinate of the hotspot for the icon. The `XmNattachment`, `XmNoffsetX`, and `XmNoffsetY` resources specify how the icon is attached to the other parts of a drag icon.

There are a number of ways in which you can customize the drag icons that are used for drag and drop. You can specify default icons for all drag and drop transfers that start from your application by setting various resources of the Screen object. When you change the default drag icons for the Screen, the toolkit handles the drag-over effects using the icons, as we discuss in the *Customizing Built-in Drag and Drop* Section. An application can also specify custom drag icons for a particular drag and drop transfer by setting resources on its DragContext object. In this case, the application has to manage the drag-over visuals using the different DragContext callback routines.

Protocols

For drag and drop to work, the initiating and receiving applications must be able to talk to each other. The Motif toolkit supports two different mechanisms by which clients can communicate with each other during drag and drop. The main information that needs to be passed back and forth during a drag concerns the location of the drag icon relative to drop sites in the receiving application. The *dynamic* drag protocol requires messaging between the two applications, while the *preregister* drag protocol does not. During the drop, the Xt selection protocol is used to transfer the data from one application to the other.

Drag Protocols

An application can quite easily support both the dynamic and preregister drag protocols, although it can just support one or neither of the protocols if necessary. If an application does not support either of the protocols, it can still participate in drag and drop, but it does not provide any visual effects during the drag. The best approach is to support both protocols so that users can specify the protocol that is used based on their needs. By default the toolkit supports both protocols, so it is easy for an application to support both as well. The code for the initiating client is the same for both protocols, while the code for the receiver is the same except for an additional procedure that can be specified for use under the dynamic protocol.

With the preregister protocol, information about all of the drop sites in an application is stored in a database. This database is kept in a property on the top-level window of the application (or on each top-level window, if there is more than one) so that it can be read by an initiating application. During a drag, the initiator uses information in the database to manage both drag-over and drag-under visuals. Drop sites in the receiving application can set some resources to control the style of drag-under effect used, but the receiver does not participate directly in the drag.

One benefit of the preregister protocol is that it does not require dynamic communication between the initiating and receiving applications, so the performance of drag and drop does not suffer if the network is heavily loaded. However, a receiving application cannot provide sophisticated drag-under effects when the preregister protocol is being used. Under this protocol, the server is grabbed during the drag, which means that the drag icon can be any size (the size is not limited to the largest cursor size, as it is for the dynamic protocol).

Under the dynamic protocol, when the drag icon moves into a receiving application's window, the initiator sends a message to the application. Based on this message, the toolkit on the receiving side determines whether or not the drag icon is in a valid drop site. The toolkit also initializes state and operation information for the receiver, although the receiving application can update this information using its `XmNdragProc`. Based on the movement of the drag icon, the initiator receives the updated message back in one of its drag-related callbacks.

The benefit of the dynamic protocol is that the receiving application can provide sophisticated drag-under effects and drag processing using its XmNdragProc. However, the application does not have to provide these effects, as the toolkit provides some basic effects by default. The dynamic protocol also has some drawbacks. One drawback is that the messaging is expensive in terms of network traffic and may lead to unacceptable performance if the network is heavily loaded. Another limitation is that the image used for the drag icon can only be as large as the largest cursor supported by the system running the application.

The Display object provides two resources that can be set to indicate which protocol the toolkit should use when an application is the initiating or the receiving application in a drag and drop transfer. These resources are XmNdragInitiatorProtocolStyle and XmNdragReceiverProtocolStyle. An application can set the resources if it needs to specify a particular protocol, or they can be set by the user in a resource file. We describe the different values for the resources and how the actual protocols are determined in the *Customizing Built-in Drag and Drop* Section.

Drop Protocol

The protocol that is used to transfer data when the drop occurs encompasses the Xt incremental and non-incremental selection protocols. The DropTransfer object created by the receiving application handles the drop protocol. When the DropTransfer object is created using XmDropTransferStart(), the receiver specifies resources that indicate the list of desired targets, as well as an XmNtransferProc that handles the data once it has been converted by the initiator. The toolkit processes the requests one at a time by calling the XmNconvertProc of the initiating client. This procedure processes the request and passes the data back to the XmNtransferProc.

The DragContext and DropTransfer objects both have XmNincremental resources that specify whether or not the data transfer is incremental. Incremental transfers are used when the data is too large for a single X protocol request. No matter how the two resources are set, the toolkit handles the transfer of data using the underlying Xt selection mechanisms. Both the initiator and the receiver are informed about the completion of the entire transfer once all of the subtransfers are done, if there are any.

The Programming Model

If you review what we've just covered and put all of the pieces together, it creates a complex picture from the programmer's perspective. Fortunately, unless you are trying to do something really complicated, you can ignore many of the pieces and only use what you need. This section describes the complete picture by laying out the responsibilities of both the initiating and receiving applications for each step of a drag and drop operation. Figure 22-5 shows the steps graphically.

Even though most applications contain both drag sources and drop sites, it makes sense to think about the two roles separately, as the programming requirements for each are separate.

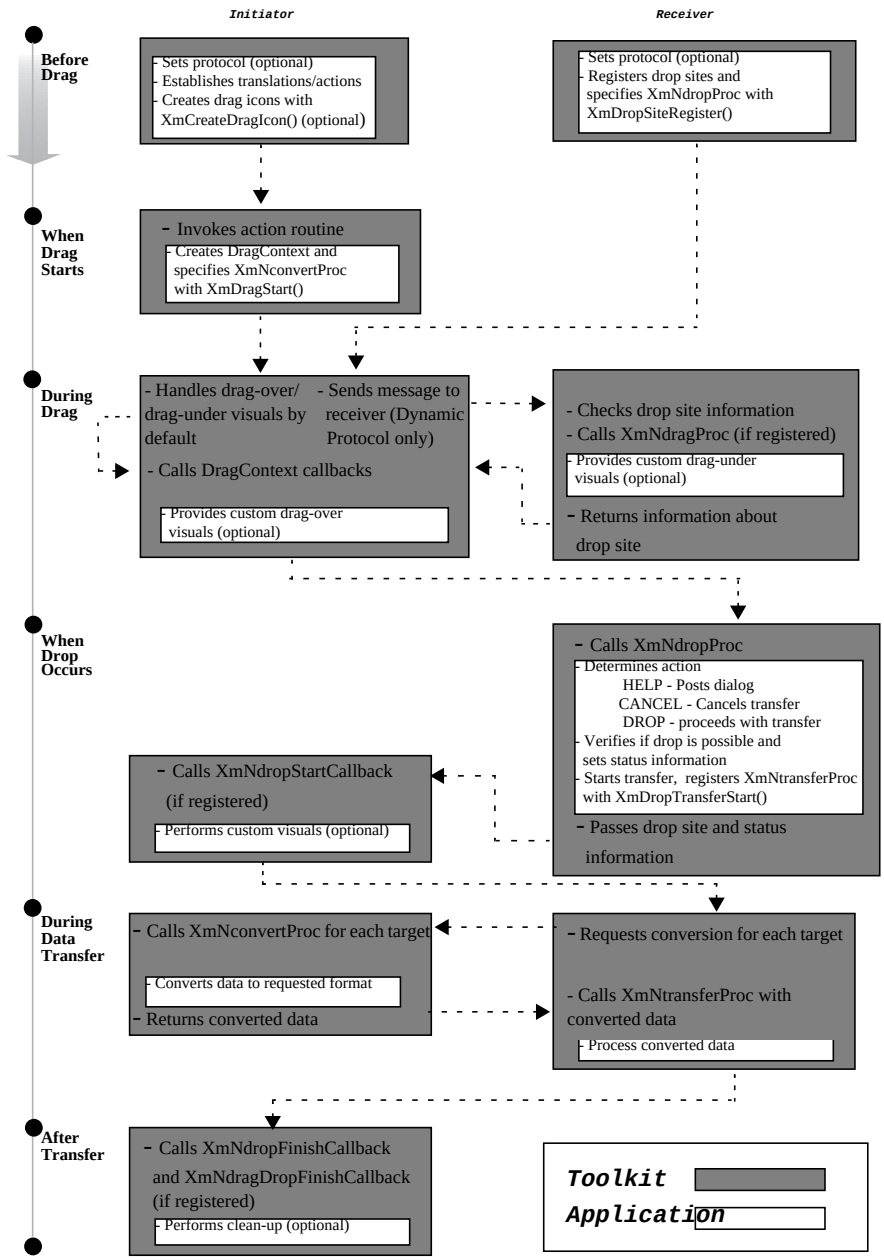


Figure 22-5 Drag and drop programming model

If the initiator and receiver are in the same application, then the same toolkit is used by both parties. Otherwise, each application is using a separate toolkit.

Before a Drag Starts

During the initialization and setup of the user interface, the initiating application needs to create any custom drag icons that it wants to use for drag-over visual effects. The initiator also needs to set up translations or event handlers to deal with `ButtonPress` events for the second mouse button. The initiator (or the user) can specify the drag protocol if necessary.

The receiving application needs to register widgets as drop sites. For each drop site, the receiver must specify the valid data targets and the `XmNdropProc` that takes over when a drop occurs in the dropsite. The receiver can also specify an `XmNdragProc` to handle special processing during the drag and custom drag-under visuals for the drop site. The receiver can query and modify the stacking order of drop sites, as well as update information about drop sites while the application is running.

When a Drag Starts

When the user starts a drag operation, the toolkit on the initiating side takes control. The application needs to create a DragContext by calling `XmDragStart()`. It must specify the valid targets for the operation and the `XmNconvertProc` that processes data transfer requests from the receiving client. The application can also specify callbacks that are invoked at various points during the drag, custom drag-over visual effects, and a drop callback that is called when the drop occurs. Receiving clients are not involved in this step.

During the Drag

By default, the toolkit on the initiating side handles all of the drag-over and drag-under visuals under both the preregister and dynamic protocols. If the preregister protocol is being used, the receiving client is not involved during the drag, but the initiating application can provide custom drag-over effects. These effects are handled by the various callbacks that can be specified fora DragContext. At any point during the drag, the initiator can cancel the transfer by calling `XmDragCancel()`.

Under the dynamic protocol, the initiating application sends messages to receiving clients to get drop site information. The toolkit on the receiving side handles these messages. If the receiver has registered an `XmNdragProc`, it is invoked each time a message is sent to the receiver. This routine can provide custom drag-under visuals and other special processing. After the `XmNdragProc` is finished, information about the drop site is passed back to the initiator, and the DragContext callbacks are invoked, so the initiator can still perform any special processing and provide custom drag-overvisuals.

When a Drop Occurs

When the user drops the data, the toolkit on the receiving side takes over from the toolkit on the initiating side. The XmNdropProc for the drop site determines what action the user has requested.If the user has requested help, the receiving application should display a help dialog and see if the user wants to proceed. If the user cancels the transfer, the drop does not proceed. Otherwise, the XmNdropProc determines if the transfer is possible by checking the targets supported by drag source.

If the drop is valid, the receiving client starts the transfer of data by calling XmDropTransferStart(). If the transfer is not valid, the routine still needs to be called to clean up the operation. If the initiator has registered an XmNdropStartCallback on its DragContext, it is invoked now. Other than this callback, the initiating client plays no role when the drop occurs.

During the Data Transfer

When the receiver calls XmDropTransferStart(), it must specify a list of data and target formats that it wants from the initiating application.The routine creates a DropTransfer object that can be updated during the transfer. The receiver must also specify an XmNtransferProc to handle the data once it has been converted by the initiator.The receiver can cancel the transfer at any point.

For each target requested by the receiver, the XmNconvertProc of the initiator is called to convert the data to the specified format. The formatted data is passed back to the receiver's XmNtransferProc. Once the entire data transfer is complete, the XmNdropFinishCallback and XmNdragDropFinishCallback callbacks of the initiating client's DragContext are invoked, if they have been specified.

Customizing Built-in Drag and Drop

The Text and TextField widgets, the List widget, and the Label widget and its subclasses all support drag and drop functionality by default*. When you use these widgets in an application, they provide built-in drag and drop capabilities. All of the widgets are drag sources for textual data, while just the Text and TextField widgets are registered as drop sites for text.

With a Label widget or a button, the user can drag the entire text string of the component by starting a drag in the component. The Label widget and its subclasses are also drag sources for graphical data, but there are no built-in drop sites for graphical data. However, when these components are in a menu, they do not function as drag sources. These components are not drop sites because they are meant to be read-only components in a user

* In Motif 2.0 and later, drag and drop for the Scale, Label and LabelGadget are by default turned off if the XmDisplay resource XmNenableUnselectableDrag is True.

interface. Most applications would not want the user to be able to change the label on a button by dropping text on it. However, if you want to provide this type of functionality, it is easy to register a Label or a button as a drop site using the technique we describe in the *Working With Drop Sites* Section.

The user can drag the text of either a single item or the current selection in a Listwidget. If the pointer is over a selected item when the drag is started, the text of the selected item is used for the drag. If multiple items are selected, the text of all of the selected items is used, where the items are separated by newlines. If the drag is started over an unselected item, the text of that item is transferred by drag and drop. The List widget is not a drop site because its items are not meant to be modified by the user. If you want to allow the user to modify a List by dropping items in it, however, you can register the widget as a drop site.

In Motif 1.2, only the Text and TextField widgets have built-in drop site functionality. The user can drop textual data from any drag source in these widgets. The widgets also function as drag sources, so the user can move and copy the current selection within and between Text and TextField widgets.

In Motif 2.0 and later, the set of widgets supporting built-in drop site capability is expanded to include the Container. In this instance, the data being transferred is widget-based, and comes from the Container itself: it allows the user to move IconGadget children within the same Container by dragging with the mouse.

Applications that simply use the built-in drag and drop capabilities of the Motif widgets can still customize various aspects of the functionality. This section explores the different types of customization that are possible.

Specifying the Drag Protocol

Motif supports two different protocols for communication between applications during a drag. The dynamic protocol passes messages between the two applications about the location of drop sites, while the preregister protocol keeps track of drop site information in a database. Since the preregister protocol does not require communication between applications, it can provide better performance on a heavily-loaded network. However, the dynamic protocol offers the advantage of sophisticated drag-under visual effects.

The programmer or the user can specify the drag protocol for an application by setting the `XmNdragInitiatorProtocolStyle` and `XmNdragReceiverProtocolStyle` resources defined by the Display object. Motif creates a Display object automatically for an application when it creates the first shell on a particular display. If an application uses multiple displays, it has a Display object for each one. An application can retrieve the Display object for a specified display using `XmGetXmDisplay()`.

The `XmNdragInitiatorProtocolStyle` and `XmNdragReceiverProtocolStyle` resources indicate the preferred drag protocol for an application when it is acting as an

initiator and as a receiver, respectively, in a drag and drop transfer. Each resource can be set to one of the following values:

XmDRAG_PREREGISTER
: This value means that the application can only support the preregister drag protocol.

XmDRAG_DYNAMIC
: This value indicates that the application can only support the dynamic drag protocol.

XmDRAG_NONE
: This value means that drag and drop is disabled for the application.

XmDRAG_DROP_ONLY
: This value specifies that the application does not support either drag protocol, but it does support drag and drop transfers. The user can transfer data using drag and drop, but there are no visual effects during the drag.

XmDRAG_PREFER_DYNAMIC
: This value means that the application supports both the preregister and dynamic protocols, but it prefers to use the dynamic protocol.

XmDRAG_PREFER_PREREGISTER
: This value means that the application supports both drag protocols, but it prefers to use the preregister protocol. The value is the default value for XmNdragReceiverProtocolStyle.

XmDRAG_PREFER_RECEIVER
: This value indicates that the application supports both the preregister and dynamic protocols, but it defers to the preference of the receiving application. The value can only be specified for the XmNdragInitiatorProtocolStyle resource, and it is the default value for the resource.

The actual protocol that is used during a drag and drop transfer is based on the preferences specified by the initiating and receiving applications. The protocol can change during a drag as the drag icon enters and leaves top-level windows. Table 22-2 shows how the protocol is resolved based on the preferred protocols for the initiator and the receiver.

Table 22-2 Drag Protocol Resolution

	Receiver Protocol Style			
Initiator Protocol Style	Preregister	Prefer Preregister	Prefer Dynamic	Dynamic
Preregister	Preregister	Preregister	Preregister	Drop Only
Prefer Preregister	Preregister	Preregister	Preregister	Dynamic
Prefer Receiver	Preregister	Preregister	Dynamic	Dynamic
Prefer Dynamic	Preregister	Dynamic	Dynamic	Dynamic
Dynamic	Drop Only	Dynamic	Dynamic	Dynamic

If two applications cannot find an agreeable protocol style, the XmDRAG_DROP_ONLY style is used. In this case, there are no drag-over or drag-under visuals except for the initial drag icon. An application can also explicitly set the protocol resources to XmDRAG_DROP_ONLY, in which case the application does not provide any visual effects during the drag.

If an application sets the resources XmNdragInitiatorProtocolStyle or XmNdragReceiverProtocolStyle to XmDRAG_NONE, the application does not participate in drag and drop as an initiator or a receiver, respectively. This value is useful for disabling drag and drop functionality, as we discuss in the next section.

The actual protocol used for a drag and drop transfer controls the visual effects that the user sees during the drag. Under the preregister protocol, the server is grabbed so the drag icon can be a pixmap of arbitrary size. The drag icon uses the depth and colormap of the drag source widget, so it can be a color image. When the dynamic protocol is used, the drag icon is implemented using the X cursor, so it must be a bitmap and is limited in size (use XQueryBestCursor() to determine the largest size for a particular hardware configuration).

An application should support both the dynamic and preregister protocols so that the user can select the protocol based on his needs. Since the toolkit supports both protocols by default, an application can easily support both as well. The code for handling drag sources is the same under both protocols. Drop sites can specify an optional XmNdragProc routine that is invoked under the dynamic protocol and can be used to provide sophisticated drag-under effects.

The only reason that you should specify the XmNdragInitiatorProtocolStyle and XmNdragReceiverProtocolStyle resources in application code is if your application is going to support only one of the drag protocols. In this case, you should set the resources to force the application to use the supported protocol. You can retrieve the Display object for the application using XmGetXmDisplay() and then use XtVaSetValues() to specify the resources. You can also use XtVaGetValues() to check the values of the protocol resources.

If your application supports both drag protocols, you can specify the protocol resources in an app-defaults file to indicate the application's preferred protocol. By default, an application uses the preregister protocol because XmNdragInitiatorProtocolStyle is set to XmDRAG_PREFER_RECEIVER and XmNdragReceiverProtocolStyle is set to XmDRAG_PREFER_PREREGISTER. If you have implemented custom drag-under visuals with an XmNdragProc, you should set the protocol resources to XmDRAG_PREFER_DYNAMIC so that the dynamic protocol is used whenever possible. You can set these resources in an app-defaults file as follows:

```
*DragInitiatorProtocolStyle: DRAG_PREFER_DYNAMIC
*DragReceiverProtocolStyle:  DRAG_PREFER_DYNAMIC
```

If you set the protocol resources in an app-defaults file, users can specify their own values in a resource file. Users that want to ensure good performance should specify a preference for the preregister protocol, while users that want sophisticated drag-under effects should indicate a preference for the dynamic protocol.

Turning Off Drag and Drop Functionality

If you do not want to provide drag and drop in an application, you can turn off the functionality in a number of ways. The most effective way to turn off the functionality is to set both XmNdragInitiatorProtocolStyle and XmNdragReceiverProtocolStyle to XmDRAG_NONE. These settings completely disable drag and drop for the application. You can also set just one of the resources to this value to prevent an application from participating in drag and drop as either an initiator or a receiver.

You can also selectively turn off individual drag sources in an application. To prevent a widget from providing its default drag source functionality, you need to override the translation for the second mouse button for the widget, as shown in the following code fragment:

```
static char dragTranslations[] = "#override <Btn2Down>: DoNothing()";
static XtActionsRec dragActions[] = {
    {"DoNothing", (XtActionProc) DoNothing}
};
```

True to its name, the DoNothing() action routine does nothing. Once you parse the translation table and add the actions to the application, you can use the translation to set the XmNtranslations resources of all of the widgets that you do not want to function as drag sources.

There are two different ways to disable the drop site functionality of a Text or TextField widget. If you want to turn off the drop site permanently, you can call XmDropSiteUnregister() for the widget. This routine removes the drop site associated with the widget, so you have to reregister it if you want to enable the drop site. To disable a drop site temporarily, it is easier to use the XmNdropSiteActivity resource defined by the DropSite object. This resource can be set to either XmDROP_SITE_ACTIVE or XmDROP_SITE_INACTIVE. When a drop site is inactive, it does not participate in drag and drop. You can set a drop site inactive using XmDropSiteUpdate(), as shown in the following code fragment:

```
Widget  text_w;
Arg     args[5];
int     n = 0;
...
XtSetArg (args[n], XmNdropSiteActivity, XmDROP_SITE_INACTIVE); n++;
XmDropSiteUpdate (text_w, args, n);
...
```

Even though drop sites are associated with widgets, you have to set DropSite resources using XmDropSiteUpdate(), not XtVaSetValues().

One situation in which you would probably want to disable a built-in drop site is when the widget is designed to be output-only. If you set the XmNeditable resource of a Text or TextField widget to False, the user cannot drop data in the widget because it is uneditable. However, the toolkit still displays the default drag-under visual effects in this case, so the widget appears as though it functions as a drop site. To make it clear that the widget is not a drop site, you can disable the drop site using one of the techniques we just described. If the widget is always uneditable, it is fine to use XmDropSiteUnregister(), but if the widget changes state, you are better off setting XmNdropSiteActivity. For certain other types of read-only widgets, an alternative exists in Motif 2.0 and later; set the XmDisplay resource XmNenableUnselectableDrag to True to turn off dragging in Label, LabelGadget, and Scale widgets.

When you set a Text or TextField widget insensitive, the user cannot interact with the widget, so it doesn't make sense for the widget to function as a drop site. However, there is currently a bug in the implementation of drag and drop such that the user can drop text in an insensitive widget. To prevent this problem, whenever you change the sensitivity of a Text widget, you should set the XmNdropSiteActivity resource to match the sensitivity.

Modifying the Visual Effects

Motif provides resources that both the user and the programmer can use to change the default drag-over visual effects that are used during a drag and drop transfer. The Screen object provides the following resources:

```
XmNdefaultSourceCursorIcon      XmNdefaultMoveCursorIcon
XmNdefaultCopyCursorIcon        XmNdefaultLinkCursorIcon
XmNdefaultValidCursorIcon       XmNdefaultInvalidCursorIcon
XmNdefaultNoneCursorIcon
```

These resources specify the default icons for all the components of a drag icon, including the different operations and states.

Motif creates a Screen object automatically for an application when it creates the first shell on a particular screen. If an application accesses multiple screens, it has a Screen object for each one. An application can retrieve the Screen object for a specified screen using XmGetXmScreen().

The drag icon resources defined by the Screen object only take effect when the XmNsourceCursorIcon, XmNoperationCursorIcon, and XmNstateCursorIcon resources have not been specified for a particular DragContext. All Motif widgets with built-in drag source functionality set the XmNsourceCursorIcon resource, so the Screen resource cannot be used to specify a different source icon for these components. The

widgets do not set the XmNoperationCursorIcon and XmNstateCursorIcon resources, so you can set the various default icons for these components.

If neither the DragContext resources nor the Screen resources are specified, Motif uses hard-coded default icons. For example, the running figure shown in Figure 22-3 is used as the source icon whenever a source icon has not been specified. Since this icon is rather arbitrary, you might want to set the XmNdefaultSourceCursorIcon resource to something more appropriate for your application.

Before you can set the Screen resources in application code, you must create DragIcon objects for the different resources. In the *Working With Drag Sources* Section we describe how to create a drag icon using XmCreateDragIcon(). Once the drag icon exists, you can retrieve the Screen object using XmGetXmScreen() and set its resources, as shown in the following code fragment:

```
Widget drag_icon, screen, toplevel;
...
screen = XmGetXmScreen (XtScreen (toplevel));
XtVaSetValues (screen, XmNdefaultSourceCursorIcon, drag_icon, NULL);
...
```

The specified icon is used whenever the source icon has not been set for the DragContext for a drag and drop transfer.

The Screen resources can also be set in a resource file. In this case, the icons can be specified as bitmap files, so the application does not have to create DragIcon objects. Both the icon and an optional mask can be specified using resources as follows:

```
*defaultSourceCursorIcon.pixmap: icon.xbm
*defaultSourceCursorIcon.mask:   icon_mask.xbm
```

Although it is convenient to be able to set the Screen resources in a resource file, this feature really isn't that useful since the Motif widgets and most applications specify their drag icons using DragContext resources.

The resources XmNvalidCursorForeground, XmNinvalidCursorForeground, and XmNnoneCursorForeground of the DragContext can be used to further distinguish between the different states in a drag and drop transfer. These resources can be specified in a resource file as follows:

```
*validCursorForeground:   green
*invalidCursorForeground: red
*noneCursorForeground:    yellow
```

In this case, the drag icon changes color as the user moves it between components that are valid drop sites, components that are invalid drop sites, and components that are not drop sites. While it is possible to modify some aspects of the drag-over effects using Screen and DragContext resources, if you really want to provide customized visual effects, you need

to understand more about the implementation of drag and drop. In the *Working With Drop Sites* Section we discuss how to provide custom drag-over effects.

Working With Drag Sources

Many applications work with data other than text. In order to provide drag and drop capabilities, these applications need to create drag sources for the data they manipulate. In this section, we describe the steps you need to follow to create a new drag source. We use an example program that displays all the files in a directory and allows the user to drag the files. However, in order for this drag to succeed, we need another application that understands files as objects and allows the user to drop files. In the *Working With Drop Sites* Section, we present a text editor that handles the dropping of file data, but for now we are just going to consider the ability to drag a file. Example 22-1 shows the *file_manager.c* application, which we are going to describe in detail in the following sections.*

Example 22-1 The file_manager.c program

```
/* file_manager.c -- displays all of the files in the current directory
** and creates a drag source for each file. The user can drag the
** contents of the file to another application that understands
** dropping file data. Demonstrates creating a drag source, creating
** drag icons, and handling data conversion.
*/

#include <Xm/Screen.h>
#include <Xm/ScrolledW.h>
#include <Xm/RowColumn.h>
#include <Xm/Form.h>
#include <Xm/Label.h>
#include <Xm/AtomMgr.h>
#include <Xm/DragDrop.h>
#include <X11/Xos.h>
#include <stdio.h>
#include <sys/stat.h>

typedef struct {
    char       *file_name;
    Boolean    is_directory;
} FileInfo;

/* global variable -- arbitrarily limit number of files to 256 */
FileInfo     files[256];
void         StartDrag(Widget, XEvent *, String *, Cardinal *);

/* translations and actions. Pressing mouse button 2 calls
** StartDrag to start a drag transaction
```

* `XtAppInitialize`() is considered deprecated in X11R6. `XmInternAtom`() is marked for deprecation in Motif 2.0 and later.

```
*/
static char dragTranslations[] = "#override <Btn2Down>: StartDrag()";
static XtActionsRec dragActions[] = {
    {"StartDrag", (XtActionProc) StartDrag}
    };

main (int argc, char *argv[])
{
    Arg                 args[12];
    int                 num_files, n, i = 0;
    Widget              toplevel, sw, panel, form, label;
    Display             *dpy;
    Atom                FILE_CONTENTS, FILE_NAME, DIRECTORY;
    XtAppContext        app;
    XtTranslations      parsed_trans;
    char                *p, buf[256];
    FILE                *pp, *popen();
    struct stat         s_buf;
    Pixmap              file, dir, pixmap;
    Pixel               fg, bg;

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

    /* intern the Atoms for data targets */
    dpy = XtDisplay (toplevel);
    FILE_CONTENTS = XInternAtom (dpy, "FILE_CONTENTS", False);
    FILE_NAME = XInternAtom (dpy, "FILE_NAME", False);
    DIRECTORY = XInternAtom (dpy, "DIRECTORY", False);

    /* use popen to get the files in the directory */
    sprintf (buf, "/bin/ls .");
    if (!(pp = popen (buf, "r"))) {
        perror (buf);
        exit (1);
    }

    /* read output from popen -- store filename and type */
    while (fgets (buf, sizeof (buf), pp) && (i < 256)) {
        if (p = index (buf, '\n'))
            *p = 0;
        if (stat (buf, &s_buf) == -1)
            continue;
        else if ((s_buf.st_mode &S_IFMT) == S_IFDIR)
            files[i].is_directory = True;
        else if (!(s_buf.st_mode & S_IFREG))
            continue;
        else
            files[i].is_directory = False;
        files[i].file_name = XtNewString (buf);
        i++;
    }
    pclose (pp);
```

```
num_files = i;

/* create a scrolled window to contain the file labels */
n = 0;
XtSetArg (args[n], XmNwidth, 200);                   n++;
XtSetArg (args[n], XmNheight, 300);                  n++;
XtSetArg (args[n], XmNscrollingPolicy, XmAUTOMATIC); n++;
sw = XmCreateScrolledWindow (toplevel, "sw", args, n);
panel = XmCreateRowColumn (sw, "panel", NULL, 0);
/* get foreground and background colors and create label pixmaps */
XtVaGetValues (panel, XmNforeground, &fg, XmNbackground, &bg, NULL);
file = XmGetPixmap (XtScreen (panel), "file.xbm", fg, bg);
dir = XmGetPixmap (XtScreen (panel), "dir.xbm", fg, bg);
if (file == XmUNSPECIFIED_PIXMAP || dir == XmUNSPECIFIED_PIXMAP) {
    puts ("Couldn't load pixmaps");
    exit (1);
}

parsed_trans = XtParseTranslationTable (dragTranslations);
XtAppAddActions (app, dragActions, XtNumber (dragActions));

/* create image and filename Labels for each file */
for (i = 0; i < num_files; i++) {
    form = XmCreateForm (panel, "form", NULL, 0);

if (files[i].is_directory)  pixmap = dir;
else                        pixmap = file;

n = 0;
/* specify translation for drag and index into file array */
XtSetArg (args[n], XmNtranslations, parsed_trans);         n++;
XtSetArg (args[n], XmNuserData, i);                        n++;
XtSetArg (args[n], XmNlabelType, XmPIXMAP);                n++;
XtSetArg (args[n], XmNlabelPixmap, pixmap);                n++;
XtSetArg (args[n], XmNtopAttachment, XmATTACH_FORM);       n++;
XtSetArg (args[n], XmNbottomAttachment, XmATTACH_FORM);    n++;
XtSetArg (args[n], XmNleftAttachment, XmATTACH_FORM);      n++;
XtSetArg (args[n], XmNrightAttachment, XmATTACH_POSITION); n++;
XtSetArg (args[n], XmNrightPosition, 25);                  n++;
label = XmCreateLabel (form, "type", args, n);
XtManageChild (label);

n = 0;
XtSetArg (args[n], XmNalignment, XmALIGNMENT_BEGINNING);   n++;
XtSetArg (args[n], XmNtopAttachment, XmATTACH_FORM);       n++;
XtSetArg (args[n], XmNbottomAttachment, XmATTACH_FORM);    n++;
XtSetArg (args[n], XmNrightAttachment, XmATTACH_FORM);     n++;
XtSetArg (args[n], XmNleftAttachment, XmATTACH_POSITION);  n++;
XtSetArg (args[n], XmNleftPosition, 25);                   n++;
    label = XmCreateLabel (form, files[i].file_name, args, n);
    XtManageChild (label);
    XtManageChild (form);
}
```

```
    XtManageChild (panel);
    XtManageChild (sw);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

/* StartDrag() -- action routine called by initiator when a drag starts
** (in this case, when mouse button 2 is pressed). It starts
** the drag processing and establishes a drag context.
*/
void StartDrag (Widget widget, XEvent *event, String *params,
                    Cardinal *num_params)
{
    Arg          args[10];
    int          n, i;
    Display      *dpy;
    Atom         FILE_CONTENTS, FILE_NAME, DIRECTORY;
    Atom         exportList[2];
    Widget       drag_icon, dc;
    Pixel        fg, bg;
    Pixmap       icon, iconmask;
    XtPointer    ptr;
    Boolean      ConvertProc(Widget, Atom *, Atom *, Atom *,
                             XtPointer *, unsigned long *, int *);
    void         DragDropFinish(Widget, XtPointer, XtPointer);

    Boolean      ConvertProc();
    void         DragDropFinish();

    /* intern the Atoms for data targets */
    dpy = XtDisplay (widget);
    FILE_CONTENTS = XInternAtom (dpy, "FILE_CONTENTS", False);
    FILE_NAME = XInternAtom (dpy, "FILE_NAME", False);
    DIRECTORY = XInternAtom (dpy, "DIRECTORY", False);

    /* get background and foreground colors and fetch index into file
    ** array from XmNuserData.
    */
    XtVaGetValues (widget, XmNbackground, &bg, XmNforeground, &fg, XmNuserData,
                    &ptr, NULL);

    /* create pixmaps for drag icon -- either file or directory */
    i = (int) ptr;
    if (files[i].is_directory) {
        icon = XmGetPixmapByDepth (XtScreen (widget), "dir.xbm", 1, 0, 1);
        iconmask = XmGetPixmapByDepth (XtScreen (widget), "dirmask.xbm",
                                                                1, 0, 1);
    }
    else {
        icon = XmGetPixmapByDepth (XtScreen (widget), "file.xbm", 1, 0, 1);
        iconmask = XmGetPixmapByDepth (XtScreen (widget), "filemask.xbm",
                                                                1, 0, 1);
    }
    if (icon == XmUNSPECIFIED_PIXMAP || iconmask == XmUNSPECIFIED_PIXMAP) {
```

```
            puts ("Couldn't load pixmaps");
            exit (1);
        }
        n = 0;
        XtSetArg (args[n], XmNpixmap, icon);   n++;
        XtSetArg (args[n], XmNmask, iconmask); n++;
        drag_icon = XmCreateDragIcon (widget, "drag_icon", args, n);

        /* specify resources for DragContext for the transfer */
        n = 0;
        XtSetArg (args[n], XmNblendModel, XmBLEND_ALL);       n++;
        XtSetArg (args[n], XmNcursorBackground, bg);          n++;
        XtSetArg (args[n], XmNcursorForeground, fg);          n++;
        XtSetArg (args[n], XmNsourceCursorIcon, drag_icon);   n++;

        /* establish the list of valid target types */
        if (files[i].is_directory) {
            exportList[0] = DIRECTORY;
            XtSetArg (args[n], XmNexportTargets, exportList); n++;
            XtSetArg (args[n], XmNnumExportTargets, 1);       n++;
        }
        else {
            exportList[0] = FILE_CONTENTS;
            exportList[1] = FILE_NAME;
            XtSetArg (args[n], XmNexportTargets, exportList); n++;
            XtSetArg (args[n], XmNnumExportTargets, 2);       n++;
        }
        XtSetArg (args[n], XmNdragOperations, XmDROP_COPY);   n++;
        XtSetArg (args[n], XmNconvertProc, ConvertProc);      n++;
        XtSetArg (args[n], XmNclientData, widget);            n++;

        /* start the drag and register a callback to clean up when done */
        dc = XmDragStart (widget, event, args, n);
        XtAddCallback (dc, XmNdragDropFinishCallback, DragDropFinish, NULL);
    }

    /* ConvertProc() -- convert the file data to the format requested
    ** by the drop site.
    */
    Boolean ConvertProc ( Widget           widget,
                          Atom             *selection,
                          Atom             *target,
                          Atom             *type_return,
                          XtPointer        *value_return,
                          unsigned long    *length_return,
                          int              *format_return)
    {
        Display          *dpy;
        Atom             FILE_CONTENTS, FILE_NAME, MOTIF_DROP;
        XtPointer        ptr;
        Widget           label;
        int              i;
        char             *text;
        struct stat      s_buf;
```

```
FILE            *fp;
long            length;
String          str;

/* intern the Atoms for data targets */
dpy = XtDisplay (widget);
FILE_CONTENTS = XInternAtom (dpy, "FILE_CONTENTS", False);
FILE_NAME = XInternAtom (dpy, "FILE_NAME", False);
MOTIF_DROP = XInternAtom (dpy, "_MOTIF_DROP", False);

/* check if we are dealing with a drop */
if (*selection != MOTIF_DROP)
    return False;

/* get the drag source widget */
XtVaGetValues (widget, XmNclientData, &ptr, NULL);
label = (Widget) ptr;
if (label == NULL)
    return False;

/* get the index into the file array from XmNuserData from the
** drag source widget.
*/
XtVaGetValues (label, XmNuserData, &ptr, NULL);
i = (int) ptr;
/* this routine processes only file contents and file name */
if (*target == FILE_CONTENTS) {
    /* get the contents of the file */
    if (stat (files[i].file_name, &s_buf) == -1 ||
                (s_buf.st_mode & S_IFMT) != S_IFREG ||
                !(fp = fopen (files[i].file_name, "r")))
        return False;
    length = s_buf.st_size;

    if (!(text = XtMalloc ((unsigned) (length + 1))))
        return False;
    else if (fread (text, sizeof (char), length, fp) != length)
        return False;
    else
        text[length] = 0;
    fclose (fp);

    /* format the value for transfer */
    *type_return = FILE_CONTENTS;
    *value_return = (XtPointer) text;
    *length_return = length;
    *format_return = 8;
    return True;
}
else if (*target == FILE_NAME) {
    str = XtNewString (files[i].file_name);
    /* format the value for transfer */
    *type_return = FILE_NAME;
    *value_return = (XtPointer) str;
```

```
            *length_return = strlen (str) + 1;
            *format_return = 8;
            return True;
        }
        else
            return False;
    }

    /* DragDropFinish() -- clean up after a drag and drop transfer. */
    void DragDropFinish (Widget widget, XtPointer client_data, XtPointer call_data)
    {
        Widget source_icon = NULL;
        XtVaGetValues (widget, XmNsourceCursorIcon, &source_icon, NULL);
        if (source_icon)
            XtDestroyWidget (source_icon);
    }
```

The output of this application is shown in Figure 22-6.

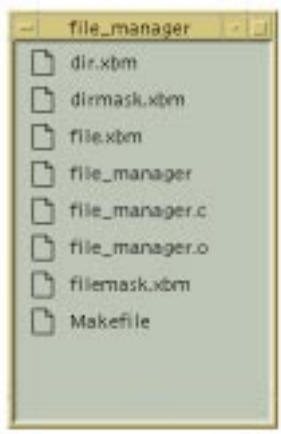


Figure 22-6 The output of the file_manager program

The application gets the names of all of the files in the current directory* and displays the filenames using Label widgets. Each file has a file or folder image next to it, depending on whether it is a regular file or a directory. The images are the drag sources for manipulating the files. If the user presses the middle mouse button over one of the symbols, the pointer changes to a drag icon and he can drag the file to another application that has a drop site that understands files.

Creating a Drag Source

When the application reads the files in the directory, it creates a global array of structures that contain information about the files. This information is used to keep track of filenames

* We use `popen()` here, but you should use `opendir()` and `readdir()`.

and file types. For each file, the application creates two Label widgets: an image that represents the type of the file and a string that specifies the filename. To link the image Labels to the array, the application passes the index of each file in the array as the XmNuserData resource for the associated Label. This value can be retrieved and used to access the information in the array.

Depending on whether a file is a regular file or a directory, the application places an image of a file or a folder next to the filename Label. Each image is created using XmGetPixmap() and specified as the XmNlabelPixmap for the appropriate image Labels. The images are also used for drag icons during the drag operation, as we describe in the next section. For more information on XmGetPixmap(), see the *Pixmaps* Section in Chapter 3.

In order to specify that the file images are drag sources, we have to establish translations for the Label widgets that are used for the images. Label widgets already have drag source functionality, so we need to decide whether to override or augment this functionality. Since the existing translation merely allows the user to drag the pixmap image for the Label, we override the translation as shown in the following code fragment:

```
static char dragTranslations[] = "#override <Btn2Down>: StartDrag()";
static XtActionsRec dragActions[] = {
    {"StartDrag", (XtActionProc) StartDrag}
};
```

The application parses the translation table and adds the action using XtParseTranslationTable() and XtAppAddActions(), respectively. The new translation table is specified for the XmNtranslations resource for each of the image Labels.

The only other operation performed in main() that is relevant for the drag functionality is the interning of atoms for target types. We use the FILE_NAME target that is defined by the ICCCM, as well as two of our own targets, FILE_CONTENTS and DIRECTORY. We chose these target names ourselves because the ICCCM does not define any targets that are suitable for our purposes. We create atoms for these targets using XInternAtom(), as shown in the following code fragment*:

```
Widget     toplevel;
Display    *dpy;
Atom       FILE_CONTENTS, FILE_NAME, DIRECTORY;

dpy = XtDisplay (toplevel);

FILE_CONTENTS = XInternAtom (dpy, "FILE_CONTENTS", False);
FILE_NAME = XInternAtom (dpy, "FILE_NAME", False);
DIRECTORY = XInternAtom (dpy, "DIRECTORY", False);
```

* As of Motif 2.0, XmInternAtom() is marked for deprecation.

Although we don't actually use the atoms in `main()`, we intern them so that they are cached by the Motif toolkit. When we intern the atoms in other routines in the application, they are retrieved from the cache.

Starting the Drag

When the user starts a drag, the `DragStart()` action routine is called. This routine creates a custom dragicon and calls `XmDragStart()` to start the drag. To create the appropriate drag icon, we need to know whether the drag source represents a file or a directory, so we fetch the `XmNuserData` from the Label widget that is the drag source. We use this value to access the appropriate structure in the `files` array and determine the type of file the user is manipulating.

Once we know what type of file we are dealing with, we can create the source icon for the drag. We use the same pixmap as for the image Label, so the drag icon is either a file image or a folder image. We use `XmGetPixmapByDepth()` to create both the icon and an iconmask so that we can specify a depth of `1`. We call `XmCreateDragIcon()` to create the drag icon, as shown in the following code fragment from Example 22-1:

```
n = 0;
XtSetArg (args[n], XmNpixmap, icon);   n++;
XtSetArg (args[n], XmNmask, iconmask); n++;
drag_icon = XmCreateDragIcon (widget, "drag_icon", args, n);
```

The DragIcon is created as a child of the drag source widget. We only need to specify the `XmNpixmap` and `XmNmask` resources because the DragIcon sets its other attributes, such as width and height, based on the pixmap. The DragIcon takes its foreground and background colors from its parent, so we don't need to specify these resources either. The `XmNmask` resource must be set to a pixmap of depth `1`, while the `XmNpixmap` can be any depth.

Now that we have a DragIcon object for the source icon, we can call `XmDragStart()` to start the drag as shown below:

```
n = 0;
XtSetArg (args[n], XmNblendModel, XmBLEND_ALL);     n++;
XtSetArg (args[n], XmNcursorBackground, bg);        n++;
XtSetArg (args[n], XmNcursorForeground, fg);        n++;
XtSetArg (args[n], XmNsourceCursorIcon, drag_icon); n++;

if (files[i].directory) {
    exportList[0] = DIRECTORY;
    XtSetArg (args[n], XmNexportTargets, exportList); n++;
    XtSetArg (args[n], XmNnumExportTargets, 1);       n++;
} else {
    exportList[0] = FILE_CONTENTS;
    exportList[1] = FILE_NAME;
    XtSetArg (args[n], XmNexportTargets, exportList); n++;
    XtSetArg (args[n], XmNnumExportTargets, 2);       n++;
```

```
    }

    XtSetArg (args[n], XmNdragOperations, XmDROP_COPY); n++;
    XtSetArg (args[n], XmNconvertProc, ConvertProc);    n++;
    XtSetArg (args[n], XmNclientData, widget);          n++;
    dc = XmDragStart (widget, event, args, n);
```

This routine creates a DragContext object for the drag and drop transfer and sets a number of resources for the DragContext. The XmNsourceCursorIcon specifies the source drag icon that we just created. We also specify the background and foreground colors for the icon. The DragContext also has the resources XmNoperationCursorIcon and XmNstateCursorIcon for specifying the operation and state icons, but our drag icon does not use these parts, so we don't set the resources.

The XmNblendModel resource controls the components of the drag icon that are used during the drag. This resource can take one of the following values:

```
XmBLEND_ALL                 XmBLEND_STATE_SOURCE
XmBLEND_JUST_SOURCE         XmBLEND_NONE
```

XmBLEND_ALL indicates that all three parts of the drag icon should be used, while XmBLEND_STATE_SOURCE causes only the state and source icons to be used. We specify the value XmBLEND_ALL since we want our source icon to be used for the drag icon, as well as the default system operation and status icons. XmBLEND_NONE means that the DragContext does not generate a drag icon.

Another important set of resources are the XmNexportTargets and XmNnumExportTargets resources. These resources specify the data targets to which the drag source can convert the actual data. The XmNexportTargets resource contains a list of target atoms.If the file is a directory, we specify the DIRECTORY target. Otherwise, we specify both the FILE_CONTENTS and FILE_NAME targets, which means that the drag source can provide both a filename and the actual contents of the file to a drop site. In order for the drag to succeed, another application must use at least one of these targets for a drop site so that the user has some place to drop the data.

The XmNdragOperations resource specifies all of the operations that are supported by the application. This value is specified as a bit mask formed by combining the following possible values:

```
XmDROP_COPY        XmDROP_MOVE        XmDROP_LINK        XmDROP_NOOP
```

For the limited purpose of this application, we specify XmDROP_COPY because we only allow the user to copy the contents of a file. A fully-functional file manager application would probably also support moving and copying files within the directory structure, but that functionality is beyond the scope of our discussion. During the drag, the operations supported by the current drop site are matched against those supported by the drag source to see if the transfer is possible.

The final DragContext resource that we specify is the XmNconvertProc. This resource indicates the procedure that is called to convert the actual data into the format requested by the drop site when the drop occurs. We specify the ConvertProc() routine for our application; this routine is described in the next section. We also set XmNclientData to the Label widget that started the drag, so that we have access to the filename and file type data stored about that Label, as this information is needed to process the drop.

After we create the DragContext and start the drag with XmDragStart(), we register a callback routine for the XmNdragDropFinishCallback so that we can destroy the DragIcon that we created. This routine is discussed further in the *Cleaning Up* Section later in this chapter.

Converting the Data

When a drop occurs, a procedure that has been registered by the drop site is called to verify that the drop can take place. This procedure checks the status of the operation and then starts the data transfer. The receiving application requests the format that it wants to receive the data in; the receiver can even request the data in multiple formats, if they are available. For each requested data target, the initiating application's XmNconvertProc is invoked. In our case, this is the ConvertProc() routine. Since we are not using incremental transfer, this routine is of type XtConvertSelectionProc, which takes the following form:

```
typedef Boolean (*XtConvertSelectionProc)( Widget         widget,
                                           Atom           *selection,
                                           Atom           *target,
                                           Atom           *type_return,
                                           XtPointer      *value_return,
                                           unsigned long  *length_return,
                                           int            *format_return)
```

The *widget* parameter is the DragContext for the drag operation, *selection* is the selection atom, which in this case is _MOTIF_DROP, and target is the type of information requested about the selection. The *type_return*, *value_return*, *length_return*, and *format_return* parameters return the type, value, length, and format of the converted data. The routine should return True if the conversion succeeds and False otherwise. For more information about this procedure type, see Volume 4, *X Toolkit Intrinsics Programming Manual*, and the appropriate reference page in Volume 5, *X Toolkit Intrinsics Reference Manual*.

The ConvertProc() routine in Example 22-1 starts by retrieving the Label widget from the XmNclientData resource of the DragContext. The goal is to get an index into the files array so that we can access information about the file. The index is stored in the XmNuserData resource of the Label widget. Once we have the index, we can use it to get the filename from the array.

Our conversion routine only handles requests for a filename or the contents of a file. If `target` is set to FILE_CONTENTS, `ConvertProc()` retrieves the contents of the file and formats the data for transfer back to the receiving client. The contents of the file are passed as a pointer to the text, using the `value_return` parameter. If the drop site has requested the FILE_NAME target, the routine returns the filename in `value_return`. In either case, the `length_return` argument is set to the length of the text, and `format_return` is set to `8` to specify the length of each of the elements in `value_return`. The `return_type` parameter is set to the appropriate target. If the drop site has requested any other target, the routine returns `False` to indicate that the transfer has failed.

The conversion routine does not handle the DIRECTORY target, partly because we have not implemented any drop sites that understand the target. A real file manager application would want to support the dragging of directories to allow the user to modify the file system using drag and drop. In this case, the conversion procedure would need to have another branch for handling the DIRECTORY target.

Since the drag source only supports the copy operation, the conversion routine does not have to worry about deleting the existing data. With a copy operation, the `XmNconvertProc` returns a pointer to the data so that when the operation is done, both the initiator and the receiver have a copy of the data. With a move operation, the initiating application returns a pointer to the data and then waits for the receiver to tell it to delete the data. The receiving application gets the data, stores it, and then specifies the DELETE target to handle this situation. When the initiating client gets this target, it can safely delete the data. With a link operation, the initiator again passes a pointer to the data, but in this case the receiver uses the pointer to establish a link to the data.

Modifying an Existing Drag Source

In *file_manager.c*, we decided to replace the existing drag capabilities of the image Label widgets and provide our own functionality instead. By default, the Labels would function as graphical drag sources, but since there are no drop sites that support graphical data, there is no reason to preserve this functionality.

However, if you want to provide the default functionality for a drag source as well as your own functionality, the set up of the drag source becomes more complicated. Each Motif widget that acts as a drag source has a translation and action that starts the drag. Since the existing action calls `XmDragStart()` for the transfer, another action routine cannot call `XmDragStart()` again. The solution to this problem is to write an action routine that retrieves the DragContext for the transfer and modifies its resources.

In our application, we want to augment the drag source functionality of the filename Labels. If the user drags the Label to a drop site that understands file objects, the actual file is transferred. Otherwise, the default drag functionality for the Label causes the text of the Label to be passed to the dropsite. The first thing that we need to do is modify the

translations for the Label widgets. Since we want to provide the default functionality, the new translation calls the widget's existing drag action routine followed by our own action. The existing drag action routine for the Label widget is `ProcessDrag()`, so the translations and actions for the application can be defined as follows:

```
static char dragTranslations[] = "#override <Btn2Down>: StartDrag()";
static char newdragTranslations[] =
                    "#override <Btn2Down>: ProcessDrag() UpdateDrag()";
static XtActionsRec dragActions[] = {
    {"StartDrag", (XtActionProc) StartDrag},
    {"UpdateDrag", (XtActionProc) UpdateDrag}
};
```

As always, the translations need to be parsed using `XtParseTranslationTable()`, and the actions need to be registered using `XtAppAddActions()`. Now, when we create each of the filename Labels, we can specify the new translation for the `XmNtranslations` resource, as shown in the following code fragment:

```
parsed_trans_text = XtParseTranslationTable (newdragTranslations);
...
n = 0;
XtSetArg (args[n], XmNtranslations, parsed_trans_text); n++;
XtSetArg (args[n], XmNuserData, i); n++;
...
label = XmCreateLabel (form, files[i].file_name, args, n);
```

Note that we also specify the index in the `files` array as the `XmNuserData` for these widgets, just as we did for the image Labels in Example 22-1.

The `UpdateDrag()` action routine is invoked after the Label's default drag action, which means that `XmDragStart()` has already been called for the operation. Our action routine retrieves the DragContext for the operation and modifies it, as shown in Example 22-2*.

Example 22-2 The UpdateDrag() routine

```
void (*convert_proc) (Widget, Atom *, Atom *, Atom *,
                          XtPointer *, unsigned long *, int *);

void UpdateDrag (Widget widget, XEvent *event, String *params,
                     Cardinal *num_params)
{
    Arg              args[10];
    int              n, m, i;
    Display          *dpy;
    Atom             FILE_CONTENTS, FILE_NAME, DIRECTORY;
    Widget           drag_icon, dc;
    Pixel            fg, bg;
    Pixmap           icon, iconmask;
    XtPointer        ptr;
    Boolean          NewConvertProc(Widget, Atom *, Atom *, Atom *,
```

* `XmInternAtom()` is marked for deprecation from Motif 2.0.

```
                                XtPointer *, unsigned long *, int *);
void            DragDropFinish(Widget, XtPointer, XtPointer);
Cardinal        numExportTargets;
Atom            *exportTargets, *newTargets;

/* intern the Atoms for data targets */
dpy = XtDisplay (widget);
FILE_CONTENTS = XInternAtom (dpy, "FILE_CONTENTS", False);
FILE_NAME = XInternAtom (dpy, "FILE_NAME", False);
DIRECTORY = XInternAtom (dpy, "DIRECTORY", False);

/* get background and foreground colors and fetch index into file
** array from XmNuserData.
*/
XtVaGetValues (widget, XmNforeground, &fg, XmNbackground, &bg, XmNuserData,
                    &ptr, NULL);

/* create pixmaps for drag icon -- either file or directory */
i = (int) ptr;
if (files[i].is_directory) {
    icon = XmGetPixmapByDepth (XtScreen (widget), "dir.xbm", 1, 0, 1);
    iconmask = XmGetPixmapByDepth (XtScreen (widget), "dirmask.xbm",
                                                        1, 0, 1);
}
else {
    icon = XmGetPixmapByDepth (XtScreen (widget), "file.xbm", 1, 0, 1);
    iconmask = XmGetPixmapByDepth (XtScreen (widget), "filemask.xbm",
                                                        1, 0, 1);
}
if (icon == XmUNSPECIFIED_PIXMAP || iconmask == XmUNSPECIFIED_PIXMAP) {
    puts ("Couldn't load pixmaps");
    exit (1);
}
n = 0;
XtSetArg (args[n], XmNpixmap, icon);   n++;
XtSetArg (args[n], XmNmask, iconmask); n++;
drag_icon = XmCreateDragIcon (widget, "drag_icon", args, n);

/* get the DragContext and retrieve info about it */
dc = XmGetDragContext (widget, event->xbutton.time);
n = 0;
XtSetArg (args[n], XmNexportTargets, &exportTargets);       n++;
XtSetArg (args[n], XmNnumExportTargets, &numExportTargets); n++;
XtSetArg (args[n], XmNconvertProc, &convert_proc);          n++;
XtGetValues (dc, args, n);

/* add new targets to the list of targets */
n = 0;
if (files[i].is_directory) {
    newTargets = (Atom *) XtMalloc (sizeof (Atom) * (numExportTargets + 1));
    for (m = 0; m < numExportTargets; m++)
        newTargets[m] = exportTargets[m];
    newTargets[m] = DIRECTORY;
    XtSetArg (args[n], XmNexportTargets, newTargets); n++;
```

```
            XtSetArg (args[n], XmNnumExportTargets, numExportTargets + 1); n++;
        } else {
            newTargets = (Atom *) XtMalloc (sizeof (Atom) * (numExportTargets + 2));
            for (m = 0; m < numExportTargets; m++)
                newTargets[m] = exportTargets[m];
            newTargets[m] = FILE_CONTENTS;
            newTargets[m+1] = FILE_NAME;
            XtSetArg (args[n], XmNexportTargets, newTargets);              n++;
            XtSetArg (args[n], XmNnumExportTargets, numExportTargets + 2); n++;
        }

        /* modify other DragContext resources */
        XtSetArg (args[n], XmNblendModel, XmBLEND_ALL);     n++;
        XtSetArg (args[n], XmNcursorBackground, bg);        n++;
        XtSetArg (args[n], XmNcursorForeground, fg);        n++;
        XtSetArg (args[n], XmNsourceCursorIcon, drag_icon); n++;
        XtSetArg (args[n], XmNdragOperations, XmDROP_COPY); n++;
        XtSetArg (args[n], XmNconvertProc, NewConvertProc); n++;
        XtSetArg (args[n], XmNclientData, widget);          n++;
        XtSetValues (dc, args, n);
        XtAddCallback (dc, XmNdragDropFinishCallback, DragDropFinish, NULL);
    }
```

This routine performs many of the same tasks as the `StartDrag()` action routine, such as accessing the appropriate structure in the files array and creating a DragIcon for the source icon. The main difference is that we use `XmGetDragContext()` to retrieve the current DragContext object, rather than creating one using `XmDragStart()`.

The routine retrieves the values of the `XmNexportTargets`, `XmNnumExportTargets`, and `XmNconvertProc` resources using `XtGetValues()` so that it can preserve the existing functionality. The appropriate new targets are added to the list of targets based on the type of the file, and `XmNexportTargets` is set to the new list. The `NewConvertProc()` routine is used for the `XmNconvertProc`. The rest of the DragContext resources are specified as in `StartDrag()`, and the DragContext is modified using `XtSetValues()`.

There is only one difference between the `NewConvertProc()` routine and `ConvertProc()` in *file_manager.c*. Instead of simply returning `False` if the requested target is not FILE_CONTENTS or FILE_NAME, `NewConvertProc()` calls the conversion procedure retrieved from the Label widget, as shown in the following fragment:

```
(*convert_proc) ( widget, selection, target, type_return, value_return,
                  length_return, format_return);
```

Essentially, our conversion routine handles our data targets and passes other targets to the Label widget's default conversion procedure.

Providing Custom Drag-over Visuals

The DragContext has a number of callback routines that the initiating application can use to provide custom drag-over visuals. These callbacks are invoked when different events occur during the drag, like when the drag icon enters or leaves a drop site. The DragContext provides the following callback routines for monitoring the drag:

```
XmNdragMotionCallback            XmNdropSiteEnterCallback
XmNdropSiteLeaveCallback         XmNoperationChangedCallback
XmNtopLevelEnterCallback         XmNtopLevelLeaveCallback
```

The names of the routines are fairly self-explanatory. Each callback has its own special callback structure that contains the relevant information about the current state of the drag operation. For example, the `XmNdropSiteEnterCallback` uses a callback structure of type `XmDropSiteEnterCallbackStruct`, which is defined as follows:

```
typedef struct {
    int             reason;
    XEvent          *event;
    Time            timeStamp;
    unsigned char   operation;
    unsigned char   operations;
    unsigned char   dropSiteStatus;
    Position        x;
    Position        y;
} XmDropSiteEnterCallbackStruct, *XmDropSiteEnterCallback;
```

The `reason` field in this structure is always `XmCR_DROP_SITE_ENTER`. The `operation` and `operations` fields specify the current operation and the set of supported operations, respectively. The `dropSiteStatus` element indicates whether or not the current drop site is valid, based on the targets supported by the drag source and the drop site. This field can have one of the following values:

```
XmDROP_SITE_VALID      XmDROP_SITE_INVALID      XmNO_DROP_SITE
```

The `operation`, `operations`, and `dropSiteStatus` fields are initialized by the toolkit based on the values of different resources for both the drag source and the drop site. If the drop site has registered an `XmNdragProc` and the dynamic protocol is being used, this routine can update these fields as necessary before the data is passed to the callback routine. A drop site might want to update these fields if it is performing any special processing or simulating multiple drop sites.

All of the callback structures for the DragContext callback routines have a `reason` field that indicates why the callback was invoked. The callback structures also provide information that is relevant to the particular routine; they are all similar to the `XmDropSiteEnterCallbackStruct`. See the DragContext reference page in Volume 6 B, *Motif Reference Manual*, for complete information about the different callback structures.

When an application creates the DragContext for a drag, it can register routines for the different callback resources. These routines can perform any special processing that is necessary, as well as handle custom drag-over effects for the transfer. The typical way to handle drag-over effects is to modify the various drag icon resources of the DragContext during the drag. The XmNsourcePixmapIcon, XmNsourceCursorIcon, XmNoperationCursorIcon, and XmNstateCursorIcon resources specify the different components of the drag icon. The XmNvalidCursorForeground, XmNinvalidCursorForeground, and XmNnoneCursorForeground resources of the DragContext can be used to further distinguish between the different states during a drag.

The XmNsourcePixmapIcon is used under the preregister protocol and can be any size, while the XmNsourceCursorIcon is used for the dynamic protocol and is limited to the size of the largest cursor for a particular platform. If you want to specify a color icon, you must use the XmNsourcePixmapIcon resource. If XmNsourcePixmapIcon is not specified, the value of XmNsourceCursorIcon is used. If this resource has not been specified, the default source icon for the Screen object is used.

At any point during a drag, the initiating client can call XmDragCancel() to cancel the transfer. The user can also cancel the operation by pressing the ESCAPE key.The initiating client can retrieve additional information about the current drop site by calling XmDropSiteRetrieve() during the drag.

After the user drops the data in a drop site, the drag source has one last chance to check the status of the transfer and provide custom visual effects. After the receiving client's XmNdropProc completes, the DragContext's XmNdropStartCallback is invoked. This routine has a callback structure of type XmDropStartCallbackStruct, which is defined as follows:

```
typedef struct {
    int              reason;
    XEvent           *event;
    Time             timeStamp;
    unsigned char    operation;
    unsigned char    operations;
    unsigned char    dropSiteStatus;
    unsigned char    dropAction;
    Position         x;
    Position         y;
} XmDropStartCallbackStruct, *XmDropStartCallback;
```

The reason field is set to XmCR_DROP_START, while the operation, operations, and dropSiteStatus fields are set as described previously. The dropAction field is set to XmDROP if the user has simply dropped the data, XmDROP_HELP if the user has requested help on the drop site, or XmDROP_CANCEL if the user has cancelled the transfer.

Cleaning Up

The initiating client can also register callbacks that are invoked after a drag and drop transfer has completed. The XmNdropFinishCallback is called after the receiver's XmNtransferProc has finished processing all of the data targets requested by the receiver. This routine receives as call_data a callback structure of the type XmDropFinishCallbackStruct, where the reason field is XmCR_DROP_FINISH.

The XmNdragDropFinishCallback is invoked when the entire operation has completed, which is immediately after theXmNdropFinishCallback.In this case, the callback structure is an XmDragDropFinishCallbackStructure, and reason is XmCR_DRAG_DROP_FINISH. Our application uses this callback to destroy the drag icon that we created, as shown below:

```
void DragDropFinish (Widget widget, XtPointer client_data, XtPointer call_data)
{
    Widget source_icon = NULL;
    XtVaGetValues (widget, XmNsourceCursorIcon, &source_icon, NULL);
    if (source_icon)
        XtDestroyWidget (source_icon);
}
```

The widget passed to the callback routine is the DragContext object for the drag and drop transfer. The routine retrieves the source icon from the DragContext and destroys it using XtDestroyWidget().

Working With Drop Sites

In order to handle data from drag sources that provide something other than textual data, an application has to register drop sites that understand other types of data. To make the *file_manager.c* application useful, we need an application that has drop sites that can handle file objects. In this section, we are going to modify the text editor from Chapter 18, *Text Widgets*, so that it understands file data. The application contains two drop sites that handle files: the main text entry area and a filename status area. The Example 22-3 shows the main(), HandleDropLabel(), HandleDropText(), and TransferProc() routines for *editor_dnd.c*. The rest of the routines in the application are the same as in the *A Text Editor* Section in Chapter 18, so we have not shown them here.*

Example 22-3 The editor_dnd.c program

```
/* editor_dnd.c -- create an editor application that contains drop sites
** that understand file data. A file can be dragged from another
** application and dropped in the text entry area or the filename status
** area.
*/
```

* XtVaAppInitialize() is considered deprecated in X11R6. XmInternAtom() is marked for deprecation from Motif 2.0.

```
#include <Xm/Text.h>
#include <Xm/TextF.h>
#include <Xm/LabelG.h>
#include <Xm/PushBG.h>
#include <Xm/RowColumn.h>
#include <Xm/MainW.h>
#include <Xm/Form.h>
#include <Xm/FileSB.h>
#include <Xm/SeparatoG.h>
#include <Xm/DragDrop.h>
#include <X11/Xos.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>

#define FILE_OPEN 0
#define FILE_SAVE 1
#define FILE_EXIT 2
#define EDIT_CUT 0
#define EDIT_COPY 1
#define EDIT_PASTE 2
#define EDIT_CLEAR 3
#define SEARCH_FIND_NEXT 0
#define SEARCH_SHOW_ALL 1
#define SEARCH_REPLACE 2
#define SEARCH_CLEAR 3

/* global variables */
void (*drop_proc) (Widget, XtPointer, XtPointer);
Widget text_edit, search_text, replace_text, text_output;
Widget toplevel, file_label;

main (int argc, char *argv[])
{
    XtAppContext   app_context;
    Display        *dpy;
    Atom           FILE_CONTENTS, FILE_NAME;
    Widget         main_window, menubar, form, search_panel, label;
    Widget         sep1, sep2;
    void           file_cb(Widget, XtPointer, XtPointer);
    void           edit_cb(Widget, XtPointer, XtPointer);
    void           search_cb(Widget, XtPointer, XtPointer);
    Arg            args[10];
    int            n = 0;
    XmString       open, save, exit, exit_acc, file, edit, cut, clear,
                   copy, paste, search, next, find, replace;
    Cardinal       numImportTargets;
    Atom           *importTargets, *newTargets;
    Atom           importList[2];
    void           HandleDropLabel(Widget, XtPointer, XtPointer);
    void           HandleDropText(Widget, XtPointer, XtPointer);

    XtSetLanguageProc (NULL, NULL, NULL);
```

```
toplevel = XtVaOpenApplication (&app_context, "Demos", NULL, 0, &argc,
                                argv, NULL, sessionShellWidgetClass, NULL);
dpy = XtDisplay (toplevel);
FILE_CONTENTS = XInternAtom (dpy, "FILE_CONTENTS", False);
FILE_NAME = XInternAtom (dpy, "FILE_NAME", False);
main_window = XmCreateMainWindow (toplevel, "main_window", NULL, 0);

/* Create a simple MenuBar that contains three menus */
file = XmStringCreateLocalized ("File");
edit = XmStringCreateLocalized ("Edit");
search = XmStringCreateLocalized ("Search");
menubar = XmVaCreateSimpleMenuBar (main_window, "menubar",
                                   XmVaCASCADEBUTTON, file, 'F',
                                   XmVaCASCADEBUTTON, edit, 'E',
                                   XmVaCASCADEBUTTON, search, 'S',
                                   NULL);
XmStringFree (file);
XmStringFree (edit);
XmStringFree (search);

/* First menu is the File menu -- callback is file_cb() */
open = XmStringCreateLocalized ("Open...");
save = XmStringCreateLocalized ("Save...");
exit = XmStringCreateLocalized ("Exit");
exit_acc = XmStringCreateLocalized ("Ctrl+C");
XmVaCreateSimplePulldownMenu (menubar, "file_menu", 0, file_cb,
                        XmVaPUSHBUTTON, open, 'O', NULL, NULL,
                        XmVaPUSHBUTTON, save, 'S', NULL, NULL,
                        XmVaSEPARATOR,
                        XmVaPUSHBUTTON, exit, 'x', "Ctrl<Key>c", exit_acc,
                        NULL);
XmStringFree (open);
XmStringFree (save);
XmStringFree (exit);
XmStringFree (exit_acc);

/* ...create the "Edit" menu -- callback is edit_cb() */
cut = XmStringCreateLocalized ("Cut");
copy = XmStringCreateLocalized ("Copy");
clear = XmStringCreateLocalized ("Clear");
paste = XmStringCreateLocalized ("Paste");
XmVaCreateSimplePulldownMenu (menubar, "edit_menu", 1, edit_cb,
                            XmVaPUSHBUTTON, cut, 't', NULL, NULL,
                            XmVaPUSHBUTTON, copy, 'C', NULL, NULL,
                            XmVaPUSHBUTTON, paste, 'P', NULL, NULL,
                            XmVaSEPARATOR,
                            XmVaPUSHBUTTON, clear, 'l', NULL, NULL,
                            NULL);
XmStringFree (cut);
XmStringFree (copy);
XmStringFree (paste);

/* create the "Search" menu -- callback is search_cb() */
next = XmStringCreateLocalized ("Find Next");
find = XmStringCreateLocalized ("Show All");
```

```
replace = XmStringCreateLocalized ("Replace Text");
XmVaCreateSimplePulldownMenu (menubar, "search_menu", 2, search_cb,
                              XmVaPUSHBUTTON, next, 'N', NULL, NULL,
                              XmVaPUSHBUTTON, find, 'A', NULL, NULL,
                              XmVaPUSHBUTTON, replace, 'R', NULL, NULL,
                              XmVaSEPARATOR,
                              XmVaPUSHBUTTON, clear, 'C', NULL, NULL,
                              NULL);
XmStringFree (next);
XmStringFree (find);
XmStringFree (replace);
XmStringFree (clear);
XtManageChild (menubar);

/* create a form work are */
form = XmCreateForm (main_window, "form", NULL, 0);

/* create horizontal RowColumn inside the form */
n = 0;
XtSetArg (args[n], XmNorientation, XmHORIZONTAL);      n++;
XtSetArg (args[n], XmNpacking, XmPACK_TIGHT);          n++;
XtSetArg (args[n], XmNtopAttachment, XmATTACH_FORM);   n++;
XtSetArg (args[n], XmNleftAttachment, XmATTACH_FORM);  n++;
XtSetArg (args[n], XmNrightAttachment, XmATTACH_FORM); n++;
search_panel = XmCreateRowColumn (form, "search_panel", args, n);

/* Create two TextField widgets with Labels... */
label = XmCreateLabelGadget (search_panel, "Search Pattern", NULL, 0);
XtManageChild (label);

search_text = XmCreateTextField (search_panel, "search_text", NULL, 0);
XtManageChild (search_text);

label = XmCreateLabelGadget (search_panel, "Replace Pattern", NULL, 0);
XtManageChild (label);

replace_text = XmCreateTextField (search_panel, "replace_text", NULL, 0);
XtManageChild (replace_text);
XtManageChild (search_panel);

n = 0;
XtSetArg (args[n], XmNeditable, False);                 n++;
XtSetArg (args[n], XmNcursorPositionVisible, False);    n++;
XtSetArg (args[n], XmNshadowThickness, 0);              n++;
XtSetArg (args[n], XmNleftAttachment, XmATTACH_FORM);   n++;
XtSetArg (args[n], XmNrightAttachment, XmATTACH_FORM);  n++;
XtSetArg (args[n], XmNbottomAttachment, XmATTACH_FORM); n++;
text_output = XmCreateTextField (form, "text_output", args, n);
XtManageChild (text_output);

n = 0;
XtSetArg (args[n], XmNleftAttachment, XmATTACH_FORM);     n++;
XtSetArg (args[n], XmNrightAttachment, XmATTACH_FORM);    n++;
XtSetArg (args[n], XmNbottomAttachment, XmATTACH_WIDGET); n++;
XtSetArg (args[n], XmNbottomWidget, text_output);         n++;
```

```
sep2 = XmCreateSeparatorGadget (form, "sep2", args, n);
XtManageChild (sep2);

/* create file status area */
n = 0;
XtSetArg (args[n], XmNalignment, XmALIGNMENT_BEGINNING);  n++;
XtSetArg (args[n], XmNleftAttachment, XmATTACH_FORM);     n++;
XtSetArg (args[n], XmNrightAttachment, XmATTACH_FORM);    n++;
XtSetArg (args[n], XmNbottomAttachment, XmATTACH_WIDGET); n++;
XtSetArg (args[n], XmNbottomWidget, sep2);                n++;
file_label = XmCreateLabelGadget (form, "Filename:", args, n);
XtManageChild (file_label);

/* register the file status label as a drop site */
n = 0;
importList[0] = FILE_CONTENTS;
importList[1] = FILE_NAME;
XtSetArg (args[n], XmNimportTargets, importList);               n++;
XtSetArg (args[n], XmNnumImportTargets, XtNumber (importList)); n++;
XtSetArg (args[n], XmNdropSiteOperations, XmDROP_COPY);         n++;
XtSetArg (args[n], XmNdropProc, HandleDropLabel);               n++;
XmDropSiteRegister (file_label, args, n);

n = 0;
XtSetArg (args[n], XmNleftAttachment, XmATTACH_FORM);     n++;
XtSetArg (args[n], XmNrightAttachment, XmATTACH_FORM);    n++;
XtSetArg (args[n], XmNbottomAttachment, XmATTACH_WIDGET); n++;
XtSetArg (args[n], XmNbottomWidget, file_label);          n++;
sep1 = XmCreateSeparatorGadget (form, "sep1", args, n);
XtManageChild (sep1);

/* create text entry area */
n = 0;
XtSetArg (args[n], XmNrows, 10);                          n++;
XtSetArg (args[n], XmNcolumns, 80);                       n++;
XtSetArg (args[n], XmNeditMode, XmMULTI_LINE_EDIT);       n++;
XtSetArg (args[n], XmNtopAttachment, XmATTACH_WIDGET);    n++;
XtSetArg (args[n], XmNtopWidget, search_panel);           n++;
XtSetArg (args[n], XmNleftAttachment, XmATTACH_FORM);     n++;
XtSetArg (args[n], XmNrightAttachment, XmATTACH_FORM);    n++;
XtSetArg (args[n], XmNbottomAttachment, XmATTACH_WIDGET); n++;
XtSetArg (args[n], XmNbottomWidget,     sep1);            n++;
text_edit = XmCreateScrolledText (form, "text_edit", args, n);
XtManageChild (text_edit);

/* retrieve drop site info so that we can modify it */
n = 0;
XtSetArg (args[n], XmNimportTargets, &importTargets);       n++;
XtSetArg (args[n], XmNnumImportTargets, &numImportTargets); n++;
XtSetArg (args[n], XmNdropProc, &drop_proc);                n++;
XmDropSiteRetrieve (text_edit, args, n);

/* add FILE_CONTENTS and FILE_NAME to the list of targets */
newTargets = (Atom *) XtMalloc (sizeof (Atom) * (numImportTargets + 2));
```

```
    for (n = 0; n < numImportTargets; n++)
        newTargets[n] = importTargets[n];
    newTargets[n] = FILE_CONTENTS;
    newTargets[n+1] = FILE_NAME;

    /* update the drop site */
    n = 0;
    XtSetArg (args[n], XmNimportTargets, newTargets); n++;
    XtSetArg (args[n], XmNnumImportTargets, numImportTargets+2); n++;
    XtSetArg (args[n], XmNdropProc, HandleDropText); n++;
    XmDropSiteUpdate (text_edit, args, n);

    XtManageChild (form);
    XtManageChild (main_window);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app_context);
}

/* HandleDropLabel() -- start the data transfer when data is dropped in
** the filename status area.
*/
void HandleDropLabel (Widget widget, XtPointer client_data,
                      XtPointer call_data)
{
    Display                 *dpy;
    Atom                    FILE_CONTENTS, FILE_NAME;
    XmDropProcCallback      DropData;
    XmDropTransferEntryRec  transferEntries[2];
    XmDropTransferEntry     transferList;
    Arg                     args[10];
    int                     n, i;
    Widget                  dc;
    Cardinal                numExportTargets;
    Atom                    *exportTargets;
    Boolean                 file_name = False;
    void                    TransferProc(Widget, XtPointer, Atom *, Atom *,
                                         XtPointer, unsigned long *, int);


    /* intern the Atoms for data targets */
    dpy = XtDisplay (toplevel);
    FILE_CONTENTS = XInternAtom (dpy, "FILE_CONTENTS", False);
    FILE_NAME = XInternAtom (dpy, "FILE_NAME", False);
    DropData = (XmDropProcCallback) call_data;
    dc = DropData->dragContext;

    /* retrieve the data targets and search for FILE_NAME */
    n = 0;
    XtSetArg (args[n], XmNexportTargets, &exportTargets);       n++;
    XtSetArg (args[n], XmNnumExportTargets, &numExportTargets); n++;
    XtGetValues (dc, args, n);
    for (i = 0; i < numExportTargets; i++) {
        if (exportTargets[i] == FILE_NAME) {
            file_name = True;
            break;
```

```
        }
    }

    /* make sure we have a drop that is a copy operation and one of
    ** the targets is FILE_NAME. if not, set the status to failure.
    */
    n = 0;
    if ((!file_name) || (DropData->dropAction != XmDROP) ||
                        (DropData->operation != XmDROP_COPY)) {
        XtSetArg (args[n], XmNtransferStatus, XmTRANSFER_FAILURE); n++;
        XtSetArg (args[n], XmNnumDropTransfers, 0);                n++;
    }
    else {
        /* set up transfer requests for drop site */
        transferEntries[0].target = FILE_CONTENTS;
        transferEntries[0].client_data = (XtPointer) text_edit;
        transferEntries[1].target = FILE_NAME;
        transferEntries[1].client_data = (XtPointer) file_label;
        transferList = transferEntries;
        XtSetArg (args[n], XmNdropTransfers, transferEntries); n++;
        XtSetArg (args[n], XmNnumDropTransfers,
                                XtNumber (transferEntries));  n++;
        XtSetArg (args[n], XmNtransferProc, TransferProc);     n++;
    }
    XmDropTransferStart (dc, args, n);
}

/* HandleDropText() -- start the data transfer when data is dropped in
** the text entry area.
*/
void HandleDropText (Widget widget, XtPointer client_data, XtPointer call_data)
{
    Display               *dpy;
    Atom                  FILE_CONTENTS, FILE_NAME;
    XmDropProcCallback    DropData;
    XmDropTransferEntryRec transferEntries[2];
    XmDropTransferEntry   transferList;
    Arg                   args[10];
    int                   n, i;
    Widget                dc;
    Cardinal              numExportTargets;
    Atom                  *exportTargets;
    Boolean               file_contents = False;
    void                  TransferProc(Widget, XtPointer, Atom *, Atom *,
                                       XtPointer, unsigned long *, int);

    /* intern the Atoms for data targets */
    dpy = XtDisplay (toplevel);
    FILE_CONTENTS = XInternAtom (dpy, "FILE_CONTENTS", False);
    FILE_NAME = XInternAtom (dpy, "FILE_NAME", False);
    DropData = (XmDropProcCallback) call_data;
    dc = DropData->dragContext;

    /* retrieve the data targets and search for FILE_CONTENTS */
```

```
    n = 0;
    XtSetArg (args[n], XmNexportTargets, &exportTargets); n++;
    XtSetArg (args[n], XmNnumExportTargets, &numExportTargets); n++;
    XtGetValues (dc, args, n);
    for (i = 0; i < numExportTargets; i++) {
        if (exportTargets[i] == FILE_CONTENTS) {
            file_contents = True;
            break;
        }
    }
    if (file_contents) {
        /* make sure we have a drop that is a copy operation.
        ** if not, set the status to failure.
        */
        n = 0;
        if ((DropData->dropAction != XmDROP) ||
                                (DropData->operation != XmDROP_COPY)) {
            XtSetArg (args[n], XmNtransferStatus, XmTRANSFER_FAILURE); n++;
            XtSetArg (args[n], XmNnumDropTransfers, 0); n++;
        }
        else {
            /* set up transfer requests for drop site */
            transferEntries[0].target = FILE_CONTENTS;
            transferEntries[0].client_data = (XtPointer) text_edit;
            transferEntries[1].target = FILE_NAME;
            transferEntries[1].client_data = (XtPointer) file_label;
            transferList = transferEntries;
            XtSetArg (args[n], XmNdropTransfers, transferEntries); n++;
            XtSetArg (args[n], XmNnumDropTransfers,
                                    XtNumber (transferEntries)); n++;
            XtSetArg (args[n], XmNtransferProc, TransferProc); n++;
        }
        XmDropTransferStart (dc, args, n);
    }
    else
        (*drop_proc) (widget, client_data, call_data);
    }

/* TransferProc() -- handle data transfer of converted data from drag
** source to drop site.
*/
void TransferProc (Widget widget, XtPointer client_data, Atom *seltype,
            Atom *type, XtPointer value, unsigned long *length, int format)
{
    Display     *dpy;
    Atom        FILE_CONTENTS, FILE_NAME;
    Widget      w;
    XmString    string;
    char        label[256];

    /* intern the Atoms for data targets */
    dpy = XtDisplay (toplevel);
    FILE_CONTENTS = XInternAtom (dpy, "FILE_CONTENTS", False);
    FILE_NAME = XInternAtom (dpy, "FILE_NAME", False);
    w = (Widget) client_data;
```

```
        if (*type == FILE_CONTENTS)
            XmTextSetString (w, value);
        else if (*type == FILE_NAME) {
            sprintf (label, "Filename: %s", value);
            string = XmStringCreateLocalized (label);
            XtVaSetValues (w, XmNlabelString, string, NULL);
            XmStringFree (string);
        }
    }
```

The application basically has the same functionality as *editor.c* in Chapter 18, *Text Widgets*. The only difference in the interface is the *Filename:* status area that displays the name of the current file. This status area is also a drop site for file objects, so the user can drag a file from the *file_manager.c* application and drop it in this area. When a file is dropped here, the filename is displayed in the status area, and the contents of the file are copied into the ScrolledText object. The ScrolledText object has also been modified to function as a drop site for file data, so the user can drop a file in the text entry area. Figure 22-7 shows the output of the application before and after a file has been dropped in the file status area.

Before

After

Figure 22-7 The output of the editor_dnd program

Creating a Drop Site

The file status area is a Label widget, so it does not have any drop site capabilities by default. In order for the widget to function as a drop site, we have to register it using XmDropSiteRegister(), as shown below:

```
n = 0;
importList[0] = FILE_CONTENTS;
importList[1] = FILE_NAME;
XtSetArg (args[n], XmNimportTargets, importList); n++;
XtSetArg (args[n], XmNnumImportTargets, XtNumber (importList)); n++;
XtSetArg (args[n], XmNdropSiteOperations, XmDROP_COPY); n++;
XtSetArg (args[n], XmNdropProc, HandleDropLabel); n++;
XmDropSiteRegister (file_label, args, n);
```

This routine registers information about the drop site in a DropSite object using resources that are specified as for a normal widget. Since drop sites are referenced by their associated widget, however, the resources cannot be set using XtVaSetValues().

The XmNimportTargets resource specifies the data targets that the drop site can handle. We use the FILE_CONTENTS and FILE_NAME targets that we have interned using XInternAtom()*. The drop site only supports copy operations, so XmNdropSiteOperations is set to XmDROP_COPY. The final resource that we specify is the XmNdropProc. This callback is invoked when a drop occurs in the drop site; it is responsible for starting the transfer of data from the drag source to the drop site. The HandleDropLabel() routine handles the drop for the file status area, as we describe in *Handling the Drop* Section later in this chapter.

Modifying an Existing Drop Site

The *editor_dnd.c* application also allows the user to drag a file from *file_manager.c* to the main text entry area and drop it. This action causes the contents of the file to be copied to the Text widget. By default, the Text widget also has its own drop site functionality that allows the user to drop textual data. We want to modify the drop site to incorporate our own functionality but still allow the user to drag and drop textual data in the widget. The Text widget has already been registered as a drop site by the Motif toolkit, so we do not need to call XmDropSiteRegister(). In fact, if we did call that routine, we would override the default functionality.

Instead, we call XmDropSiteRetrieve() to get the values of the XmNimportTargets, XmNnumImportTargets, and XmNdropProc resources for the Text widget drop site, as shown in the following fragment:

```
n = 0;
XtSetArg (args[n], XmNimportTargets, &importTargets);       n++;
```

* XmInternAtom() is marked for deprecation as of Motif 2.0.

```
XtSetArg (args[n], XmNnumImportTargets, &numImportTargets); n++;
XtSetArg (args[n], XmNdropProc, &drop_proc);                n++;
XmDropSiteRetrieve (text_edit, args, n);
```

Although a drop site is always associated with a widget, the `XtVaGetValues()` routine cannot be used to retrieve drop site resources, as the resources are stored separately from the widget in a DropSite object. We retrieve the `XmNimportTargets` resource so that we can add our own targets to the list of data targets for the drop site. A drop site can only have one `XmNdropProc` associated with it, so we need to get the existing routine and store it before we specify our own routine.

Once we have the data targets for the drop site, we create a new list that contains the existing targets, as well as the FILE_CONTENTS and FILE_NAME targets. We use `XmDropSiteUpdate()` to modify the drop site:

```
n = 0;
XtSetArg (args[n], XmNimportTargets, newTargets);              n++;
XtSetArg (args[n], XmNnumImportTargets, numImportTargets + 2); n++;
XtSetArg (args[n], XmNdropProc, HandleDropText);               n++;
XmDropSiteUpdate (text_edit, args, n);
```

The `HandleDropText()` routine processes the drops that occur in the Text widget. We explain this routine in detail in the following section.

If you need to update information for a number of drop sites, you should use the `XmDropSiteStartUpdate()` and `XmDropSiteEndUpdate()` routines, as they optimize the process. After a call to `XmDropSiteStartUpdate()`, `XmDropSiteUpdate()` can be called repeatedly for different drop sites. When you are finished updating all of the drop sites, call `XmDropSiteEndUpdate()`.

Handling the Drop

When a drop occurs, the receiving application takes over and the `XmNdropProc` for the drop site is called. This callback provides a callback structure of type `XmDropProcCallbackStruct`, which is defined as follows:

```
typedef struct {
    int              reason;
    XEvent           *event;
    Time             timeStamp;
    Widget           dragContext;
    Position         x;
    Position         y;
    unsigned char    dropSiteStatus;
    unsigned char    operation;
    unsigned char    operations;
    unsigned char    dropAction;
} XmDropProcCallbackStruct, *XmDropProcCallback;
```

The reason field is always XmCR_DROP_MESSAGE, and dragContext specifies the DragContext object for the drag operation that caused the drop. The dropSiteStatus element is set to either XmDROP_SITE_VALID or XmDROP_SITE_INVALID, depending on the targets that are supported by the drop site and the drag source. The callback routine can change this value if necessary.

The operations and operation fields are set to the possible operations for the drag source data and the current operation, respectively. The dropAction field specifies the action requested by the user. If this field is set to XmDROP, the user has requested a normal drop; if it is set to XmDROP_HELP, the user has requested help for the drop site. We discuss providing help for a drop site in the next section.

The main task of the XmNdropProc is to determine whether or not the operation is possible and to start the data transfer by calling XmDropTransferStart(). This routine creates a DropTransfer object that keeps track of information about the data transfer. The HandleDropLabel() routine initiates the data transfer for the file status drop site, as shown in the following code fragment from Example 22-3:

```
n = 0;
if ((!file_name) || (DropData->dropAction != XmDROP) ||
                    (DropData->operation != XmDROP_COPY)) {
    XtSetArg (args[n], XmNtransferStatus, XmTRANSFER_FAILURE); n++;
    XtSetArg (args[n], XmNnumDropTransfers, 0);                n++;
}
else {
    transferEntries[0].target = FILE_CONTENTS;
    transferEntries[0].client_data = (XtPointer) text_edit;
    transferEntries[1].target = FILE_NAME;
    transferEntries[1].client_data = (XtPointer) file_label;
    transferList = transferEntries;
    XtSetArg (args[n], XmNdropTransfers, transferEntries);               n++;
    XtSetArg (args[n], XmNnumDropTransfers, XtNumber (transferEntries)); n++;
    XtSetArg (args[n], XmNtransferProc, TransferProc);                   n++;
}
XmDropTransferStart (dc, args, n);
```

If the action requested by the user is not a normal drop or if the operation is not a copy operation, we do not process the data transfer. However, we still have to call XmDropTransferStart() to clean up after the whole drag and drop operation. In this case, we set the XmNtransferStatus resource to XmTRANSFER_FAILURE to indicate that the transfer should not proceed. We also set XmNnumDropTransfers to 0.

Otherwise, the drop can proceed, so we establish a list of target data types that we want to receive using the XmNdropTransfers and XmNnumDropTransfer resources. Each entry in XmNdropTransfers is an XmDropTransferEntryRec, which is defined as follows:

```
typedef struct {
    XtPointer    client_data;
```

```
    Atom        target;
} XmDropTransferEntryRec, *XmDropTransferEntry;
```

The `target` field specifies the requested data target, and `client_data` passes any additional data that is necessary to the routine that processes the data transfer. We specify the FILE_CONTENTS and FILE_NAME targets. For each target, we pass the widget that is modified by the data from the drag source as `client_data`. For the FILE_CONTENTS format, the widget is the text entry area `text_edit`, while for FILE_NAME, the widget is the file status area `file_label`.

The final resource that we specify for the DropTransfer is the `XmNtransferProc` routine. This routine is of type `XtSelectionCallbackProc`; it is responsible for actually processing the formatted data that is received from the drag source. The routine is called for each target data type requested by the drop site. This routine takes the following form:

```
typedef void (*XtSelectionCallbackProc)( Widget          widget,
                                         XtPointer       client_data,
                                         Atom            *selection,
                                         Atom            *type,
                                         XtPointer       value,
                                         unsigned long   *length,
                                         int             *format);
```

The *widget* parameter is the widget that requested the data, and *client_data* is the data specified in the `client_data` field of the `XmDropTransferEntryRec` that is being processed. The *type*, *value*, *length*, and *format* arguments contain the data that was converted by the drag source in its `XmNconvertProc`.

The `TransferProc()` routine in Example 22-3 checks the `type` to determine what needs to be done with the data. If the data is FILE_CONTENTS data, the text in value is placed in the Text widget with `XmTextSetString()`. Otherwise, the text is used to create a new value for `XmNlabelString` for the file status area. Since the file status area requests both target data types, both formats are processed by `TransferProc()`.

The `HandleDropText()` routine for the ScrolledText object is very similar to `HandleDropLabel()`. The main difference is that the routine for the text area checks the `XmNexportTargets` resource of the DragContext object to determine whether or not the drag source provides file data. If it does, `HandleDropText()` initiates the data transfer just as in `HandleDropLabel()`. Otherwise, the text routine calls the `XmNdropProc` that we retrieved from the Text widget when we modified the drop site. By calling the original drop routine, we allow the Text widget to process textual data as it would by default. As a result, the user can drop a file object in the text entry area, as well as manipulate textual data in the widget using drag and drop.

Once a data transfer is in progress, additional targets for the DropTransfer object can be specified using `XmDropTransferAdd()`. The primary use of this routine is for move operations. In this case, the drop site receives a copy of the data from the drag source and

then requests that the source delete the data. Once the drop site has stored the data, it can call XmDropTransferAdd() to specify the DELETE target, which indicates to the initiating application that it should delete the data.

Providing Help

Since it is not always obvious what will happen when data is dropped on a particular drop site, the user can request help on a drop site by pressing the HELP or F1 key when the drag icon is over the drop site. An application should provide help information for its drop sites to assist users in understanding the drag and drop capabilities of the application. When the user requests help, the drop site should respond by posting an InformationDialog that explains what would happen and allows the user to proceed with the drop or cancel it.

When the user presses HELP while the drag icon is over a drop site, the XmNdropProc for the drop site is called with the dropAction field in the callback structure set to to XmDROP_HELP. Example 22-4 shows a new HandleDropLabel() routine for the *editor_dnd.c* application that provides help for the file status drop site. The example also shows the HandleDropOK() and HandleDropCancel() callback routines for the help dialog.*

Example 22-4 The HandleDropLabel(), HandleDropOK(), and HandleDropCancel() routines

```
    /* HandleDropLabel() -- start the data transfer when data is dropped in
    ** the filename status area.
    */
    void HandleDropLabel (Widget widget, XtPointer client_data, XtPointer call_
data)
    {
        Display                             *dpy;
        Atom                                FILE_CONTENTS, FILE_NAME;
        XmDropProcCallback                  DropData;
        XmDropTransferEntryRec              transferEntries[2];
        XmDropTransferEntry                 transferList;
        Arg                                 args[10];
        int                                 n, i;
        Widget                              dc;
        Cardinal                            numExportTargets;
        Atom                                *exportTargets;
        Boolean                             file_name = False;
        static XmDropProcCallbackStruct     client;
        static Widget                       dialog = NULL;
        XmString                            message;

        void    HandleDropOK(Widget, XtPointer, XtPointer);
        void    HandleDropCancel(Widget, XtPointer, XtPointer);
```

* XmStringCreateLtoR(), XmMessageBoxGetChild() are deprecated from Motif 2.0 onwards. XmStringGenerate() is only available from Motif 2.0 onwards. XmInternAtom() is marked for deprecation from Motif 2.0 onwards.

```
void    TransferProc (Widget, XtPointer, Atom *, Atom *, XtPointer,
                      unsigned long *, int);

/* intern the Atoms for data targets */
dpy = XtDisplay (toplevel);
FILE_CONTENTS = XInternAtom (dpy, "FILE_CONTENTS", False);
FILE_NAME = XInternAtom (dpy, "FILE_NAME", False);
DropData = (XmDropProcCallback) call_data;
dc = DropData->dragContext;
/* retrieve the data targets and search for FILE_NAME */
n = 0;
XtSetArg (args[n], XmNexportTargets, &exportTargets);       n++;
XtSetArg (args[n], XmNnumExportTargets, &numExportTargets); n++;
XtGetValues (dc, args, n);
for (i = 0; i < numExportTargets; i++) {
    if (exportTargets[i] == FILE_NAME) {
        file_name = True;
        break;
    }
}
/* if one of the targets is not FILE_NAME, transfer fails */
if (!file_name) {
    n = 0;
    XtSetArg (args[n], XmNtransferStatus, XmTRANSFER_FAILURE); n++;
    XtSetArg (args[n], XmNnumDropTransfers, 0);                n++;
}
/* check if the user has requested help */
else if (DropData->dropAction == XmDROP_HELP) {
    /* create a dialog if it doesn't already exist */
    if (!dialog) {
        n = 0;
        message = XmStringGenerate ((XtPointer) help_str,
                        XmFONTLIST_DEFAULT_TAG, XmCHARSET_TEXT, NULL);
        XtSetArg (args[n], XmNdialogStyle,
                        XmDIALOG_FULL_APPLICATION_MODAL); n++;
        XtSetArg (args[n], XmNtitle, "Drop Help");         n++;
        XtSetArg (args[n], XmNmessageString, message);     n++;
        dialog = XmCreateInformationDialog (toplevel, "help", args, n);
        XmStringFree (message);
        XtUnmanageChild (XtNameToWidget (dialog, “Help”));
        XtAddCallback (dialog, XmNokCallback, HandleDropOK,
                        (XtPointer) &client);
        XtAddCallback (dialog, XmNcancelCallback, HandleDropCancel,
                        (XtPointer) &client);
    }
    /* set up the callback structure for when the user proceeds
    ** with the drop and pass it as client data to the callbacks
    ** for the buttons.
    */
    client.dragContext = dc;
    client.x = DropData->x;
    client.y = DropData->y;
    client.dropSiteStatus = DropData->dropSiteStatus;
    client.operation = DropData->operation;
```

```
        client.operations = DropData->operations;
        XtManageChild (dialog);
        return;
    }
    else if (DropData->operation != XmDROP_COPY) {
        /* if the operation is not a copy, the transfer fails */
        n = 0;
        XtSetArg (args[n], XmNtransferStatus, XmTRANSFER_FAILURE); n++;
        XtSetArg (args[n], XmNnumDropTransfers, 0);                n++;
    }
    else {
        /* set up transfer requests since this is a normal drop */
        n = 0;
        transferEntries[0].target = FILE_CONTENTS;
        transferEntries[0].client_data = (XtPointer) text_edit;
        transferEntries[1].target = FILE_NAME;
        transferEntries[1].client_data = (XtPointer) file_label;
        transferList = transferEntries;
        XtSetArg (args[n], XmNdropTransfers, transferEntries); n++;
        XtSetArg (args[n], XmNnumDropTransfers,
                           XtNumber (transferEntries));      n++;
        XtSetArg (args[n], XmNtransferProc, TransferProc);     n++;
    }
    XmDropTransferStart (dc, args, n);
}

/* HandleDropOK() -- callback routine for OK button in drop site help
** dialog that processes the drop as normal.
*/
void HandleDropOK (Widget widget, XtPointer client_data, XtPointer call_data)
{
    Display                     *dpy;
    Atom                        FILE_CONTENTS, FILE_NAME;
    XmDropProcCallbackStruct    *DropData;
    XmDropTransferEntryRec      transferEntries[2];
    XmDropTransferEntry         transferList;
    Arg                         args[10];
    int                         n;
    Widget                      dc;
    void                        TransferProc (Widget, XtPointer, Atom *,
                                              Atom *, XtPointer,
                                              unsigned long *, int);

    /* intern the Atoms for data targets */
    dpy = XtDisplay (toplevel);
    FILE_CONTENTS = XInternAtom (dpy, "FILE_CONTENTS", False);
    FILE_NAME = XInternAtom (dpy, "FILE_NAME", False);

    /* get the callback structure passed via client data */
    DropData = (XmDropProcCallbackStruct *) client_data;
    dc = DropData->dragContext;

    n = 0;
    /* if operation is not a copy, the transfer fails */
```

```
        if (DropData->operation != XmDROP_COPY) {
            XtSetArg (args[n], XmNtransferStatus, XmTRANSFER_FAILURE); n++;
            XtSetArg (args[n], XmNnumDropTransfers, 0);                n++;
        }
        else {
            /* set up transfer requests to process data transfer */
            transferEntries[0].target = FILE_CONTENTS;
            transferEntries[0].client_data = (XtPointer) text_edit;
            transferEntries[1].target = FILE_NAME;
            transferEntries[1].client_data = (XtPointer) file_label;
            transferList = transferEntries;
            XtSetArg (args[n], XmNdropTransfers, transferEntries); n++;
            XtSetArg (args[n], XmNnumDropTransfers,
                                XtNumber (transferEntries));      n++;
            XtSetArg (args[n], XmNtransferProc, TransferProc);     n++;
        }
        XmDropTransferStart (dc, args, n);
    }
    /* HandleDropCancel() -- callback routine for Cancel button in drop site
    ** help dialog that cancels the transfer.
    */
    void HandleDropCancel (Widget widget, XtPointer client_data,
                           XtPointer call_data)
    {
        XmDropProcCallbackStruct     *DropData;
        Arg                          args[10];
        int                          n;
        Widget                       dc;

        /* get the callback structures passed via client data */
        DropData = (XmDropProcCallbackStruct *) client_data;
        dc = DropData->dragContext;
        /* user has cancelled the transfer, so it fails */
        n = 0;
        XtSetArg (args[n], XmNtransferStatus, XmTRANSFER_FAILURE); n++;
        XtSetArg (args[n], XmNnumDropTransfers, 0);                n++;
        XmDropTransferStart (dc, args, n);
    }
```

When the user requests help on the file status drop site, the application displays a help dialog, as shown in Figure 22-8.

Figure 22-8 The drag and drop help dialog

The new `HandleDropLabel()` routine handles the case when the `dropAction` field is set to `XmDROP_HELP`. In this case, the routine creates an InformationDialog if it has not already been created. The `HandleDropOK()` and `HandleDropCancel()` routines are registered for the *OK* and *Cancel* buttons in the dialog. If the dialog already exists, the necessary fields in the `client` structure are specified so that the callback structure information is passed to the callback routines as client data. If the user has performed a normal drop operation, the drop proceeds just as it did in *editor_dnd.c*.

The `HandleDropOK()` routine is invoked when the user presses the *OK* button in the help dialog. This routine proceeds with the drop by calling `XmDropTransferStart()`. The status of the transfer is based on whether the drop performs a copy operation or not. `HandleDropCancel()` cancels the drop when the user presses the *Cancel* button by calling `XmDropTransferStart()` with `XmNtransferStatus` set to `XmTRANSFER_FAILURE`. One thing to note about both of these procedures is that they get the `XmDropProcCallbackStruct` from the `client_data` parameter, since the `call_data` parameter is the callback structure for the dialog.

Providing Custom Drag-under Visuals

Under the preregister protocol, the drop site does not participate during the drag. The initiating application handles the drag-under visual effects based on the value of the `XmNanimationStyle` resource for the drop site. This resource can have one of the following values:

```
XmDRAG_UNDER_HIGHLIGHT        XmDRAG_UNDER_SHADOW_OUT
XmDRAG_UNDER_SHADOW_IN        XmDRAG_UNDER_PIXMAP
XmDRAG_UNDER_NONE
```

The default value is `XmDRAG_UNDER_HIGHLIGHT`, which means that a highlighting rectangle is displayed around the drop site when the drag icon enters it. The drop site can also be displayed with an inset or outset shadow using `XmDRAG_UNDER_SHADOW_OUT` and `XmDRAG_UNDER_SHADOW_IN`, respectively. The `XmDRAG_UNDER_PIXMAP` value

specifies that a special pixmap is displayed in the drop site when the drag icon is in it; the XmNanimationPixmap and XmNanimationMask resources indicate the pixmap that is used. If XmNanimationStyle is set to XmDRAG_UNDER_NONE, there are no animation effects unless they are provided by the XmNdragProc.

Under the dynamic protocol, the drop site can participate in the drag by specifying an XmNdragProc. This callback routine is invoked when the drag icon enters or leaves the drop site, when the drag icon moves within the drop site, and when the operation changes while the icon is in the drop site. The callback receives a callback structure of the type XmDragProcCallbackStruct, which is defined as follows:

```
typedef struct {
    int                reason;
    XEvent             *event;
    Time               timeStamp;
    Widget             dragContext;
    Position           x;
    Position           y;
    unsigned char      dropSiteStatus;
    unsigned char      operation;
    unsigned char      operations;
    Boolean            animate;
} XmDragProcCallbackStruct, *XmDragProcCallback;
```

The reason field is set to one of the following, depending upon the event that triggered the callback:

```
XmCR_DROP_SITE_ENTER_MESSAGE      XmCR_DROP_SITE_LEAVE_MESSAGE
XmCR_DRAG_MOTION_MESSAGE          XmCR_OPERATION_CHANGED_MESSAGE
```

The dragContext field specifies the current DragContext object, while dropSiteStatus is set to either XmDROP_SITE_VALID or XmDROP_SITE_INVALID, based on the values of XmNimportTargets and XmNexportTargets for the drop site and the drag source, respectively. The operations and operation fields are set to the possible operations for the drag source data and the current operation, respectively. The value of operations is based on the value of the XmNdragOperations resource for the DragContext, while the value of operation is based on operations and the value of XmNdropSiteOperations.

The XmNdragProc can change the values of these three fields based on any special processing it performs, such as handling simulated drop sites. When the routine is done, the toolkit uses these values of the fields to initialize the fields in the callback structure that is passed to the corresponding DragContext callback routine in the initiating application.

The animate field specifies whether the toolkit or the receiving client is handling drag-under effects for the drop site. If the value is True, as it is by default, the toolkit handles the effects based on the XmNanimationStyle resource. The receiving client can set the field to False so that it is responsible for providing drag-under effects. The main use of

the XmNdragProc is for providing specialized drag-under effects, such as actual animation, that the toolkit itself does not support.

Summary

The drag and drop capabilities provided by Motif are highly customizable, so an application can use the toolkit to implement whatever functionality is necessary. The examples in this chapter have demonstrated many of the techniques that an application needs to use to provide drag and drop functionality, but they really just scratch the surface of what is possible.

Our examples implement the drag and drop features directly in application code because that is sufficient for our purposes. However, if you are developing real applications, you should think seriously about encapsulating drag and drop functionality in widgets, so that you can reuse the components in all of the applications.

In Chapter 23, *The Uniform Transfer Model*, we introduce the schemes introduced in Motif 2.0 which build on top of the mechanisms described in this chapter and Chapter 21, *The Clipboard*. These schemes allow the programmer to transfer data between widgets or the clipboard, either through the primary selection mechanisms or drag and drop, using a single programming interface.

In this chapter:
- *Overview*
- *Exporting the Data*
- *Requesting the Data Format*
- *Importing the Data*
- *Batched Data Transfer*
- *An Example*
- *Summary*

23

The Uniform Transfer Model

This chapter describes the new features introduced in Motif 2.0 for transferring application data in a consistent and uniform manner.

In Motif 1.2, whenever widget data needed to be transferred in the application, a distinct set of code had to be written depending upon the nature of the transfer. Since Motif widgets support several styles of data transfer - to the primary or secondary X Selection, to the Motif Clipboard, or to another widget through the Drag and Drop mechanisms - the task of providing a fully featured interface could be somewhat detailed and repetitive in the implementation. This was repetitive because the data to be transferred might in fact be the same in each case. It was certainly detailed, because several implementations would be required, one for each of the supported data transfer methods.

From Motif 2.0, the Uniform Transfer Model makes it possible to transfer data using any of the transfer methods using a single programming interface. The UTM is designed to allow applications to use common code for all the supported data transfer requirements. It is also designed to ease the way in which new transfer targets can be added to the system. All of the existing pre-Motif 2.0 transfer methodologies have been re-written to utilize the Uniform Transfer Model internally where appropriate.

The Motif data transfer utilities have also been interwoven with the Trait mechanisms. In the absence of transfer code written by the programmer, the Trait system steps in and provides default data transfer capability for a large range of built-in data types. Although the details of Traits are beyond the scope of this book, belonging as they do much more to a widget authors manual, suffice it to say that because of this default Trait behavior, the programmer only needs to consider writing Uniform Transfer Model code in order to transfer data in a new and application-specific format. The Traits will transfer everything else automatically, under the assumption that it is the current widget selection (whatever that may mean) which is the data to be transferred. The set of available built-in target types is given in Table 22-1 of Chapter 22, *Drag and Drop*, although not all built-in target types are transferrable in every widget context: what a widget can export or import by default is widget class specific.

Overview

The Uniform Transfer Model is implemented through two new callback resources, the XmNconvertCallback, and the XmNdestinationCallback, and a new procedure, the Transfer procedure, which can also be considered as just another callback, although it is not specified using widget resources. In essence, the XmNconvertCallback is responsible for exporting the data, the XmNdestinationCallback requests the import format, the Transfer procedure performs the actual import of the arriving data. The XmNdestinationCallback routine negotiates with the XmNconvertCallback to find the best data format, and then sets up the Transfer callback to do the work. Data is transferred to and from the source and destination through elements in the callback structure supplied to each callback. In addition to transferring data, the callbacks can also transfer protocol when and if required. The destination of the data transfer can request a list of target types in which the source is prepared to export data, as well as requesting data in a particular and specific target format. The source in turn can inform the destination of the available export data types, as well as actually delivering the data itself. There is thus a two-way communication between the source and destination: each callback may in fact fire more than once as the two ends of the transfer communicate and negotiate the preferred format in which the data is to be transferred. The XmNconvertCallback in particular may be called multiple times. The simple case, however, need only concern itself with a single set of transactions: the destination requests data in a specific format, and the source provides it.

The XmNconvertCallback

The XmNconvertCallback resource is implemented into the Primitive widget class, and is inherited by all sub-classes. It is also implemented in the DrawingArea, Scale, and Container widget classes. The resource is thus available in all widget contexts where the user directly interacts - the widgets which the user can actually use, as opposed to a Manager which simply lays out other components. The callback is not available for Gadget classes.

The XmNdestinationCallback

The XmNdestinationCallback is directly implemented into the List, Container, DrawingArea, Text and TextField widget classes. It may seem surprising that the callback is not implemented into a wider range of classes, but if we consider what it really means to have an XmNdestinationCallback in a widget class, the matter becomes clearer.

Recall that the UTM encapsulates three distinct methods of data transfer: X selections, the Clipboard, and Drag-and-Drop. The destination of an X Selection transfer is via the X server, and this is also indirectly true for the Motif Clipboard, which is really a higher level view onto the X selection mechanisms. The destination of a Drag-and-Drop, however, is an application widget. In other words, the toolkit only needs to implement the

XmNdestinationCallback into widget classes which are not only likely to be required as a drop site, but also in which it makes sense to allow foreign (non-toolkit) targets as the type of the data transfer.

This is not to say that data cannot be transferred to other widget classes using drag-and-drop: Motif implements some Trait mechanisms into other widget classes as well, and these implement transfer of the built-in Motif target types. For example, the Label (and derived classes) and Scale widgets handle data transfer using drag-and-drop by these default Trait mechanisms. Again, the details of the Trait implementation are beyond the scope of this book.

Exporting the Data

Each XmNconvertCallback is passed a pointer to an XmConvertCallbackStruct, which is defined in the header file <*Xm/Transfer.h*>. This must be included either directly or indirectly by the programmer; it can be loaded indirectly simply by including the general Motif header <*Xm/Xm*.h>, as this includes the header internally. The XmConvertCallbackStruct is defined as follows:

```
typedef struct {
    int              reason;
    XEvent           *event;
    Atom             selection;
    Atom             target;
    XtPointer        source_data;
    XtPointer        location_data;
    int              flags;
    XtPointer        parm;
    int              parm_format;
    unsigned long    parm_length;
    Atom             parm_type;
    int              status;
    XtPointer        value;
    Atom             type;
    int              format;
    unsigned long    length;
} XmConvertCallbackStruct;
```

At first sight, the structure is somewhat daunting. However, because not all of the elements of the structure are applicable in any given context (the structure unifies the requirements of three distinct data transfer types, and some of the elements are used by the toolkit to communicate with itself), programming the callback structure is not as complex as it looks. We will start by looking at the basic common elements before looking at specific data transfer types.

Firstly, we can deduce the type of the current data transfer if we need to by inspecting the selection element. This will be one of CLIPBOARD, _MOTIF_DROP, PRIMARY, or

SECONDARY, expressed as an Atom. In other words, all the possible types of data transfer which the UTM supports.

Secondly, we place the data to transfer into the value element. At this address, we need to dynamically allocate memory to hold the transfer data, but we need not concern ourselves with freeing up the memory at the right point in the transfer process: the toolkit does this for us. It is assumed that value points to an allocated array of items of some kind - the number of such items in the array should be specified using the length element. The size of each element in the array is specified using the format element. This is not a simple *sizeof (data)* result, but a logical size: if you are transferring character-based data, format should be set to 8. For a list of short quantities, format should be set to 16. A list of long values is expressed by setting format to 32. A format of zero is reserved to mean that value contains no data. The logical type of the data being transferred, expressed as an Atom, is placed into the type element. If we define a new application-specific data transfer, we need to define a logical (and hopefully unique) name for it, and express this in the type field.

Thirdly, we need to deduce the actual data to transfer (although we already know that it is placed into the value element when we do export it). Normally this will be related to the widget selection in some way - the default Trait transfer mechanisms assume this - or to some application specific data. But it need not be. Depending upon the nature of the transfer, we may be simply asked to convert given data of one type into data of another: this XmNconvertCallback instance really is a converter. For example, the various negotiations between a source and destination might result in the destination asking the source to convert specific supplied data in a specific format. We will know if this has happened if the location_data field is not NULL. The format of the data to convert can be deduced from the target field. In principle, all we have to do is convert the data at location_data into the requested target format, and place the result into the value field, also specifying the size and length of the data at value using the format and length elements, as specified in the previous paragraph. In practice, the location_data will most likely be NULL: the interpretation of location_data is generally widget-class specific in any case. Whether converting supplied data or providing our own, this holds true: we export our data in the format as requested by the target field.

There is a special case. The destination may be requesting of us, the source, the list of available data formats in which we are prepared to export our application data. In this case, the target element will variously be the value TARGETS, _MOTIF_EXPORT_TARGETS, _MOTIF_CLIPBOARD_TARGETS, or _MOTIF_DEFERRED_CLIPBOARD_TARGETS expressed as an Atom. In response, we construct a list of Atoms representing the set of available export formats, and return this in the value field. The length field is set to the number of such Atoms, and the format field becomes 32, because Atoms are long quantities. Hopefully what happens next is that the destination will choose a preferred export target, and communicate this back to the source, whereupon the

XmNconvertCallback is invoked again, this time with a specific target chosen from our supplied list. This part of the negotiation is described below in the *Requesting the Data Format* Section.

Exporting to the Clipboard

When transferring data to the Motif Clipboard, the parm, parm_format, parm_length, and parm_type elements come into play. This is the case when the target element is set to the Atoms representing _MOTIF_CLIPBOARD_TARGETS or _MOTIF_DEFERRED_CLIPBOARD_TARGETS. The parm element contains an array of Clipboard-specific elements specifying the type of clipboard transfer required. The elements will each be one of XmCOPY, XmMOVE, XmLINK. The param_length field specifies the number of such data elements, the param_format field gives the logical size of such elements. Again, 8 means char-sized array items, 16 means short, 32 means long. The param_type element specifies the logical type of the parm data. Much of all this is over-engineering: in practice, the parm_type field will be the constant Atom XA_INTEGER. These fields do however give a consistency to the data transfer process: they replicate the logic of the value/format/length/type fields so that incoming and outgoing data is transferred in a similar fashion, albeit in different elements of the callback structure. All of these parm fields are read-only in any case, and pretty much toolkit internal.

Exporting to a Widget (Drag and Drop)

When exporting data in response to a drag-and-drop operation, the source_data element is operative. It points to an XmDragContext object, which gives the programmer a handle whereby you might specify any customised drag visuals you wish to associate with the data transfer. Drag Context resources are covered in Chapter 22, *Drag and Drop*. For other types of data transfer, the source_data element of the callback structure is simply NULL.

The status field is also probably a touch of over-engineering, used for drag-and-drop transfers. Recall that there are also widget Traits which know how to convert built-in Motif target types. These are called if no XmNconvertCallback is specified by the programmer. They may, however, be called even if the programmer does supply a callback. Whether this is the case or not depends on the status element.

If status is XmCONVERT_MERGE, the Trait mechanisms are invoked after the callback, merging any data so produced with the results of the callback. This can be useful: if the destination requests data in a built-in format, we can leave the value element as NULL and set status to XmCONVERT_MERGE, so that the toolkit does all the work.

If status is XmCONVERT_DONE, we tell the toolkit not to invoke the Trait mechanisms: the result of the conversion is only the data we supply to the value field. For efficiency reasons, we should use this if the target type is application-specific, otherwise the Traits will go to the bother of working out that they don't know how to convert the data: it is much better if they are never called in the first place.

If status is XmCONVERT_REFUSE, we tell the toolkit that the conversion process is to terminate immediately after this callback without invoking any Traits. But then neither is data transfer effective. This is the transfer failure signal - we cannot transmit data as requested for whatever reason.

The default value of status is XmCONVERT_DEFAULT, which works out to have the same behavior as XmCONVERT_MERGE. Since the whole point of the Uniform Transfer Model is that the programmer really only needs to add callbacks when transferring new application-specific data types, you probably want to change this for a start. On the other hand, the default value as it stands makes the built-in behavior work, so we can excuse the implementation.

As an example, the following is a convert procedure which exports a date in the form of three integers. The procedure also exports the list of supported targets when requested.

Example 23-1 The convert_callback() procedure

```
void convert_callback ( Widget       widget,
                        XtPointer    client_data,
                        XtPointer    call_data)
{
    XmConvertCallbackStruct *cptr = (XmConvertCallbackStruct *) call_data;
    Atom         TARGETS, EXPORTS, CB_TARGETS, DATE_TARGET;
    Atom         *targets;
    Display      *display = XtDisplay (widget);

    /* Intern the atoms */
    TARGETS = XInternAtom (display, "TARGETS", False);
    EXPORTS = XInternAtom (display, "_MOTIF_EXPORT_TARGETS", False);
    CB_TARGETS = XInternAtom (display, "_MOTIF_CLIPBOARD_TARGETS", False);
    DATE_TARGET = XInternAtom (display, "APPLICATION_DATE", False);

    /* If the destination has requested the list of targets */
    /* we return this as the convert data */
    if ((cptr->target == TARGETS) || (cptr->target == CB_TARGETS) ||
                                     (cptr->target == EXPORTS)) {
        /* A request from the destination for the supported */
        /* data types we are willing to handle */
        targets = (Atom *) XtMalloc ((unsigned) sizeof (Atom));
        targets[0] = DATE_TARGET;
        cptr->type = XA_ATOM;
        cptr->value = (XtPointer) targets;
        cptr->length = 1;
        cptr->format = 32;

        /* Merge the target with the toolkit Trait-supported targets */
        /* If we only wanted to export our own format, we would return */
        /* XmCONVERT_DONE at this point */
        cptr->status = XmCONVERT_MERGE;
    }
    else {
        /* A request from the destination for a specific data type */
```

```
        if (cptr->target == DATE_TARGET) {
            /* An unspecified routine to fetch the date */
            /* Presumably the current widget is a date field */
            /* So we pass the widget parameter to fetch this */
            void GetDate (Widget, short *, short *, short *);
            short *date;

            /* We don't worry about freeing this - the toolkit does it */
            date = (short *) XtMalloc ((unsigned) 3 * sizeof (short));

            GetDate (widget, &date[0], &date[1], &date[2]);

            cptr->value = (XtPointer) date;
            cptr->length = 3;
            cptr->type = cptr->target; /* As requested */
            cptr->format = 16; /* 8 = char, 16 = short, 32 = long */

            /* No point in calling the Trait mechanisms */
            /* Since we have already handled the target */
            cptr->status = XmCONVERT_DONE;
        }
        else {
            /* Presumably a Trait-supported built-in target */
            /* Again, if not interested, return XmCONVERT_DONE */
            cptr->status = XmCONVERT_MERGE;
        }
    }
}
```

Requesting the Data Format

Turning to the destination site, each `XmNdestinationCallback` is passed a pointer to an `XmDestinationCallbackStruct`, which is defined in the <*Xm/Transfer.h*> header file. The structure is defined as follows:

```
typedef struct {
    in              reason;
    XEvent          *event;
    Atom            selection;
    XtEnum          operation;
    int             flags;
    XtPointer       transfer_id;
    XtPointer       destination_data;
    XtPointer       location_data;
    Time            time;
} XmDestinationCallbackStruct;
```

Just as not all elements of the `XmConvertCallbackStruct` are applicable in any given data transfer type, so the various fields of the `XmNdestinationCallback` become operative at different times.

Firstly, the selection element specifies the type of data transfer, and it may have the values CLIPBOARD, PRIMARY, SECONDARY, or _MOTIF_DROP, expressed as an Atom.

Secondly, the flags element indicates whether the source of the data transfer is the same as the destination. Possible values are XmCONVERTING_SAME and XmCONVERTING_NONE, the latter value indicating different source and destination locations.

The transfer_id element is simply a unique identifier for the current transaction. This field is read-only. As far as application programming is concerned, it serves as a hook or identifier which is passed to further transfer functions which are discussed in the *Transfer Routines* section below.

The time field is a server timestamp, and is read-only and internal to the toolkit.

The location_data field specifies where the data is to be transferred. The interpretation of this field is widget-class specific. For example, if the destination is a Container, location_data points to an XPoint structure which describes the x, y coordinates of the transfer. If location_data is NULL, this should be interpreted as a request to deposit the transfer data at the current cursor location within the destination widget concerned.

The destination_data element provides information concerning the destination of the transfer operation. If the selection is _MOTIF_DROP, the callback has been invoked by an XmDropProc at a drop site, and the destination_data element points to an XmDropProcCallbackStruct. If selection is SECONDARY, destination_data contains an Atom which represents the format to which the selection owner believes that the data should be converted. Otherwise, for clipboard or primary selection, the element is NULL.

The operation field indicates the type of data transfer; for a clipboard transfer, it can have the values XmCOPY, XmLINK, or XmMOVE. Otherwise, the value is XmOTHER, and destination_data will indicate the type of transfer required where appropriate.

Transfer Routines

As far as application programming of drag-and-drop is concerned, much of the above information is irrelevant, and can be treated as toolkit internal. The average destination callback simply performs the following tasks:

- it either asks the source of the data to provide a list of available export targets, or asks for a specific export target,
- it sets up a further routine which will perform the actual data import.

It does both of these things using the routine XmTransferValue(), which is defined as follows:

```
void XmTransferValue ( XtPointer          transfer_id,
                       Atom               target,
                       XtCallbackProc     transfer_routine,
```

```
                    XtPointer          client_data,
                    unsigned long      timestamp)
```

The *transfer_id* parameter is simply passed through from the transfer_id element of the XmDestinationCallbackStruct passed to the destination callback itself. The target parameter specifies the format in which the source XmNconvertCallback should export the data. Specially, it can be a request for the available list of export targets, and this is performed by passing TARGETS expressed as an Atom. The *transfer_routine* is a further procedure which will perform the actual data transfer; any application-specific data which is to be passed to the routine is specified in the *client_data* parameter. Lastly, the *timestamp* parameter is a unique server timestamp for the current transfer. The function XtLastTimestampProcessed() should be used to construct this value.

As an example, the following code fragment illustrates an XmNdestinationCallback which requests the list of available targets from the source.

Example 23-2 The destination_callback() procedure

```
void destination_callback ( Widget         w,
                            XtPointer      client_data,
                            XtPointer      call_data)
{
extern void transfer_procedure (Widget, XtPointer, XtPointer);
XmDestinationCallbackStruct *dptr =
                    (XmDestinationCallbackStruct *) call_data;
Atom TARGETS = XInternAtom (XtDisplay (w), "TARGETS", False);

XmTransferValue ( dptr->transfer_id,
                  TARGETS,
                  transfer_procedure,
                  NULL,
                  XtLastTimestampProcessed (XtDisplay (w)));
}
```

Despite the apparent complexity surrounding the various callback structures associated with the Uniform Transfer Model, and the very dry paragraphs which were required to describe them, when it boils down to a real example, the required application code turns out to be extremely simple. We ignore almost every field of the XmDestinationCallbackStruct passed to us: only the current transfer_id is of any real interest. We construct an Atom in order to query the list of targets at the source of the data, we specify a procedure to perform the actual data transfer, and finally send the request off to the source using the UTM routine XmTransferValue(). The procedure to perform the data transfer, transfer_procedure, is described in the following section. Whether requesting a list of available targets or asking for a specific data format, the code is essentially the same: only the Atom changes.

Importing the Data

A Transfer Procedure has to be prepared to perform two tasks. Its primary role is to collect the exported data passed to it from callback structure fields, and process this in an appropriate fashion. However, there is a second possibility: the source of the data may have sent a list of available export formats in response to a destination callback request, and it is also the responsibility of the transfer procedure to choose from the available options.

Note that it is not strictly necessary to have separate destination and transfer procedures: a single callback could in fact perform all the destination side tasks, in which case the transfer procedure as specified through XmTransferValue() would be the destination callback itself. This however makes the destination callback unnecessarily complex, you would have to discriminate between the various states of the transfer using the callback reason field, and maintain a separate block of code in the callback for each case. Since logically the XmNdestinationCallback and the transfer procedure are performing separate tasks, we prefer to split the destination side into separate destination and transfer callbacks, and recommend that you do the same. In any case, the callback structure passed to a transfer and a destination callback is not the same: you would have to perform some suitable casting of the callback call_data to each type if you insist on maintaining a single destination side routine.

Each transfer procedure (as registered through XmTransferValue()) is passed an XmSelectionCallbackStruct when invoked. Again, the structure is defined in the header file <*Xm/Transfer.h*>. The definition is as follows:

```
typedef struct {
    int              reason;
    XEvent           *event;
    Atom             selection;
    Atom             target;
    Atom             type;
    XtPointer        transfer_id;
    int              flags;
    int              remaining;
    XtPointer        value;
    unsigned long    length;
    int              format;
} XmSelectionCallbackStruct;
```

The interpretation of each of the elements is exactly the same as the corresponding fields in the XmConvertCallbackStruct and XmDestinationCallbackStruct definitions. The value, length, and format fields specify the incoming data from the source of the transfer. The type of the transfer is expressed in the target element: this will either be a specific export data target, or TARGETS if the incoming data is a reply to a request for the supported range of export types.

The following example is a transfer procedure which is hoping to receive a date, as exported in the code of Example 23-1. On receipt of the date, it simply displays it under the assumption that this destination is a TextField widget.

Example 23-3 The transfer_callback() routine

```
void transfer_callback ( Widget       w, /* The destination widget */
                         XtPointer    client_data,
                         XtPointer    call_data)
{
    XmSelectionCallbackStruct *sptr =
                                (XmSelectionCallbackStruct *) call_data;
    Atom       TARGETS, EXPORTS, CB_TARGETS, DATE_TARGET;
    Display    *display = XtDisplay (w);
    Atom       *targets, choice;
    int        i;

    choice = (Atom) 0;
    TARGETS = XInternAtom (display, "TARGETS", False);
    EXPORTS = XInternAtom (display, "_MOTIF_EXPORT_TARGETS", False);
    CB_TARGETS = XInternAtom (display, "_MOTIF_CLIPBOARD_TARGETS", False);
    DATE_TARGET = XInternAtom (display, "APPLICATION_DATE", False);

    if ((sptr->type == XA_ATOM) &&
                ((sptr->target == TARGETS) || (sptr->target == CB_TARGETS) ||
                 (sptr->target == EXPORTS))) {
        /* The source has sent us a list of available data formats */
        /* in which it is prepared to export the data */
        /* We get to choose one */
        /* The value field contains the list of available targets */
        targets = (Atom *) sptr->value;

        for (i = 0; i < sptr->length; i++) {
            if (targets[i] == DATE_TARGET) {
                /* We like this: its our own preferred format */
                choice = targets[i];
            }
        }

        /* If the source is not prepared to export in a format */
        /* of our choice, we can either let the toolkit handle it */
        /* under the assumption its a built-in data type */
        /* or we can signal that the transfer is no good */

        /* Lets assume we are only interested in our own data transfer */
        if (choice == (Atom) 0) {
            XmTransferDone (sptr->transfer_id, XmTRANSFER_DONE_FAIL);
            return;
        }

        /* On the other hand, if we have chosen a target */
        /* We simply go back to the source asking for it */
        XmTransferValue (sptr->transfer_id,
                         choice,
```

```
                        transfer_callback, /* Round we go again */
                        NULL,
                        XtLastTimestampProcessed (display));
        }
        else {
            /* The source has sent us a specific data format */
            /* It **ought** to be DATE_TARGET, but better check... */
            if (sptr->target == DATE_TARGET) {
                /* Three integers, we assume */
                /* We really ought to check sptr->length */
                short *date = (short *) sptr->value;
                char buf[32];

                /* Not pretty, but it will do for here */
                (void) sprintf (buf, "%d/%d/%d", date[0], date[1], date[2]);
                XmTextFieldSetString (w, buf);
                XmTransferDone (sptr->transfer_id, XmTRANSFER_DONE_SUCCEED);
            }
            else {
                /* We should not be here. Someone has written a */
                /* UTM procedure which is out of step. */
                XmTransferDone (sptr->transfer_id, XmTRANSFER_DONE_FAIL);
            }
        }
    }
```

The example is straight forward. The top half of the `transfer_callback`() returns to the source the preferred import target. The list of available options has been sent to the routine in the `value` element of the callback structure. All the routine has to do is pick from the list, and go back to the source asking for the preferred data format. The bottom half of the routine is the case where the source has sent the data in the preferred format. The callback only has to extract the data from the `value` field, and process it. The only other issue is how the callback terminates the transfer, whether if there has been an error, or if the data has all been processed. It does this through the routine `XmTransferDone`(), which is specified as follows:

```
void XmTransferDone (XtPointer transfer_id, XmTransferStatus status)
```

The `transfer_id` parameter is the unique handle on the current transfer, and is passed through from of the relevant callback structure element. The `status` field indicates why the transfer is terminated. Possible values are:

```
XmTRANSFER_DONE_SUCCEED          XmTRANSFER_DONE_FAIL
XmTRANSFER_DONE_CONTINUE         XmTRANSFER_DONE_DEFAULT
```

`XmTRANSFER_DONE_SUCCEED`, `XmTRANSFER_DONE_FAIL` are self-evident. The value `XmTRANSFER_DONE_DEFAULT` requests that the Trait mechanisms take over to handle the current transfer target. `XmTRANSFER_DONE_CONTINUE` is complex to describe, involving as it does Motif `_MOTIF_SNAPSHOT` operations, which are part of the implementation of deferred clipboard data transfer. We are going to treat all this as entirely toolkit internal.

Batched Data Transfer

If we need to send multiple separate pieces of information as part of the data transfer, we can in principle simply call XmTransferValue() multiple times from within the UTM callbacks. This is not necessarily efficient or transfer-order safe. For example, if a source declares that it can support export multiple targets, the destination might choose not simply to pick one from the list, but decide to ask for more than one of the available target types. Each data type will involve a separate invocation of the convert and transfer callbacks at each end, so that many messages will go back and forth. A certain degree of synchronization is required to make all this work successfully.

To get round this problem, there are two routines available which batch up XmTransferValue() requests. To initiate a new batch, call the routine XmTransferStartRequest(). Then call XmTransferValue() as needed. To end the batch, use the routine XmTransferSendRequest(). These routines are defined as follows:

```
void XmTransferStartRequest (XtPointer transfer_id)
void XmTransferSendRequest  (XtPointer transfer_id, Time time)
```

These are extremely simple to use: the transfer_id is taken from the current callback data structure, and the time is specified using XtLastTimestampProcessed().

An Example

Example 23-4 creates two components: a SpinBox configured to display the date, and a TextField. The SpinBox we will treat as the source of the transfer, and the TextField as the destination.

Example 23-4 The utm.c program

```
/* utm.c: transfers data between a specimen SpinBox and
** a TextField using the Uniform Transfer Model
*/

#include <X11/Intrinsic.h>
#include <Xm/Transfer.h>
#include <Xm/RowColumn.h>
#include <Xm/SpinB.h>
#include <Xm/TextF.h>
#include <Xm/Label.h>

/*
** GetDate(): fetches the various elements of a date from multiple
** SpinBox TextField children.
*/
void GetDate (Widget text, short *day, short *month, short *year)
{
    int d, m, y;
```

```
    Widget spinb = XtParent (text);

    XtVaGetValues (XtNameToWidget (spinb, "days"), XmNposition, &d, 0);
    XtVaGetValues (XtNameToWidget (spinb, "months"), XmNposition, &m, 0);
    XtVaGetValues (XtNameToWidget (spinb, "years"), XmNposition, &y, 0);

    *day = (short) d;
    *month = (short) m;
    *year = (short) y;
}

/*
** convert_callback(): exports data in the format requested
** by the destination of the data transfer.
*/
void convert_callback ( Widget       widget,
                        XtPointer    client_data,
                        XtPointer    call_data)
{
    XmConvertCallbackStruct *cptr = (XmConvertCallbackStruct *) call_data;
    Atom         TARGETS, EXPORTS, CB_TARGETS, DATE_TARGET;
    Atom         *targets;
    Display      *display = XtDisplay (widget);

    /* Intern the atoms */
    TARGETS = XInternAtom (display, "TARGETS", False);
    EXPORTS = XInternAtom (display, "_MOTIF_EXPORT_TARGETS", False);
    CB_TARGETS = XInternAtom (display, "_MOTIF_CLIPBOARD_TARGETS", False);
    DATE_TARGET = XInternAtom (display, "APPLICATION_DATE", False);

    /* If the destination has requested the list of targets */
    /* we return this as the convert data */
    if ((cptr->target == TARGETS) || (cptr->target == CB_TARGETS) ||
                                (cptr->target == EXPORTS)) {
        /* A request from the destination for the supported */
        /* data types we are willing to handle */

        targets = (Atom *) XtMalloc ((unsigned) sizeof (Atom));
        targets[0] = DATE_TARGET;
        cptr->type = XA_ATOM;
        cptr->value = (XtPointer) targets;
        cptr->length = 1;
        cptr->format = 32;
        cptr->status = XmCONVERT_MERGE;
    }
    else {
        /* A request from the destination for a specific data type */
        if (cptr->target == DATE_TARGET) {
            short *date;
            date = (short *) XtMalloc ((unsigned) 3 * sizeof (short));

            GetDate (widget, &date[0], &date[1], &date[2]);

            cptr->value = (XtPointer) date;
```

```
                cptr->length = 3;
                cptr->type = cptr->target; /* As requested */
                cptr->format = 16; /* 8 = char, 16 = short, 32 = long */
                cptr->status = XmCONVERT_DONE;
        }
        else {
            /* Presumably toolkit built-in type */
            /* Let the Traits take over */
            cptr->value = (XtPointer) 0;
            cptr->length = 0;
            cptr->format = 0;
            cptr->type = cptr->target;
            cptr->status = XmCONVERT_MERGE;
        }
    }
}

/*
** transfer_callback(): performs the import of the data at
** the destination site.
*/
void transfer_callback ( Widget       w, /* The destination widget */
                         XtPointer    client_data,
                         XtPointer    call_data)
{
    XmSelectionCallbackStruct *sptr =
                            (XmSelectionCallbackStruct *) call_data;
    Atom       EXPORTS, TARGETS, CB_TARGETS, DATE_TARGET;
    Display    *display = XtDisplay (w);
    Atom       *targets, choice;
    int        i;

    choice = (Atom) 0;
    TARGETS = XInternAtom (display, "TARGETS", False);
    EXPORTS = XInternAtom (display, "_MOTIF_EXPORT_TARGETS", False);
    CB_TARGETS = XInternAtom (display, "_MOTIF_CLIPBOARD_TARGETS", False);
    DATE_TARGET = XInternAtom (display, "APPLICATION_DATE", False);

    if ((sptr->type == XA_ATOM) && ((sptr->target == TARGETS) ||
                                    (sptr->target == CB_TARGETS) ||
                                    (sptr->target == EXPORTS))) {
        /* The source has sent us a list of available data formats */
        /* in which it is prepared to export the data */
        /* We get to choose one */
        /* The value field contains the list of available targets */
        targets = (Atom *) sptr->value;

        for (i = 0; i < sptr->length; i++) {
            if (targets[i] == DATE_TARGET) {
                /* We like this: its our own preferred format */
                choice = targets[i];
            }
        }
```

```
        /* If the source is not prepared to export in a format */
        /* of our choice, we can either let the toolkit handle it */
        /* under the assumption its a built-in data type */
        /* or we can signal that the transfer is no good */

        /* Lets assume we are only interested in our own data transfer */
        if (choice == (Atom) 0) {
            XmTransferDone (sptr->transfer_id, XmTRANSFER_DONE_FAIL);
            return;
        }

        /* On the other hand, if we have chosen a target */
        /* We simply go back to the source asking for it */
        XmTransferValue (sptr->transfer_id,
                         choice,
                         transfer_callback, /* Round we go again */
                         NULL,
                         XtLastTimestampProcessed (display));
    }
    else {
        /* The source has sent us a specific data format */
        /* It **ought** to be DATE_TARGET, but better check... */
        if (sptr->target == DATE_TARGET) {
            /* Three integers, we assume */
            /* We really ought to check sptr->length */
            short *date = (short *) sptr->value;
            char buf[32];

            /* Not pretty, but it will do for here */
            (void) sprintf (buf, "%d/%d/%d", date[0], date[1], date[2]);
            XmTextFieldSetString (w, buf);

            XmTransferDone (sptr->transfer_id, XmTRANSFER_DONE_SUCCEED);
        }
        else {
            /* We should not be here. Someone has written a */
            /* convert procedure which is out of step. */

            XmTransferDone (sptr->transfer_id, XmTRANSFER_DONE_FAIL);
        }
    }
}

/*
** destination_callbvack: simply asks the source for the
** set of formats it is prepared to send the data in,
** and sets up a transfer procedure to import the data when sent.
*/
void destination_callback ( Widget       w,
                            XtPointer    client_data,
                            XtPointer    call_data)
{
    XmDestinationCallbackStruct *dptr =
                               (XmDestinationCallbackStruct *) call_data;
```

```
    Atom TARGETS = XInternAtom (XtDisplay (w), "TARGETS", False);

    XmTransferValue (dptr->transfer_id,
                     TARGETS,
                     transfer_callback,
                     NULL,
                     XtLastTimestampProcessed (XtDisplay (w)));
}

main (int argc, char *argv[])
{
    Widget         toplevel, spin, rowcol;
    Widget         day, month, year, text;
    XtAppContext   app;
    Arg            args[8];
    int            n;

    XtSetLanguageProc (NULL, NULL, NULL);

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

    rowcol = XmCreateRowColumn (toplevel, "rowcol", NULL, 0);

    /* Create the SpinBox */
    spin = XmCreateSpinBox (rowcol, "spin", NULL, 0);

    /* Create the Days field */
    n = 0;
    XtSetArg (args[n], XmNspinBoxChildType, XmNUMERIC); n++;
    XtSetArg (args[n], XmNcolumns, 2);                  n++;
    XtSetArg (args[n], XmNeditable, FALSE);             n++;
    XtSetArg (args[n], XmNminimumValue, 1);             n++;
    XtSetArg (args[n], XmNmaximumValue, 31);            n++;
    XtSetArg (args[n], XmNposition, 1);                 n++;
    XtSetArg (args[n], XmNwrap, TRUE);                  n++;

    day = XmCreateTextField (spin, "days", args, n);
    XtManageChild (day);

    /* Create the Months field */
    n = 0;
    XtSetArg (args[n], XmNspinBoxChildType, XmNUMERIC); n++;
    XtSetArg (args[n], XmNcolumns, 2);                  n++;
    XtSetArg (args[n], XmNeditable, FALSE);             n++;
    XtSetArg (args[n], XmNminimumValue, 1);             n++;
    XtSetArg (args[n], XmNmaximumValue, 12);            n++;
    XtSetArg (args[n], XmNposition, 1);                 n++;
    XtSetArg (args[n], XmNwrap, TRUE);                  n++;
    month = XmCreateTextField (spin, "months", args, n);
    XtManageChild (month);

    n = 0;
```

```
        XtSetArg (args[n], XmNspinBoxChildType, XmNUMERIC); n++;
        XtSetArg (args[n], XmNcolumns, 4);                  n++;
        XtSetArg (args[n], XmNeditable, FALSE);             n++;
        XtSetArg (args[n], XmNminimumValue, 1900);          n++;
        XtSetArg (args[n], XmNmaximumValue, 2100);          n++;
        XtSetArg (args[n], XmNposition, 2000);              n++;
        XtSetArg (args[n], XmNwrap, TRUE);                  n++;
        year = XmCreateTextField (spin, "years", args, n);
        XtManageChild (year);

        /* The destination of the data transfer */
        n = 0;
        XtSetArg (args[n], XmNeditable, FALSE); n++;
        text = XmCreateTextField (rowcol, "drop-site", args, n);
        XtManageChild (text);

        /* Now program the UTM */
        XtAddCallback (day, XmNconvertCallback, convert_callback, NULL);
        XtAddCallback (month, XmNconvertCallback, convert_callback, NULL);
        XtAddCallback (year, XmNconvertCallback, convert_callback, NULL);
        XtAddCallback (text, XmNdestinationCallback, destination_callback, NULL);

        XtManageChild (spin);
        XtManageChild (rowcol);
        XtRealizeWidget (toplevel);
        XtAppMainLoop (app);
    }
```

The output of the example is as given in Figure 23-1.

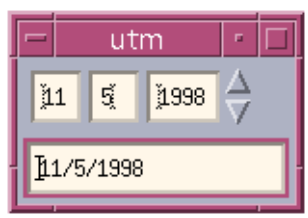


Figure 23-1 Output of the utm program

It should be possible to transfer the formatted SpinBox date to the lower TextField by any of the supported transfer methods.

- selecting any SpinBox TextField, pressing the COPY key, moving the cursor over the lower TextField, then press the PASTE key.
- drag-and-drop out of one of any SpinBox TextField into the lower TextField
- select data in any SpinBox TextField, then press the middle mouse button over the lower TextField.

Summary

The Uniform Transfer Model rationalises the various means of data transfer in the Motif toolkit into a coherent single methodology. The model is simple to understand: a single callback resource at the source of the data, a single callback resource at the destination. The destination callback negotiates with the source in order to determine the best import data format. It then sets up a transfer procedure to perform the actual data import. While a first glance at the callback structures associated with each of these procedures seems to indicate an unwarranted degree of complexity, in actual practice many of the data elements are internal to Motif, and are the means by which it unifies the various types of data transfer in the toolkit. The actual tasks we need to perform in order to transfer data turn out to be quite simple.

In this chapter:
- *Renditions*
- *Render Tables*
- *Tab Lists*
- *An Example*
- *Render Tables and Resource Files*
- *Missing Fonts and Renditions*
- *Summary*

24

Render Tables

This chapter describes the new features introduced in Motif 2.0 for rendering compound strings.

In Motif 1.2, compound strings were rendered with respect to an `XmFontList`, which encapsulates a list of X fonts and font sets. By separating the contents of a compound string from the fonts with which it is associated, it was possible to display the same compound string in different ways in distinct widget contexts, simply by changing the `XmFontList` for each context.

In Motif 2.0 and later, the separation of content from rendition information is continued and extended. The `XmFontList` is however now obsolete. In its place is the notion of a Render Table, represented by the `XmRenderTable` type. Throughout Motif, all the places which used to contain an `XmFontList` resource also now support an `XmRenderTable` equivalent.

A Render table consists of a sequence of `XmRendition` objects. Unlike an `XmFontList`, however, a Rendition object describes rather more than simply the fonts with which a compound string is drawn. A Rendition also describes color, line style, and columnar information. In Motif 2.0 and later, we can have multi-color compound strings inside a multi-column List. A Rendition object is also optimized: fonts can be loaded dynamically at the point of rendition rather than having to be pre-loaded at or before widget creation.

Render Tables and Rendition objects are shareable across contexts, and independently reference-counted. They are also inherited within the widget hierarchy, thus enabling a degree of consistent appearance for the application.

For backwards compatibility, the `XmFontList` is maintained as a type, although internally it is re-implemented as a skeleton `XmRenderTable` containing only font information. Any specified `XmRenderTable` resource for the widget concerned takes precedence.

Renditions

An XmRendition object is a pseudo-widget. Although not a true widget, it has many of the properties of one, namely resource attributes and a resource-style interface. Rendition attributes can also be specified in a resource file.

Creating Renditions

An XmRendition object is created through the routine XmRenditionCreate(), defined as follows:

```
XmRendition XmRenditionCreate ( Widget          widget,
                                XmStringTag     tag,
                                Arg             *argList,
                                Cardinal        argCount)
```

The *widget* parameter does not have to be related in any way to the place where the new XmRendition object is to be applied: it is simply used to find a connection to the X server, so that font and color resources of the XmRendition can be loaded. The *tag* parameter identifies the XmRendition object: compound strings which contain embedded tags are matched against this name when deciding whether to apply the XmRendition to the rendering process of the string. In effect, the XmRendition tag takes the place of the deprecated XmFontList tag. If *tag* is NULL, the value _MOTIF_DEFAULT_LOCALE is assigned. The *argList* and *argCount* parameters specify the resources of the XmRendition object, and are in exactly the same format as the usual argument lists supplied to widget creation routines.

Rendition Resources

XmRendition resources fall mainly into three groups: those which specify the font, those which specify the color, and those which specify the tab or multi-columnar data.

The font is specified through either the XmNfontName of the XmNfont resource. The XmNfontName resource is specified using a standard XLFD font description string. The XmNfont resource can be specified either as an X font (XFontStruct *) or an X font set (XFontSet); whichever you supply, you also need to set the XmNfontType resource to either XmFONT_IS_FONT or XmFONT_IS_FONTSET respectively. Specifying the XmNfont resource means of course that you need to load the font beforehand using XLoadFont(), XLoadQueryFont() or similar, described in Volume 1, *Xlib Programming Manual*. An alternative is to arrange to load the font when it is actually needed for rendering. This is done by specifying the XmNfontName in conjunction with the XmNloadModel resource. If the load model is XmLOAD_IMMEDIATE, the font name is loaded when the rendition is created. Otherwise, with a load model of XmLOAD_DEFERRED, the font is loaded when actually required. The following code fragment creates an XmRendition object which is configured with a deferred font.

```
extern Widget widget;
XmRendition rendition;
Arg args[4];
Cardinal n = 0;

XtSetArg (args[n], XmNfontName, "-*-courier-bold-o-*--*-140-*"); n++;
XtSetArg (args[n], XmNloadModel, XmLOAD_DEFERRED);               n++;
XtSetArg (args[n], XmNfontType, XmFONT_IS_FONT);                 n++;
rendition = XmRenditionCreate (widget, "my_bold_font", args, n);
```

If an XmNfont resource is specified, it takes precedence over any XmNfontName which is also specified. In addition to specifying the font, it is also possible to specify an underline or strike-through style for the font rendition. The resource XmNunderlineType controls any underlining. It has the following possible values:

```
XmDOUBLE_DASHED_LINE          XmDOUBLE_LINE
XmSINGLE_DASHED_LINE          XmSINGLE_LINE
XmNO_LINE
```

Similarly, a strike-through line can be specified using the XmNstrikethruType resource*. This resource has exactly the same range of values as the XmNunderlineType resource.

There are two color resources which can be specified for a rendition. These are XmNrenditionBackground and XmNrenditionForeground, which are Pixel-valued.

Multi-column specification is performed using the XmNtabList resource. Since this subject requires knowledge of other object types, the XmTab and XmTabList, each of which really require a section to themselves, discussion of this resource is reserved for the *Tab Lists* Section later in the chapter.

At this point something needs to be said about the way in which Render Tables work in order to understand the default values associated with each of the resources of an XmRendition object. When a particular compound string component is rendered, any tag associated with the component is matched against renditions in the current render table, starting at the head of the table. Any given rendition in the table may only specify a portion of the rendition information, for example just the foreground color. However, we cannot just draw a color, we also need to draw using a font, and maybe also a line style. There may indeed be renditions in the table which specify these, but they can all have different rendition tags which do not match the current component tag. This is where the notion of inheritance comes in. If any resource in a rendition has the reserved value XmAS_IS, its actual value is calculated by moving back up through the render table. The order of renditions in a render table is therefore important because it determines the order of

* It ought to have been properly named XmNstrikeThroughType, in our humble opinion.

inheritance. The default value for all resources is indeed XmAS_IS, except for the color resources, which default to XmUNSPECIFIED_PIXEL.

The last XmRendition attribute to mention is the XmNtag resource. This is simply the name passed through from the XmRenditionCreate() routine, and it defaults to _MOTIF_DEFAULT_LOCALE. The value NULL is therefore never applied to this resource, although an empty string can be used as the tag. This resource should not be manipulated by the programmer, who should treat the attribute as private to the toolkit.

The following code fragment fully specifies every resource for an XmRendition object. The rendition is unnamed when created, so that it defaults to _MOTIF_DEFAULT_LOCALE. Again, discussion of the XmTabList is reserved until later in the chapter.

```
extern Widget    widget;
XmRendition      rendition;
Arg              args[10];
Cardinal         n = 0;
Pixel            fg = ...; /* Whatever */
Pixel            bg = ...; /* Whatever */
XmNtabList       tlist = ...; /* Discussed later */

XtSetArg (args[n], XmNfontName, “fixed”);          n++;
XtSetArg (args[n], XmNfontType, XmFONT_IS_FONT);   n++;
XtSetArg (args[n], XmNloadModel, XmLOAD_DEFERRED); n++;
XtSetArg (args[n], XmNunderlineType, XmNO_LINE);   n++;
XtSetArg (args[n], XmNstrikethruType, XmNO_LINE);  n++;
XtSetArg (args[n], XmNrenditionForeground, fg);    n++;
XtSetArg (args[n], XmNrenditionBackground, bg);    n++;
XtSetArg (args[n], XmNtabList, tlist);             n++;
rendition = XmRenditionCreate (widget, NULL, args, n);
```

Retrieving Rendition Resources

The attributes of an XmRendition object can be fetched using the routine XmRenditionRetrieve(). This routine has the following signature:

```
void XmRenditionRetrieve ( XmRendition      rendition,
                           Arg              *argList,
                           Cardinal         argCount)
```

Juts as we need to pass the address of a variable when fetching resources using XtGetValues(), so we need to pass an address to fetch a *rendition* resource. The following code fragment outlines the scheme:

```
extern XmRendition   rendition;
Arg                  args[4];
Cardinal             n = 0;
Pixel                fg;
unsigned char        underline;

XtSetArg (args[n], XmNunderlineType, &underline); n++;
XtSetArg (args[n], XmNrenditionForeground, &fg);  n++;
XmRenditionRetrieve (rendition, args, n);
```

All resources for an XmRendition object can be fetched at any time, except for the XmNtag resource, which should not be touched.

Updating Rendition Resources

Updating the rendition resources is also straightforward, and involves the routine XmRenditionUpdate(), which is defined as follows:

```
void XmRenditionUpdate ( XmRendition    rendition,
                         Arg            *argList,
                         Cardinal       argCount)
```

Again, all rendition resources can be dynamically changed except XmNtag. If the XmNfontName resource is changed, the XmNfont value is immediately set to NULL internally irrespective of whether the load model is XmLOAD_DEFERRED or XmLOAD_IMMEDIATE.

Freeing a Rendition

When a rendition object is no longer required, it should be freed using the routine XmRenditionFree(). This function has the prototype:

```
void XmRenditionFree (XmRendition rendition)
```

Note that rendition objects are shareable: we can add the same rendition object to multiple render tables, and remove the rendition from a table, freeing as and when required, because the rendition is reference counted: XmRenditionFree() does not actually free the object until the count is zero. How we add or remove a rendition from a render table is covered in the next section.

Render Tables

A render table, represented by the opaque type XmRenderTable, is a set of rendition objects. An XmRenderTable is not simply an array of XmRendition objects, but a distinct opaque type into which rendition objects must be explicitly merged.

Creating Render Tables

Rendition objects are added to a render table using the function XmRenderTableAddRenditions(), which is defined thus:

```
XmRenderTable
XmRenderTableAddRenditions ( XmRenderTable    old_table,
                             XmRendition      *merge_renditions,
                             Cardinal         new_rendition_count,
                             XmMergeMode      merge_mode)
```

The *old_table* parameter is the table into which we want to add the renditions *merge_renditions*. If *old_table* is NULL, a new render table is formed from the *merge_*

renditions. Otherwise the *merge_renditions* are merged into the *old_table*. Clearly it is possible to have potential conflicts, because a rendition in the *old_table* and in the *merge_renditions* may have the same tag. How to resolve conflicts is determined by the *merge_mode* parameter. If *merge_mode* is XmMERGE_REPLACE, any rendition in old_table with the same tag as a rendition in the *merge_renditions* is ignored: *merge_renditions* take precedence. If *merge_mode* is XmMERGE_SKIP, the *old_table* takes precedence, and renditions are merged from *merge_renditions* if and only if *old_table* does not contain a rendition with a matching tag. These two cases are straightforward: use only the rendition in the *old_table*, or in the new merge list. More complex however are the cases described by the *merge_mode* values XmMERGE_NEW and XmMERGE_OLD. The value XmMERGE_NEW gives precedence to renditions in the *merge_renditions* list, except that if any resources associated with a rendition in the *merge_renditions* list have the value XmAS_IS, the value is copied from any rendition in the *old_table* with a matching tag. XmMERGE_OLD is similar: *old_table* renditions take precedence, but any resource in a rendition in *old_table* which is XmAS_IS takes its value from any rendition in *merge_renditions* with the same tag. The degree of control which can be exercised over the creation and manipulation of render tables using the various merge_mode values is quite complex. The simple case, however, is straightforward: the following specimen code creates a new render table by merging in two newly allocated rendition objects, then applies it to an unspecified widget:

```
extern Widget      widget;
XmRendition        renditions[2];
XmRenderTable      rtable;
Arg                args[4];
Cardinal           n;

n = 0;
XtSetArg (args[n], XmNfontName, “fixed”);           n++;
XtSetArg (args[n], XmNfontType, XmFONT_IS_FONT);    n++;
XtSetArg (args[n], XmNloadModel, XmLOAD_IMMEDIATE); n++;
renditions[0] = XmRenditionCreate (widget, NULL, args, n);

n = 0;
XtSetArg (args[n], XmNfontName, “-*-courier-bold-o-*--*-140-*”); n++;
XtSetArg (args[n], XmNfontType, XmFONT_IS_FONT);                 n++;
XtSetArg (args[n], XmNloadModel, XmLOAD_DEFERRED);               n++;
renditions[1] = XmRenditionCreate (widget, “bold”, args, n);

rtable = XmRenderTableAddRenditions ( NULL, renditions,
                                      XtNumber (renditions),
                                      XmMERGE_NEW);

XtVaSetValues (widget, XmNrenderTable, rtable, NULL);
```

Freeing Render Tables

An XmRenderTable is a dynamically allocated object. We must arrange to free the memory ourselves when we are finished using the table. This is done through the function XmRenderTableFree(), which is simply defined as follows:

```
void XmRenderTableFree (XmRenderTable table)
```

Render tables, like the rendition objects which they contain, are reference-counted internally by Motif. The *table* may not in fact be freed after calling XmRenderTableFree() unless the reference count becomes zero. XmRenderTableFree() does not free the constituent renditions: these need to be deallocated separately in their own right. Note that if we apply a render table to a widget through the XmNrenderTable resource, the widget takes a copy (or rather, updates the reference count) of the table and the renditions which it contains, and so the following code outlines the correct scheme when creating widgets using render tables:

```
extern Widget      parent;
Widget             new_widget;
XmRenderTable      render_table;
XmRendition        renditions[MAX_RENDITIONS];
Arg                args[MAX_ARGS];
int                i, n;

/* Create the renditions we require */
for (i = 0; i < MAX_RENDITIONS; i++) {
    ...
    renditions[i] = XmRenditionCreate (parent, “some_tag”, args, n);
    ...
}

/* Create the Render Table */
render_table = XmRenderTableAddRenditions ( NULL, renditions,
                                            XtNumber (renditions),
                                            XmMERGE_NEW);

/* Create a new widget with the given render table */
/* Or indeed apply to an existing widget using XtSetValues() */
/* Either way, the widget “takes a copy” */
n = 0;
XtSetArg (args[n], XmNrenderTable, render_table); n++;
...
new_widget = XmCreatePushButton (parent, “name”, args, n);

/* Free the allocated space */
/* Firstly, the constituent renditions */
for (i = 0; i < MAX_RENDITIONS; i++) {
    XmRenditionFree (renditions[i]);
}

/* Now the render table itself */
XmRenderTableFree (render_table);
```

Copying Render Tables

A copy of an existing render table can be achieved using the function XmRenderTableCopy(), which is defined thus:

```
XmRenderTable XmRenderTableCopy ( XmRenderTable    old_table,
                                  XmStringTag      *tags,
                                  Cardinal         num_tags)
```

The routine allocates storage for, and returns, a new render table based upon *old_table*. The *tags* parameter can be used as a filter: if *tags* is not NULL, only renditions within *old_table* whose tag is contained within the *tags* array are copied over. Otherwise, all renditions within the *old_table* are cloned. We could, for example, create a new render table which only contains the "fixed" rendition from the code fragment in the *Creating Render Tables* Section earlier in this chapter, using this routine:

```
XmRenderTable    fixed_table;
XmStringTag      tag = "fixed";
...
fixed_table = XmRenderTableCopy (new_table, &tag, 1);
```

Retrieving Renditions from Render Tables

Supposing we want to modify a rendition (or indeed multiple renditions) contained within an arbitrary render table. If we know the tag (or tags) associated with the rendition, we can fetch the renditions using either of the following routines:

```
XmRendition XmRenderTableGetRendition ( XmRenderTable    table,
                                        XmStringTag      tag)
XmRendition *XmRenderTableGetRenditions ( XmRenderTable   table,
                                          XmStringTag     *tags,
                                          Cardinal        num_tags)
```

Fetching a single rendition using XmRenderTableGetRendition() is straightforward enough. The complexity arises where we need to fetch multiple renditions. The array of renditions returned by XmRenderTableGetRenditions() is sized to the list of requested *tags*, and it may contain NULL entries if a given tag does not match anything in the render *table*. There is a one-to-one correspondence between the position of a given tag in the *tags* array and the returned render table list. In other words, the returned renditions are not necessarily in the order in which they appear in *table*, but most certainly are in the order in which the *tags* data appears. Therefore if, say, the first tag in *tags* does not match anything in *table*, the first element in the returned rendition array will be NULL. Both XmRenderTableGetRendition() and XmRenderTableGetRenditions() allocate memory which must be freed at an appropriate point by the programmer. The following code clarifies the situation:

```
extern XmRendertable   render_table;
XmStringTag            *tags[3];
XmRendition            *renditions;
int                    i;
```

```
tags[0] = (XmStringTag) “bold”;
tags[1] = (XmStringTag) “red”;
tags[2] = (XmStringTag) “no such rendition”;

renditions = XmRendertableGetRenditions ( render_table,
                                          tags,
                                          XtNumber (tags));

if (renditions != (XmRendition *) 0) {
    /* The returned renditions array is the same size as the tags array */
    /* Furthermore, the renditions are in tag-array order */
    for (i = 0; i < XtNumber (tags); I++) {
        /* But an entry can be NULL if a given tag does not match */
        if (renditions[i] == (XmRendition) 0) {
            printf (“Warning: no such rendition %s\n”, tags[i]);
        }
        else {
            /* Process the rendition, then free the allocated space */
            ...
            XmRenditionFree (renditions[i]);
        }
    }
    /* Free the allocated array pointer */
    XtFree ((char *) renditions);
}
```

This all assumes that we know the names of the tags in an arbitrary render table. Where this is not the case, we need to query the set of tags in a render table using the routine XmRenderTableGetTags(), which has the following functional prototype:

```
int XmRenderTableGetTags ( XmRenderTable    table,
                           XmStringTag      **tags)
```

The routine returns the number of renditions in the parameter *table*, and places the list of tags at the address specified by the *tags* parameter. The returned array is placed in dynamically allocated memory which the programmer should free at the appropriate point. The tags are in the order in which the renditions occur in the render table. The following routine simply lists all the tags in a given render table:

```
void ListRenditionTags (XmRenderTable table)
{
    int            count, i;
    XmStringTag    *tags;

    count = XmRenderTableGetTags (table, &tags);

    for (i = 0; i < count; i++) {
        printf (“Tag %d is %s\n”, i, tags[i]);
        XtFree (tags[i]);
    }
    XtFree ((char *) tags);
}
```

Removing Renditions

If we know the tag or tags associated with a group of rendition objects in a render table, we can remove the renditions from the table through the routine XmRenderTableRemoveRenditions(), defined as follows:

```
XmRenderTable XmRenderTableRemoveRenditions ( XmRenderTable    old_table,
                                              XmStringTag      *tags,
                                              int              tag_count)
```

The function returns a new render table formed by copying all renditions in *old_table* which do not have a matching tag in the *tags* array. A side effect of calling XmRenderTableRemoveRenditions() is that the reference count associated with *old_table* is decremented. An exception to this is where the *tags* array is NULL: the *old_table* is returned unmodified in any way, so that in effect the routine does nothing. Any renditions which are removed from old_table also have their reference counts decremented by the routine. Which means that if we remove the last rendition from a table, and the rendition is only referenced by this table, both the rendition and the table are freed by the toolkit.

Tab Lists

A TabList is the means by which Motif implements multi-column layout for compound strings. TabLists consist of Tab objects; a Tab is simply an offset across the widget where the compound string is rendered. The Tab represents a location at which to start drawing a compound string segment.

There are four aspects to achieving a multi-column layout.

1. The Tab objects themselves, describing specific locations across a widget. Each Tab specifies a single logical column starting point.
2. The TabList, which is an ordered set of Tabs.The TabList taken as a whole provides the multi-column appearance.
3. Compound string components of type XmSTRING_COMPONENT_TAB; these can be embedded into a compound string, informing the toolkit that the following compound string text component is to be drawn dependent upon the current Tab information.
4. A Render Table, which contains a TabList as a constituent Rendition resource.

Tabs and TabLists are inoperative unless the compound string to be drawn contains the special XmSTRING_COMPONENT_TAG component.

Tabs

Tabs are implemented through the XmTab object. The XmTab object is an opaque handle onto a structure which describes an offset across the widget where a compound string is

rendered. Each XmTab is a shareable, reference counted resource which can be used in multiple tablists.

Creating an XmTab

An XmTab is created using the function XmTabCreate(), which is defined as follows:

```
XmTab XmTabCreate ( float             value,
                    unsigned char     units,
                    XmOffsetModel     offset_model,
                    unsigned char     alignment,
                    char              *decimal)
```

The *value* parameter is interpreted in terms of *units*, which can be one of the following:

```
XmPIXELS
XmMILLIMETERS               Xm100TH_MILLIMETERS
XmINCHES                    Xm1000TH_INCHES
XmCENTIMETERS
XmPOINTS                    Xm100TH_POINTS
XmFONT_UNITS                Xm100TH_FONT_UNITS
```

The *offset_model* parameter determines whether the tab position is an absolute distance across the widget where rendering is to take place (XmABSOLUTE), or whether the tab position is calculated relative to the previous tab stop (XmRELATIVE). The *alignment* parameter specifies how text is aligned with respect to the tab location. Only XmALIGNMENT_BEGINNING is implemented as of Motif 2.1.10. The *decimal* parameter specifies the multi-byte character in the current locale which is used as a decimal point. This is currently unused.

The following code fragment creates a tab stop 1.5 inches from the start of compound string rendering:

```
XmTab tab;

tab = XmTabCreate ( (float) 1.5,
                    XmINCHES,
                    XmABSOLUTE,
                    XmALIGNMENT_BEGINNING,
                    ".");
```

Freeing an XmTab

When an XmTab is no longer required, the memory associated with the object should be freed using the routine XmTabFree(), which is defined as follows:

```
void XmTabFree (XmTab tab)
```

Fetching XmTab values

The values associated with an XmTab object can be fetched using the routine XmTabGetValues(). This routine is defined as follows:

```
float XmTabGetValues ( XmTab              tab,
                       unsigned char      *units,
                       XmOffsetModel      *model,
                       unsigned char      *alignment,
                       char               **decimal)
```

The interpretation of each of the parameters is directly analogous to XmTabCreate(), except that in each case an address is required to hold the returned data. The *tab* parameter is the object for which the values are required. The following code fragment outlines the basic usage of the routine:

```
extern XmTab     tab;
XmOffsetModel    offset_model;
unsigned char    units;
unsigned char    alignment;
char             *decimal;
float            value;

value = XmTabGetValues ( tab,
                         &units,
                         &offset_model,
                         &alignment,
                         &decimal);
```

Note that the returned data at the *decimal* address directly points into the *tab* object structure: *decimal* does *not* contain a dynamically allocated copy, and so it should neither be modified nor freed by the programmer.

Setting the XmTab value

The value associated with a tab object can be modified using the routine XmTabSetValues(). There is no routine available to modify the units, alignment, offset model, or decimal associated with a tab once it has been created. To do these operations, you need to explicitly remove the tab object from the tab list concerned, and create a new tab with the required attributes from scratch. XmTabSetValues() is defined as follows:

```
void XmTabSetValues (XmTab tab, float value)
```

TabLists

A Tab List, represented by the opaque type XmTabList, is an ordered set of tab objects. An XmTabList is not simply an array of XmTab objects, but a distinct opaque type into which tab objects must be explicitly merged.

Creating TabLists

XmTab objects can be added to an XmTabList using the convenience routine XmTabListInsertTabs(), which is defined as follows:

```
XmTabList XmTabListInsertTabs ( XmTabList    tablist,
                                XmTab        *tabs,
```

```
                                    Cardinal     tab_count,
                                    int          position)
```

The parameter *tablist* can be NULL, which means that a new tab list is to be formed out of the XmTab objects specified through the *tabs* array. If *tabs* is NULL, the original *tablist* is returned unmodified. This means that in effect that there is no way to create an XmTabList independently of a set of XmTab objects. The *position* parameter specifies where the *tabs* are to be inserted into *tablist*. If *position* is 0, the *tabs* are inserted at the head of *tablist*. If *position* is 1, *tabs* are inserted after the current first tab of *tablist*, and so forth. To insert using the end of the tablist, position should be specified as a negative quantity.* A negative position inserts XmTab objects in reverse order at the end of the XmTabList, such that the first new tab in the *tabs* array becomes the last tab in the newly formed tab set.

Much of the implementation of XmTabListInsertTabs() is complex and probably over-engineered. The simple case, however, is as given in the following code fragment. This creates two XmTab objects, and forms a new XmTabList from them:

```
XmTab        tabs[2];
XmTabList    tabList;

tabs[0] = XmTabCreate ((float) 1.0, XmINCHES, XmABSOLUTE,
                                XmALIGNMENT_BEGINNING, ".");
tabs[1] = XmTabCreate ((float) 1.5, XmINCHES, XmRELATIVE,
                                XmALIGNMENT_BEGINNING, ".");
tabList = XmTabListInsertTabs (NULL, tabs, XtNumber (tabs), 0);
```

Freeing TabLists

The XmTabList object uses dynamically allocated memory. This should be reclaimed when no longer in use through the routine XmTabListFree(), which is defined as follows:

```
void XmTabListFree (XmTabList tablist)
```

Manipulating Tabs in a TabList

The following routines are available for manipulating the XmTab elements in an XmTabList:

```
XmTabList    XmTabListCopy (XmTabList tablist, int offset, Cardinal count)†
XmTab        XmTabListGetTab (XmTabList tablist, Cardinal position)
XmTabList    XmTabListRemoveTabs ( XmTabList     tablist,
                                   Cardinal      *positions,
                                   Cardinal      position_count)
```

* This is not consistent with other Motif insertion routines, where zero is generally taken to mean the end of the given list of objects, and negative positions are disallowed.

† Erroneously listed as XmTabListTabCopy() in Volume 6B, *Motif Reference Manual*. Humble Apologies!

```
XmTabList    XmTabListReplacePositions ( XmTabList    tablist,
                                         Cardinal     *positions,
                                         XmTab        *tabs,
                                         Cardinal     tab_count)
int          XmTabListTabCount (XmTabList tablist)
```

XmTabListCopy() copies *count* XmTab objects from the XmTabList specified by the *tablist* parameter, starting with the tab at the position specified by *offset*. If *offset* is zero, tabs are copied from the start of the list. If *count* is zero, all tabs from *offset* to the end of the tablist are copied. If the *offset* is negative, tabs are copied in reverse order from the end of the *tablist*. Copying an entire tablist is therefore simply a matter of calling the following code:

```
extern XmTabList old_tablist;
XmTabList full_copy = XmTabListCopy (old_tablist, 0, 0);
```

An XmTab object can be fetched from a tab list using the routine XmTabListGetTab(). The routine simply fetches the tab at the designated *position* within the given *tablist*. The first tab in the list is at position zero. The routine returns a *copy* of the XmTab object at *position*, and it is the responsibility of the programmer to free the allocated memory at a suitable point using XmTabFree().

If you wanted to fetch the last XmTab object in a tablist, you would need to know the number of tabs in the list in the first place. The routine XmTabListTabCount() can be used for this: it simply returns the number of tabs in the specified *tablist*. The following code therefore fetches the last XmTab object:

```
extern XmTabList tablist;
XmTab last_tab;
int count;

count = XmTabListTabCount (tablist);

if (count > 0)
    /* TabLists offset from zero */
    last_tab = XmTabListGetTab (tablist, count - 1);
...
/* Remember to reclaim the memory: we are returned a copy */
XmTabFree (last_tab);
```

XmTab objects can be removed from a tablist using the routine XmTabListRemoveTabs(). It creates and returns a new XmTabList formed out of an existing *tablist*, but with tabs at designated *positions* excluded. The original *tablist* has its reference count internally decremented internally by the routine.

XmTabListReplacePositions() can be used to substitute a set of XmTab objects within *tablist.* The *positions* parameter specifies an array of offsets representing the locations where tabs are to be replaced. The *tabs* parameter is the set of new tabs to be merged into the *tablist*. There is a one-to-one correspondence between the offsets in the *positions* parameter and the *tabs* specifier. That is, the *nth* XmTab in the *tabs* list is

placed at the offset designated by the *nth* offset in the *positions* array. For example, the following code replaces the third and fifth tabs in an unspecified tablist:

```
extern XmTabList   old_tablist;
XmTabList          new_tablist;
XmTab              new_tabs[2];
Cardinal           positions[2];

new_tabs[0] = XmTabCreate (...);
new_tabs[1] = XmTabCreate (...);

positions[0] = 2; /* The third position - offsets are from zero */
positions[1] = 4; /* The fifth position */

new_tablist = XmTabListReplacePositions ( old_tablist,
                                          positions,
                                          new_tabs,
                                          XtNumber (new_tabs));
```

Using TabLists

TabLists are used by specifying them as an attribute of a rendition in a render table. If we require a multi-column layout, we need to make sure that the render table which is being used to render our compound strings contains a TabList.* The following code fragment outlines the general scheme of things:

```
XmTab               tabs[MAX_TABS];
XmTabList           tablist;
XmRendition         renditions[MAX_RENDITIONS];
XmRenderTable       rendertable;
Arg                 args[MAX_ARGS];
int                 i, n;
extern Widget       widget;
...
/* Create the XmTab objects */
tabs[i] = XmTabCreate ((float) 1.5, XmINCHES, XmABSOLUTE,
XmALIGNMENT_BEGINNING, ".");
...
/* Create the XmTabList from the XmTab objects */
tablist = XmTabListInsertTabs (NULL, tabs, XtNumber (tabs), 0);
...
/* Create an XmRendition that uses the XmTabList */
/* Other XmRendition attributes are ignored here */
n = 0;
XtSetArg (args[n], XmNtabList, tablist); n++;
...
renditions[i] = XmRenditionCreate (widget, "rendition tag", args, n);
...
/* Create an XmRenderTable which uses the XmRendition objects */
```

* The exception to the rule is the Container widget, which has an `XmNdetailTabList` resource independent of its `XmNrendertable` resource.

```
rendertable = XmRenderTableAddRenditions ( NULL,
                                           renditions,
                                           XtNumber (renditions),
                                           XmMERGE_NEW);
...
/* Specify the XmRenderTable for the widget concerned */
XtVaSetValues (widget, XmNrenderTable, rendertable, NULL);
...
/* The compound strings associated with widget are now */
/* drawn in multi-column format if the strings contain */
/* embedded XmSTRING_COMPONENT_TAB components */
...
/* Free the memory used above. */
/* The widget takes a copy of the XmRenderTable */
/* Or rather, increases the reference count */
/* The XmRendition takes a copy of the XmTabList */
/* (increases the reference count) */
for (i = 0; i < XtNumber (tabs); i++) {
    XmTabFree (tabs[i]);
}
XmTabListFree (tablist);

for (i = 0; i < XtNumber (renditions); i++) {
    XmRenditionFree (renditions[i]);
}

XmRenderTableFree (rendertable);
...
```

An Example

The code in Example 24-1 creates a multi-column, multi-color, multi-font List widget. It brings together all the threads of this chapter by utilizing fully the tab, tab list, rendition, and render table functionality as described above.

Example 24-1 The rendered_list.c program

```
/* rendered_list.c: illustrates all the features of
** render tables and renditions by creating a
** multi-column, multi-font, multi-color List widget.
*/

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

/* ConvertStringToPixel()
** A utility function to convert a color name to a Pixel
*/
Pixel ConvertStringToPixel (Widget widget, char *name)
{
    XrmValue from_value, to_value; /* For resource conversion */
```

```
    from_value.addr = name;
    from_value.size = strlen(name) + 1;
    to_value.addr = NULL;
    XtConvertAndStore (widget,
    XmRString, &from_value,
    XmRPixel, &to_value);

    if (to_value.addr) {
        return (*((Pixel*) to_value.addr));
    }

    return XmUNSPECIFIED_PIXEL;
}

/*
** A convenient structure to hold the data
** for creating various renditions
*/
typedef struct RenditionData_s
{
    char       *tag;
    char       *color;
    char       *font;
} RenditionData_t;

#define MAX_COLUMNS4

RenditionData_t rendition_data[MAX_COLUMNS] =
{
    { “one”, “red”, “fixed” },
    { “two”, “green”,
            “-adobe-helvetica-bold-r-normal--10-100-75-75-*-*-iso8859-1” },
    { “three”, “blue”, “bembo-bold” },
    { “four”, “orange”,
            “-adobe-*-medium-i-normal--24-240-75-75-*-*-iso8859-1” }
};

/*
** Arbitrary data to display in the List
*/
static char *poem[] =
{
    “Mary”, “had a”, “little”, “lamb”,
    “Its”, “fleece”, “was white”, “as snow”,
    “And”, “everywhere that”, “Mary”, “went”,
    “The”, “lamb was”, “sure”, “to follow”,
    (char *) 0
};

/*
** CreateListData(): routine to convert the
** poem into an array of compound strings
*/
XmStringTable CreateListData (int *count)
```

```
{
    XmStringTable    table = (XmStringTable) 0;
    int              line = 0;
    int              column = 0;
    int              index = 0;
    XmString         entry = (XmString) 0;
    XmString         row = (XmString) 0;
    XmString         tmp = (XmString) 0;
    XmString         tab;

    tab = XmStringComponentCreate (XmSTRING_COMPONENT_TAB, NULL, 0);

    while (poem[index] != (char *) 0) {
        /* create a compound string, using the rendition tag */
        entry = XmStringGenerate ((XtPointer) poem[index],
                                  NULL,
                                  XmCHARSET_TEXT,
                                  rendition_data[column].tag);

        if (row != (XmString) 0) {
            tmp = XmStringConcat (row, tab);
            XmStringFree (row);
            row = XmStringConcatAndFree (tmp, entry);
        }
        else {
            row = entry;
        }

        ++column;

        if (column == MAX_COLUMNS) {
            if (table == (XmStringTable) 0) {
                table = (XmStringTable) XtMalloc ((unsigned)
                sizeof (XmString));
                }
            else {
                table = (XmStringTable) XtRealloc ((char *) table,
                (unsigned) (line + 1) * sizeof (XmString));
            }

            table[line++] = row;
            row = (XmString) 0;
            column = 0;
        }

        index++;
    }

    XmStringFree (tab);

    table[line] = (XmString) 0;

    *count = line;
```

```
    return table;
}

main (int argc, char *argv[])
{
    Widget             toplevel, rowcol, list;
    XtAppContext       app;
    Arg                args[10];
    XmTab              tabs[MAX_COLUMNS];
    XmTabList          tablist;
    XmRendition        renditions[MAX_COLUMNS];
    XmRenderTable      rendertable;
    XmStringTable      xmstring_table;
    int                xmstring_count;
    Pixel              pixels[MAX_COLUMNS];
    int                n, i;

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

    /* Create some colors */
    for (i = 0; i < MAX_COLUMNS; i++) {
        pixels[i] = ConvertStringToPixel (toplevel,
        rendition_data[i].color);
    }

    /* Create tab stops for columnar output */
    for (i = 0; i < MAX_COLUMNS; i++) {
        tabs[i] = XmTabCreate ((float) 1.5,
                               XmINCHES,
                               ((i == 0) ? XmABSOLUTE : XmRELATIVE),
                               XmALIGNMENT_BEGINNING,
                               ".");
    }

    /* Create a tablist table which contains the tabs */
    tablist = XmTabListInsertTabs (NULL, tabs, XtNumber (tabs), 0);

    /* Create some multi-font/color renditions, and use the tablist */
    /* This will be inherited if we use it on the first rendition */
    for (i = 0; i < MAX_COLUMNS; i++) {
        n = 0;

        if (i == 0) {
            XtSetArg (args[n], XmNtabList, tablist); n++;
        }

        XtSetArg (args[n], XmNrenditionForeground, pixels[i]);   n++;
        XtSetArg (args[n], XmNfontName, rendition_data[i].font); n++;
        XtSetArg (args[n], XmNfontType, XmFONT_IS_FONT);         n++;
        renditions[i] = XmRenditionCreate (toplevel,
                                    rendition_data[i].tag,
```

```
                                    args, n);
    }

    /* Create the Render Table */
    rendertable = XmRenderTableAddRenditions ( NULL,
                                               renditions,
                                               XtNumber (renditions),
                                               XmMERGE_NEW);

    /* Create the multi-column data for the list */

    xmstring_table = CreateListData (&xmstring_count);

    /* Create the List, using the render table */
    n = 0;
    XtSetArg (args[n], XmNrenderTable, rendertable);             n++;
    XtSetArg (args[n], XmNitems, xmstring_table);                n++;
    XtSetArg (args[n], XmNitemCount, xmstring_count);            n++;
    XtSetArg (args[n], XmNwidth, 400);                           n++;
    XtSetArg (args[n], XmNvisibleItemCount, xmstring_count + 1); n++;
    list = XmCreateScrolledList (rowcol, "list", args, n);
    XtManageChild (list);

    /* Free the memory now the widget has the data */
    /* First, the compound strings */
    for (i = 0; i < xmstring_count; i++)
        XmStringFree (xmstring_table[i]);
    XtFree ((char *) xmstring_table);

    /* Secondly, the XmTab objects */
    for (i = 0; i < XtNumber (tabs); i++)
        XmTabFree (tabs[i]);

    /* Thirdly, the XmTabList object */
        XmTabListFree (tablist);

    /* Fourthly, the XmRendition objects */
    for (i = 0; i < XtNumber (renditions); i++)
        XmRenditionFree (renditions[i]);

    /* Lastly, the XmRenderTable object */
    XmRenderTableFree (rendertable);

    XtManageChild (rowcol);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}
```

The output from the program is given in Figure 24-1. Of course, the color details of the program are lost on a black-and-white printed page, although it is clear even in grey scale

that the List contains differently colored compound string segments. The multi-font and multi-column aspects of the program are fully in evidence.

Figure 24-1 The output of rendered_list

Render Tables and Resource Files

RenderTables, Renditions, Tabs, and TabLists can all be specified in resource files. Resource converters are installed for each of the various attribute types automatically by the Motif toolkit. Since the rendition objects described in this chapter are implemented very much as pseudo-widgets, the resource specifications required to describe the objects are natural and familiar. Only the specification of Tabs and Tab lists present new formats. Since renditions contain tab lists, we should start from the bottom up and describe the specification of a tab list in a resource file first.

Tabs and TabList

In a resource file, a tab list is represented by a comma-separated list of tab specifications. Each tab specification has the formal syntax:

```
tab := float [ WHITESPACE units ]
float := [ sign ] [[ DIGIT]*. ]DIGIT+
sign := +
```

Less formally, a tab is represented by a floating point number, optionally preceded by a plus sign, and optionally followed by a units description. The following are examples of correctly formed tab specifications:

```
12.4 in
+1in
+14.56 mm
0.15 font_units
+116.0 pix
```

The presence of a plus sign indicates that the tab is relative to the previous tab stop in the current tab list. If no plus sign is given, the tab is interpreted as an absolute value across the widget wherever it is used. In code, we would write the following in order to duplicate the effect of the tab resource specifications given above:

```
XmTab a = XmTabCreate (12.4, XmINCHES, XmABSOLUTE, ...)
XmTab b = XmTabCreate (1.0, XmINCHES, XmRELATIVE, ...)
XmTab c = XmTabCreate (14.56, XmMILLIMETERS, XmRELATIVE, ...)
```

The units descriptions are in the same format as a normal widget XmNunitType resource. That is, the following are acceptable to the resource converter:

```
pixels            pix         pixel
inches            in          inch
centimeters       cm          centimeter
millimeters       mm          millimeter
points            pt          point
font_units        fu          font_unit
```

Note that the converter does not directly handle the fractional unit types (Xm1000TH_INCHES, Xm1000TH_MILLIMETERS, Xm100TH_POINTS, Xm100TH_FONT_UNITS). To specify these in a resource file, you need to divide the float value by the appropriate quantity. For example, although we can write in code...

```
XmTab tab = XmTabCreate (150.0, Xm1000TH_INCHES, XmABSOLUTE, ...)
```

... in a resource file, we need to specify the following:

```
0.15 in
```

A tab list as specified in a resource file is a simple comma separated set of tab specifications. The following are examples:

```
*tabList: 1.5in, +1.5in, +1.5in
XApplication*my_rendition.tabList: 220.0 mm, 450.0 mm, +1.0 in
```

Rendition Resources

The rendition object is a pseudo-widget, and nowhere is this more true than in specifying external rendition resources. We simply use the rendition tag as the X resource key, and thereafter pretend the object is a normal widget. For example, the following resources specify a rendition whose tag is “my_rendition”. The rendition is filly loaded. As is normal, the Xm prefix is stripped off any enumerated types when specifying resource file entries.

```
*my_rendition.fontName: fixed
*my_rendition.fontType: FONT_IS_FONT
*my_rendition.loadModel: LOAD_DEFERRED
*my_rendition.renditionForeground: red
*my_rendition.renditionBackground: blue
*my_rendition.strikethruType: AS_IS
*my_rendition.underlineType: SINGLE_LINE
*my_rendition.tablist: 1in, +1.5in
```

RenderTable Resources

Just as a tablist is specified by a comma-separated list of tabs, a render table is specified externally to code by a list of rendition tags. The list of rendition tags can be whitespace or comma separated. The following are valid render table specifications:

```
*XmList.renderTable: renditionA, my_rendition
*.renderTable: r1 r2 r3, r4
```

Missing Fonts and Renditions

In Motif 1.2, whenever an attempt was made to render a compound string, the tags associated with the compound string segments were matched against tags within the XmFontList which was being used to display the string. Whenever there occurred a mismatch, if a segment referred to a fontlist tag which was not satisfied by the current XmFontList, the toolkit would simply use the first font in the font list. There was nothing that the programmer could dynamically do at that point to rectify the situation, either by directly tendering an alternative font, or indirectly by querying the user for a preferred alternative.

In Motif 2.1, the situation is entirely different. Whenever an attempt is made to render a compound string and there is a mismatch between the required string tags and the current render table, the toolkit now invokes callbacks. In the callback, the programmer can take whatever action is necessary in order to find an appropriate solution to the rendering problem. Once an appropriate font or rendition is found, the programmer simply returns the new font or rendition to the toolkit through elements in the callback structure.

The new callbacks are implemented in the XmDisplay object. There are two callback types, each of which is invoked depending upon the nature of the problem to hand: the XmNnoRenditionCallback, and the XmNnoFontCallback.

The XmNnoRenditionCallback is invoked when an attempt is made by the toolkit to draw a compound string segment, and no matching rendition can be found in the current render table.

The XmNnoFontCallback is called by the toolkit if an XmRendition object refers to a font name, and that font cannot be located on the system. Recall that fonts in a rendition object can have the load model XmLOAD_DEFERRED: this means that the potential exists for mis-specifying a font name when creating a rendition. The effects of the error are not apparent until much later, when an attempt is made to actually render using the rendition concerned. This is different to the Motif 1.2 handling of fonts because the fonts within an XmFontList are loaded immediately on creation of the object.

The XmNnoFontCallback is therefore an internal error in the specification of the attributes of a rendition. The XmNnoRenditionCallback indicates a flaw in either the render table itself, or in the tags associated with segments in the compound string.

Callback Structure

Both the XmNnoFontCallback and the XmNnoRenditionCallback are passed the following structure by the Motif toolkit when invoked:

```
typedef struct
{
    int                reason;
    XEvent             *event;
    XmRendition        rendition;
    char               *font_name;
    XmRenderTable      render_table;
    XmStringTag        tag;
} XmDisplayCallbackStruct;
```

Not all the fields of the structure are applicable to each of the callback types. The `render_table` and `tag` elements are only applicable to `XmNnoRenditionCallback` callbacks, and the `rendition` and `font_name` fields are only applicable to `XmNnoFontCallback` routines. The `reason` field will be either `XmCR_NO_FONT` or `XmCR_NO_RENDITION`.

XmNnoFontCallback

When an attempt is made to render using a rendition which refers to a font name which cannot be located, the `XmNnoFontCallback` is called. The programmer can deduce which font could not be loaded from the `font_name` element of the callback structure. She can remedy the problem simply by choosing an alternative font, and then updating the `rendition` element. The following code fragment outlines the basic scheme of operations:

```
void no_font_callback ( Widget       widget,
                        XtPointer    client_data,
                        XtPointer    call_data)
{
    XmDisplayCallbackStruct *dptr = (XmDisplayCallbackStruct *) call_data;
    Arg args[4];
    int n;

    printf ("Warning: could not load font %s\n", dptr->font_name);

    /* Just use a simple alternative */
    /* A better algorithm would try and find */
    /* a close match to the missing font type */

    n = 0;
    XtSetArg (args[n], XmNfontName, "fixed");           n++;
    XtSetArg (args[n], XmNfontType, XmFONT_IS_FONT);    n++;
    XtSetArg (args[n], XmNloadModel, XmLOAD_IMMEDIATE); n++;

    XmRenditionUpdate (dptr->rendition, args, n);
}

...
extern Display *display;
Widget xmdisplay = XmGetXmDisplay (display);
XtAddCallback (xmdisplay, XmNnoFontCallback, no_font_callback, NULL);
...
```

XmNnoRenditionCallback

If a compound string segment refers to a rendition tag which is absent from the current render table, the XmNnoRenditionCallback is called. Again, the programmer can deduce the missing information from the callback structure: the tag element identifies the missing rendition. The programmer simply has to update the render_table element field of the structure by merging a new rendition. The only issue is one of memory management: the render_table element is allocated for the purpose of the callback, and so the programmer should make sure that the old value of render_table is freed (reference count reduced) if she modifies the element. The following code fragment shows how this can be done:

```
void no_rendition_callback ( Widget       widget,
                             XtPointer    client_data,
                             XtPointer    call_data)
{
    XmDisplayCallbackStruct *dptr = (XmDisplayCallbackStruct *) call_data;
    XmRendition new_rendition;
    XmRenderTable new_table;
    Arg args[4];
    int n;

    printf ("Warning: could not find a rendition with tag %s\n", dptr->tag);

    /* Again, this algorithm is slightly defective: */
    /* we ought to try and deduce a sensible rendition */
    /* given whatever clues that the missing tag can tell us */

    n = 0;
    XtSetArg (args[n], XmNfontName, "fixed");
    XtSetArg (args[n], XmNfontType, XmFONT_IS_FONT);    n++;
    XtSetArg (args[n], XmNloadModel, XmLOAD_IMMEDIATE); n++;

    /* Make sure we create the rendition using the missing tag */
    new_rendition = XmRenditionCreate (widget, dptr->tag, args, n);

    /* Merge into the current render table */
    new_table = XmRenderTableAddRenditions (dptr->render_table,
                                            &new_rendition, 1,
                                            XmMERGE_NEW);

    /* Decrement the old table reference count */
    XmRenderTableFree (dptr->render_table);

    /* Reset the element */
    dptr->render_table = new_table;
}

...
extern Display *display;
Widget xmdisplay = XmGetXmDisplay (display);
XtAddCallback (xmdisplay,
```

```
    XmNnoRenditionCallback, no_rendition_callback, NULL);
    ...
```

Nota Bene

An important difference exists between Motif 1.2 and Motif 2.1 behavior in this respect: whereas Motif 1.2 uses the first font in an `XmFontList` by default whenever there is a mismatch between compound string segment and font list tags, in Motif 2.1 there is no default behavior. That is, if a programmer fails to provide an alternative font or rendition through the appropriate callback then the given compound string is simply not drawn at all.

Summary

Render tables are shareable resources. They extend the separation between the abstract representation of a compound string and the way it is rendered which was implicit in the Motif 1.2 `XmFontList`. What the Render Table provides is the ability to specify not only font information, but also color and multi-column detail, and as such they offer far greater control over the presentation and layout of compound strings.

A side effect of the XmFontList deprecation is a greater degree of compatibility with the underlying X Internationalization modules, because the fonts contained within the renditions of a render table are now properly only held internally in the form of `XFontStruct` or X `FontSet` data.

In this chapter:
- *Internationalized Text Output*
- *Creating Compound Strings*
- *Manipulating Compound Strings*
- *Parse Tables*
- *Rendering Compound Strings*
- *Summary*

25

Compound Strings

This chapter describes Motif's technology for encoding font changes and character directions in the strings that are used by almost all of the Motif widgets.

Compound strings are designed to address two issues frequently encountered by application designers: the use of foreign character sets to display text in other languages and the use of multiple fonts to render text. With the addition of internationalized string rendering capabilities in X11R5 onwards, the use of compound strings for internationalization purposes is theoretically no longer necessary. However, the Motif widget set still uses compound strings extensively, so applications have no choice but to create them to display text.

From Motif 2.0 onwards, the `XmFontList` is obsolete, and is replaced by the `XmRenderTable`. RenderTables are fully described in Chapter 24; briefly, a RenderTable describes a complete style by which a compound string can be rendered. In Motif 2.0 onwards, not only can we associate different fonts with a compound string, we can also render the various segments of the string in different colors, underline styles, and so forth. These and other aspects of rendering compound strings are described in the following sections.

Internationalized Text Output

The internationalization features in X11R5 onwards are based on the ANSI-C locale model. Under this model, an application uses a library that reads a customization database at run-time to get information about the user's language environment. An Xt-based application can establish its language environment (or locale) by registering a language procedure with `XtSetLanguageProc()`, as described in Chapter 18, *Text Widgets*. The language procedure returns a language string that is used by `XtResolvePathname()` to find locale-specific resource files. See Volume 4, *X Toolkit Intrinsics Programming Manual*, for more information on the localization of the resource database.

One of the important characteristics of a language environment is the *encoding* that is used to represent the *character set* for the particular language. In X, character set simply refers to a set of characters, while an encoding is a numeric representation of these characters.[*] A

charset (not the same as a character set) is an encoding in which all of the characters have the same number of bits. Charsets are often defined by standards bodies such as the International Standards Organization (ISO). For example, the ISO Latin-1 charset (ISO8859-1) defines an encoding for the characters used in all Western languages. The first half of Latin-1 is standard ASCII, while the second half (with the eighth bit set) contains accented characters needed for Western languages other than English. Character 65 in ISO Latin-1 is an uppercase "A", while 246 is a lowercase "o" with an umlaut (ö).

However, not all languages can be represented by a single charset. Japanese text commonly contains words written using the Latin alphabet, as well as phonetic characters from the *katakana* and *hirigana* alphabets, and ideographic *kanji* characters. Each of these character sets has its own charset; the phonetic and Latin charsets are 8-bits wide, while the ideographic charset is 16-bits wide. The charsets must be combined into a single encoding for Japanese text, so the encoding uses *shift sequences* to specify the character set for each character in a string.

Strings in an encoding that contains shift sequences and characters with non-uniform width can be stored in a standard `NULL`-terminated array of characters; this representation is known as a multibyte string. Strings can also be stored using a wide-character type in which each character has a fixed size and occupies one array element in the string. The text output routines in X11R5 support both multibyte and wide-character strings. To support languages that use multiple charsets, X developed the `XFontSet` abstraction for its text output routines. An `XFontSet` contains all of the fonts that are needed to display text in the current locale. The new text output routines work with font sets, so they can render text for languages that require multiple charsets. See Volume 1, *Xlib Programming Manual*, for more information on internationalized text output.

With the addition of these features in X, a developer can write an internationalized application without using the internationalization features provided by compound strings. In an internationalized application, strings are interpreted using the encoding for the current locale. To support a number of locales, the application needs to store string data in separate files from the application code. The application must provide a separate file for each of the locales supported, so that the program can read the appropriate file during localization.

However, since most Motif widgets use compound strings for representing textual data, a Motif application has to use compound strings to display text. As we describe compound strings in this chapter, we'll discuss how to use them so as not to interfere with the lower-level X internationalization features. However, since Rendition objects in Motif 2.x only refer to fonts in the form of an X FontStruct or X FontSet, the toolkit is rather more compatible with the X11R5 internationalization features than previous versions of the toolkit, and so this is less of an issue.

---

* Both of these terms are different from the definition of a font, which is a collection of glyphs used to represent the characters in an encoding.

Creating Compound Strings

Almost all of the Motif widgets use compound strings to specify textual data. Labels, PushButtons, and Lists, among others, all require their text to be given in compound string format, whether or not you require the additional flexibility compound strings provide. The only widgets that don't use compound strings are the Text and TextField widgets*. As a result, you cannot use the compound string techniques for displaying text using multiple fonts. However, these widgets do support internationalized text output, so they can display text using multiple character sets. For information on the internationalization capabilities of the Text and TextField widgets, see the *Text Widget Internationalization* Section of Chapter 18.

A compound string (`XmString`) is made of three components: a tag, a direction, and text. The tag is an arbitrary name that the programmer can use to associate a compound string with particular rendition data. In Motif 1.1, the tag was referred to as a character set. Since the tag doesn't necessarily specify a character set, Motif 1.2 referred to the entity as a font list tag. In Motif 2.0 and later, the font list is obsolete, and has been replaced with the render table. A render table consists of rendition objects: the tag now refers to the name assigned to an individual rendition object. One of the things which a rendition object contains is a font. Renditions and Render Tables can be shared between, and inherited from, other widgets.

An application can create a compound string that uses multiple fonts by concatenating separate compound strings with different rendition tags to produce a single compound string. Concatenating compound strings with different renditions is a powerful way to create graphically interesting labels and captions. More importantly, because renditions and render tables are loosely bound to compound strings via resources, you can dynamically associate new renditions with a widget while an application is running and effectively change text styles on the fly.

The Simple Case

Many applications only need to use compound strings to specify various textual resources. In this case, all that is needed is a routine that converts a standard C-style `NULL`-terminated text string into a compound string. The most basic form of conversion can be done using the `XmStringCreateLocalized()` function, as demonstrated in examples throughout this book. This routine takes the following form:

```
XmString XmStringCreateLocalized (char *text)
```

The *`text`* parameter is a common C `char` string. The value returned is of type `XmString`, which is an opaque type to the programmer.

* The Motif 2.0 CSText widget, which did support compound strings, was removed from Motif 2.1.

XmStringCreateLocalized() creates a compound string in the current locale, which is specified by the tag XmFONTLIST_DEFAULT_TAG. This routine interprets the *text* string in the current locale when creating the compound string. If you are writing an internationalized application that needs to support multiple locales, you should use XmStringCreateLocalized() to create compound strings. The routine allows you to take advantage of the lower-level internationalization features of X.

Most applications specify compound string resources in resource files. This technique is appropriate for an internationalized application, as there can be a separate resource file for each language environment that is supported. Motif automatically converts all strings that are specified in resource files into compound strings using the tag XmFONTLIST_DEFAULT_TAG, so the strings are handled correctly for the current locale. If an application needs to create a compound string programmatically, it should use XmStringCreateLocalized() to ensure that the string is interpreted in the current locale. The examples in other chapters of this book use XmStringCreateLocalized() to demonstrate the appropriate technique, even though the examples are only designed to work in the C locale.

With XmStringCreateLocalized(), you cannot explicitly specify the tag or the string direction that is used for the compound string, and in Motif 1.2, the string cannot have multiple lines.

XmStringCreateLocalized() allocates memory to store the compound string that is returned. Widgets that have compound string resources always allocate their own space and store copies of the compound string values you give them. When you are done using a compound string to set widget resources, you must free it using XmStringFree(). The following code fragment demonstrates this usage:

```
XmString   str = XmStringCreateLocalized ("Push Me");
Widget     push_b;
Arg        args[...];
int        n = 0;
...
XtSetArg (args[n], XmNlabelString, str); n++;
push_b = XmCreatePushButton (parent, "widget_name", args, n);
XmStringFree (str);
...
```

The process of creating a compound string, setting a widget resource, and then freeing the string is the most common use of compound strings. However, this process involves quite a bit of overhead, as memory operations are expensive. Memory is allocated by the string creation function and again by the internals of the widget for its own storage, and then your copy of the string must be deallocated.

The programmatic interface to the string creation process can also be achieved by using the XtVaTypedArg feature in Xt. This special resource can be used in variable argument list specifications for functions such as XtVaCreateManagedWidget() and

XtVaSetValues(). It allows you to specify a resource using a convenient type and have Xt do the conversion for you. In the case of compound strings, we could use this method to convert a C string to a compound string. The following code fragment has the same logical effect as the previous example:

```
push_b = XtVaCreateManagedWidget ( "widget_name",
                                   xmPushButtonWidgetClass,
                                   parent,
                                   XtVaTypedArg, XmNlabelString,
                                 XmRString, "Push Me", strlen ("Push Me") + 1,
                                   NULL);
```

XtVaTypedArg takes four additional parameters: the name of the resource, the type of the value specified for the resource, the value itself, and the size of the value. We set the XmNlabelString resource. We want to avoid converting the character string to a compound string, so we specify a char * value and XmRString as its type.*The string "Push Me" is the string value; the length of the string, including the NULL-terminating byte, is 8.

The XtVaTypedArg method for specifying a compound string resource is only a programmatic convenience; it does not save time or improve performance. The three-step process of creating, setting, and freeing the compound string still takes place, but it happens within Motif's compound string resource converter. Using automatic conversion is actually slower than converting a string using XmStringCreateLocalized(). However, unless you are creating hundreds of strings, the difference may well be negligible.†

The reason none of the examples in this book do not make use of the feature is that we are trying to demonstrate good programming techniques tuned to a large-scale, production-size, and quality application. Using the XtVaTypedArg method for compound strings is painfully slow when repeated over hundreds of Labels, PushButtons, Lists, and other widgets. The XtVaTypedArg method is perfectly reasonable for converting other types of resources, however. If you are converting a lot of values from one type to another, it is in your own best interest to evaluate the conversion process yourself by testing the automatic versus the manual conversion methods.

---

* This terminology may be confusing to a new Motif programmer. Xt uses the typedef String for char *. The representation type used by Xt resource converters for this type is XtRString (XmRString in Motif). A compound string, on the other hand, is of type XmString; its representation type is XmRXmString. You just have to read the symbols carefully. Resource converters are described in detail in Volume 4, *X Toolkit Intrinsics Programming Manual*, Motif Edition.

† There is some evidence that the XtVaTypedArg method outlined here for compound strings results in memory leakage, so this should be viewed with some suspicion.

Rendition Tags

Motif provides compound string creation routines that allow you to specify a tag used to associate the compound string with a rendition. This tag is a programmer-specified identifier that enables a Motif widget to pick its rendition from a render table at run-time*.

The XmStringCreate() routine allows you to specify a rendition tag. The routine takes the following form:

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

This routine creates and allocates a new compound string and associates the *tag* parameter with that string. As with any compound string, be sure to free it with XmStringFree() when you are done using it.

XmStringCreate() creates a compound string that has no specified direction. The default direction of a string may be taken from the XmNlayoutDirection resource†. This resource is defined by manager widgets; it specifies the layout and string direction for all the children of the manager. If the default direction is not adequate, XmStringDirectionCreate() can be used to create a compound string with an explicit direction, as we'll discuss shortly.‡

As of Motif 2.0, XmStringGenerate() is now the preferred routine for creating simple compound strings. This routine can convert either single or multi-byte character strings to compound string format. The routine is defined as follows:

```
XmString XmStringGenerate ( XtPointer        text,
                            XmStringTag      tag,
                            XmTextType       type,
                            XmStringTag      rendition)
```

The *text* parameter is the input string to be converted. This can be either single or multi-byte data, the *type* parameter will inform the function of the supplied *text* type. *type* can be either XmCHARSET_TEXT or XmMULTIBYTE_TEXT. The *tag* parameter is simply the name to be associated with the created compound string: if this is NULL, the default value XmFONTLIST_DEFAULT_TAG is used. If the rendition parameter is not NULL, the compound string which is returned is enclosed within matching XmSTRING_RENDITION_BEGIN and XmSTRING_RENDITION_END string components: this is the way that a compound string is created such that it is associated with a particular named rendition from the widgets' render table. The following code fragment is the simple case: convert a string to a compound string without explicitly hard-coding a rendition association:

---

* Prior to Motif 2.0, the tag was associated with a font list rather than a rendition. In Motif 1.1, the font list was referred to as a character set, but strictly speaking, it did not specify a character set.

† The XmNstringDirection resource is deprecated as of Motif 2.0, and is subsumed into the XmNlayoutDirection resource. XmNlayoutDirection is available only from Motif 2.0 onwards.

‡ XmStringCreateLtoR() is deprecated as of Motif 2.0.

```
XmString xms = XmStringGenerate ( (XtPointer) "Hello World",
                                  NULL,
                                  XmCHARSET_TEXT,
                                  NULL);
```

The actual rendition, and hence the font or font set, that is associated with the compound string is dependent on the widget that renders the string. Every Motif widget that displays text has an XmNrenderTable resource. This resource specifies a list of rendition objects for the widget; each entry in the list may have an optional tag associated with it. For example, a resource file might specify a render table as follows:

```
*renderTable: renditionA, renditionB, renditionC
*renditionA.fontName: -*-courier-*-r-*--*-120-*
*renditionA.fontType: FONT_IS_FONT
*renditionB.fontName: -*-courier-*-r-*--*-140-*
*renditionB.fontType: FONT_IS_FONT
*renditionC.fontName: -*-courier-*-r-*--*-180-*
*renditionC.fontType: FONT_IS_FONT
```

At run-time, the compound string is rendered using the rendition in the widget's render table that matches the tag specified in the compound string creation function. In Motif 1.2 onwards, the compound string rendering functions use the X11R5 text output functions, so compound strings are displayed appropriately for the current locale.

In Motif 1.2, if Motif cannot find a match, the compound string is rendered using the first item in the widget's font list, regardless of its tag. In Motif 2.0 and later, if no matching rendition can be found, the XmDisplay object XmNnoRenditionCallback is invoked. This callback, if specified, can be used to supply the missing rendition information. The XmNnoRenditionCallback is discussed in Chapter 24, *Render Tables.* After invocation of this callback, if no matching or supplied rendition can still be found, the first rendition in the render table which has the default tag XmFONTLIST_DEFAULT_TAG is used to render the compound string, provided it has a font. If this default rendition also has no font, the compound string is not drawn, and a warning message is displayed.

This loose binding between the compound string and the rendition (and hence the font or font set) used to render it is useful in a number of ways:

- The same compound string can be rendered using different fonts in different widgets simply by specifying a different render table for each widget. For example:

  ```
  *XmPushButton.renderTable: renditionA
  *XmPushButton*renditionA.fontName: -*-courier-*-r-*--*-120-*
  *XmList.renderTable: renditionA
  *XmList*renditionA.fontName: -*-helvetica-*-r-*--*-120-*
  ```

- Compound strings rendered in different fonts can be concatenated to create a multi-font compound string. The font for each segment is selected from the renditions within the widget's render table by means of a unique tag.

- Compound strings can be language-independent, with the tag used to select between fonts with different character set encodings. This is the least common use for compound strings, and as of X11R5, it is no longer needed to support internationalized text output.

Example 25-1 demonstrates how a compound string can be rendered using different fonts in different PushButton widgets.*

Example 25-1 The string.c program

```
/* string.c -- create a compound string with the "MY_TAG" tag.
** The tag defaults to the "9x15" font. Create three pushbuttons:
** pb1, pb2, and pb3. The user can specify resources so that each of the
** widgets has a different font associated with the "MY_TAG" tag
** specified in the compound string.
*/

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

String fallbacks[] = { "*MY_TAG.fontName:9x15", NULL };

main (int argc, char *argv[])
{
    Widget         toplevel, rowcol, push_b;
    XtAppContext   app;
    XmString       text;
    Display        *dpy;
    Arg            args[2];

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

    rowcol = XmCreateRowColumn (toplevel, "rowcol", NULL, 0);

    text = XmStringGenerate ( (XtPointer) "Testing, testing...",
                              XmFONTLIST_DEFAULT_TAG,
                              XmCHARSET_TEXT,
                              "MY_TAG");

    XtSetArg (args[0], XmNlabelString, text);
    push_b = XmCreatePushButtonGadget (rowcol, "pb1", args, 1);
    XtManageChild (push_b);
    push_b = XmCreatePushButtonGadget (rowcol, "pb2", args, 1);
    XtManageChild (push_b);
    push_b = XmCreatePushButtonGadget (rowcol, "pb3", args, 1);
    XtManageChild (push_b);
    XmStringFree (text);
```

* XtVaAppInitialize() is considered deprecated in X11R6. XtVaOpenApplication() is only available in X11R6. XmStringGenerate() is only available from Motif 2.0 onwards.

```
    XtManageChild (rowcol);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}
```

This simple program creates three PushButton gadgets that all use the same compound string for their labels. The rendition tag MY_TAG is associated with the 9x15 font in the fallback resources. By default, all of the buttons look the same, as shown in Figure 25-1.

Figure 25-1 Output of string program

However, Figure 25-2 shows what happens to the output when the following resources are specified:

```
*.renderTable: MY_TAG
*.fontType: FONT_IS_FONT
*pb1.*MY_TAG.fontName: -*-courier-*-r-*--*-120-*
*pb2.*MY_TAG.fontName: -*-courier-*-r-*--*-140-*
*pb3.*MY_TAG.fontName: -*-courier-*-r-*--*-180-*
```

Figure 25-2 Output of string program with rendition resources set

The rendition associated with MY_TAG for each of the PushButtons is different, so the compound string for each one is rendered in a different font. Since each render table only contains one rendition, Motif has no choice but to attempt to display the compound string using the font associated with the single rendition. The following resource specification creates the output shown in Figure 25-3:

```
*.fontType: FONT_IS_FONT
*.renderTable: MY_TAG

*pb1*.fontName: -*-courier-*-r-*--*-120-*
*pb2*.renderTable: fixed_rendition, courier_rendition
*pb2.fixed_rendition.fontName: fixed
*pb2.courier_rendition.fontName: -*-courier-*-r-*--*-140-*
*pb3*MY_TAG.fontName: fixed,-*-courier-*-r-*--*-180-*
```

Figure 25-3 Output of string program with multiple rendition resources set

In this case, the compound string in the first PushButton uses a 12-point Courier font, since that is the only rendition in the render table, it has the implicit tag _MOTIF_DEFAULT_LOCALE assigned to it, and this is used as the default. The third button uses the 18-point Courier font associated with MY_TAG. The second PushButton does not render the compound string label because there is no rendition with a matching tag, and no default unnamed rendition. Motif prints out a warning message:

```
Warning: No font found.
```

This behavior is different to that of Motif 1.2. In Motif 1.2, the toolkit would use the first font if the compound string contained a tag which did not match any of the fonts in the XmNfontList resource of the widget concerned. In Motif 2.0 and later, the programmer must supply a default font; the XmDisplay object XmNnoFontCallback is called whenever this situation arises, and the programmer can supply the missing font information in the callback. The XmNnoFontCallback procedure is discussed in Chapter 24, *Render Tables*.

The Default Tag

The constant XmFONTLIST_DEFAULT_TAG is used to tag compound strings that are created in the encoding of the current locale. When a compound string is created using XmStringCreateLocalized(), this tag is used. The equivalent compound string can also be created using XmStringCreate() with the tag explicitly set to XmFONTLIST_DEFAULT_TAG. In Motif 2.0 and later, the compound string can also be created using XmStringGenerate(), with the tag either explicitly set to XmFONTLIST_DEFAULT_TAG, or implicitly by setting the tag to NULL. Motif looks for a rendition with a matching tag when it renders the compound string. In Motif 2.0 and later, a rendition which is created with a NULL tag is assigned the tag _MOTIF_DEFAULT_LOCALE, and this will match with a compound string component which has the tag XmFONTLIST_DEFAULT_TAG.

An internationalized application can use XmFONTLIST_DEFAULT_TAG to ensure that compound strings are rendered correctly for the current locale. However, it is possible to use explicit rendition tags for locale-specific text. Explicit tags are necessary when an application wants to display compound strings using different point sizes or type styles. In this case, the compound string and the renditions associated with it need to use the same tag; the tag can be mapped to XmFONTLIST_DEFAULT_TAG using XmRegisterSegmentEncoding().

In Motif 1.1, the first font in widget's font list is the default character set for that widget. If the widget does not have a font list, it uses a default character set referred to by the constant XmSTRING_DEFAULT_CHARSET. If the user has set the LANG environment variable, its value is used for this character set. If this value is invalid or its associated font cannot be used, Motif uses the value of XmFALLBACK_CHARSET, which is vendor-defined but typically set to "ISO8859-1".

For backwards compatibility, Motif 1.2 equated XmFONTLIST_DEFAULT_TAG with XmSTRING_DEFAULT_CHARSET when it could not find an exact match between a compound string and a font list. XmFONTLIST_DEFAULT_TAG in a compound string or font list matched the tag used in creating a compound string or specified in a font list entry with the tag XmSTRING_DEFAULT_CHARSET.

Again for backwards compatibility, in Motif 2.0 and later, renditions with the tag _MOTIF_DEFAULT_LOCALE match against compound string components whose tag is XmFONTLIST_DEFAULT_TAG. A rendition created with a NULL tag is assigned the default tag _MOTIF_DEFAULT_LOCALE.

RenderTable Resources

Some Motif widgets define rendition resources that allow them to provide a consistent appearance for all of their children.

In Motif 1.2, the VendorShell widget defined the XmNbuttonFontList, XmNlabelFontList, and XmNtextFontList resources, while the MenuShell defined XmNbuttonFontList and XmNlabelFontList. These resources referred to XmFontList data, and were applied to all of the buttons, Labels, and Text widgets that are descendents of the widget. In Motif 1.1, the VendorShell and MenuShell only defined the XmNdefaultFontList resource; this resource applied to all of the children of the widget. For backwards compatibility, if one of the more specific font list resources was not set, its value was taken from XmNdefaultFontList.

In Motif 2.0 and later, the resources XmNbuttonFontList, XmNlabelFontList, XmNtextFontList, and XmNdefaultFontList are deprecated, and have been replaced by the resources XmNbuttonRenderTable, XmNlabelRenderTable, and XmNtextRenderTable. There is no putative XmNdefaultRenderTable defined by the MenuShell or VendorShell classes.

The BulletinBoard widget defines the resources XmNbuttonRenderTable, XmNlabelRenderTable, and XmNtextRenderTable primarily for use in dialog boxes. These render tables apply to the buttons, Labels, and Text widgets that descend from a BulletinBoard. For more information on the use of the resources in dialog boxes, see Chapter 5, *Introduction to Dialogs*.

All of these render table resources are designed to help you maintain a consistent interface. However, you can always specify the font for a particular button, Label, or Text widget

using the widget's XmNrenderTable resource, as this resource overrides the more general ones.

Compound String Segments

A compound string is composed of *segments*, where each segment contains a continuous sequence of text with no change in rendition tag or direction. A compound string segment can be terminated by a *separator*, which is the equivalent of a newline in a character string.* Segments can be concatenated with other segments or compound strings to create longer strings; each segment can specify a different tag and direction to make a string that uses multiple fonts and directions.

XmStringComponentCreate() provides complete control over the creation of a compound string, as it allows you to specify individual text, rendition tag, tab, and direction components which can be concatenated together into a whole†. In Motif 2.0 and later, Tab segments can also be added to a compound string; these can be used to create a multi-columnar arrangement, and is typically used by the List widget. The routine takes the following form:

```
XmString XmStringComponentCreate ( XmStringComponentType     type,
                                   unsigned int              length,
                                   XtPointer                 value)
```

String Directions

Compound strings are rendered either from left-to-right or from right-to-left. If you are going to use left-to-right strings uniformly in your applications, you really don't need to read this section.There are several ways to build a compound string that is rendered from right-to-left; the best method is dependent on the nature of your application.

If your application uses right-to-left strings for all of its widgets, you may want to use the XmNlayoutDirection resource‡. This resource specifies the layout of the component in general terms: for Primitive widgets, it also specifies the direction for compound strings, provided that the string direction is not hard-coded in the compound strings themselves. If you use this resource, you can continue to use XmStringCreate() or XmStringCreateLocalized() to create compound strings.

Most right-to-left languages display certain things, like numbers, from left-to-right, so it is not always possible to use the XmNlayoutDirection resource. In this case, you have to create compound string segments that hard-code their directional information.You can create individual string segments with a specific direction by using either

* Separators in compound strings should not be confused with the Separator widget and gadget class.

† In Motif 2.0 and later, XmStringSegmentCreate() is deprecated. XmStringComponentCreate() is only available in Motif2.0 and later.

‡ The resource XmNstringDirection is considered deprecated in Motif 2.0 and later.

XmStringDirectionCreate() or XmStringComponentCreate(). Both of these routines take an argument of type XmStringDirection, which is defined as an unsigned char. You can specify either XmSTRING_DIRECTION_R_TO_L, XmSTRING_DIRECTION_L_TO_R, or XmSTRING_DIRECTION_DEFAULT for values of this type.

When using XmStringComponentCreate(), you specify the string direction using the *value* parameter. For example, we can change the call to XmStringCreate() in Example 25-1 to the following:

```
XmStringDirection direct = XmSTRING_DIRECTION_R_TO_L;
XmString dir = XmStringComponentCreate ( XmSTRING_COMPONENT_DIRECTION,
                                         sizeof (XmStringDirection),
                                        (XtPointer) &direct);
char *cptr = "Testing, testing";
XmString text = XmStringComponentCreate ( XmSTRING_COMPONENT_TEXT,
                                          strlen (cptr),
                                          (XtPointer) cptr);
XmString xms = XmStringConcatAndFree (dir, text);
```

Obviously, you would normally do this only if you were using a font that was meant to be read from right-to-left, such as Hebrew or Arabic. The output that results from this change is shown in Figure 25-4.

Figure 25-4 Output of string program using a right-to-left direction

You can also use the function XmStringDirectionCreate() to create a compound string segment that contains only directional information. This routine takes the following form:

```
XmString XmStringDirectionCreate (XmStringDirection direction)
```

The routine returns a compound string segment that can be concatenated with another compound string to cause a directional change.

String Separators

Separators are used to break compound strings into multiple lines, in much the same way that a newline character does in a character string. To demonstrate separators, we can change the string creation line in Example 25-1 to the following:

```
text = XmStringGenerate ( (XtPointer) "Testing,\nTesting...",
                          XmFONTLIST_DEFAULT_TAG,
                          XmCHARSET_TEXT,
                          "MY_TAG");
```

We can use XmStringGenerate() because this function interprets embedded newline characters (\n) as separators. The effect of this change is shown in Figure 25-5, where the PushButtons display multiple lines of text.

Figure 25-5 Output of string program with multiple lines

XmStringCreate() does not interpret newline characters as separators; it creates a single compound string segment in which the '\n' is treated just like any other character value in the associated font or font set, as shown in Figure 25-6. XmStringComponentCreate(), however, can be told to create a separator which can be embedded into a larger compound string.

Figure 25-6 Output of string program with \n not interpreted as a separator

Most applications need newline characters to be interpreted as separators. For example, if you are using fgets() or read() to read the contents of a file, and newlines are read into the buffer, you should use XmStringGenerate() to convert the buffer into a compound string that contains separators. Example 25-2 shows a function that reads the contents of a file into a buffer and then converts that buffer into a compound string.*

Example 25-2 The ConvertFileToXmString() routine.

```
XmString ConvertFileToXmString (char *filename, int *lines)
{
    struct stat    statb;
    int            fd, len;
    char           *text;
    XmString       str;

    *lines = 0;

    if (!(fd = open (filename, O_RDONLY))) {
        XtWarning ("Internal error -- can't open file");
```

* XmStringCreateLtoR() is deprecated in Motif 2.0 and later. XmStringGenerate() is only available from Motif 2.0.

```
            return NULL;
        }

        if (fstat (fd, &statb) == -1 ||
                        !(text = XtMalloc ((len = statb.st_size) + 1)))
        {
            XtWarning ("Internal error -- can't show text");
            close (fd);
            return NULL;
        }

        (void) read (fd, text, len);
        text[len] = 0;

        str = XmStringGenerate ((XtPointer) text,
                                XmFONTLIST_DEFAULT_TAG,
                                XmCHARSET_TEXT,
                                NULL);
        XtFree (text);
        (void) close (fd);

        *lines = XmStringLineCount (str);
        return str;
    }
```

Since separators are considered to be line breaks, we can count the number of lines in the compound string using the function `XmStringLineCount()`. However, this does not imply that separators terminate compound strings or cause font changes. As we have shown, a separator can be inserted into the middle of a compound string without terminating it. The fact that separate segments are created has little significance unless you need to convert compound strings back into C strings, which we discuss in the *Compound String Conversion* Section later in this chapter.

Multiple-font Strings

Once multiple renditions with distinctive tags are specified in a render table, you can use the list to display more than one font or font set in a single compound string. You can create a multi-font string in one of two ways: create the compound text in segments or create separate compound strings. Either way, once the segments or strings have been created, they must be concatenated together to form a new compound string that has font-change information embedded in it. Example 25-3 demonstrates the creation of a compound string that uses three fonts.*

Example 25-3 The multi_font.c program

```
/* multi_font.c -- create three compound strings using 12, 14 and 18
** point fonts. The user can specify resources so that each of the strings
```

* `XtVaAppInitialize`() is considered deprecated in X11R6. `XmStringGenerate`() and `XmStringConcatAndFree`() are only available in Motif 2.0 and later.

```
** use different fonts by setting resources similar to that shown
** by the fallback resources.
*/

#include <Xm/Label.h>

String fallbacks[] = {
    "*.renderTable: TAG1, TAG2, TAG3",
    "*.fontType: FONT_IS_FONT",
    "*TAG1.fontName: -*-courier-*-r-*--*-120-*",
    "*TAG2.fontName: -*-courier-bold-o-*--*-140-*",
    "*TAG3.fontName: -*-courier-medium-r-*--*-180-*",
    NULL
};

main (int argc, char *argv[])
{
    Widget         toplevel, label;
    XtAppContext   app;
    XmString       s1, s2, s3, text, tmp;
    Arg            args[2];
    String         string1 = "This is a string ",
                   string2 = "that contains three ",
                   string3 = "separate fonts.";

    XtSetLanguageProc (NULL, NULL, NULL);
    toplevel = XtVaOpenApplication (&app, "String", NULL, 0, &argc, argv,
                                    fallbacks, sessionShellWidgetClass, NULL);
    s1 = XmStringGenerate ((XtPointer) string1, NULL,
                                            XmCHARSET_TEXT, "TAG1");
    s2 = XmStringGenerate ((XtPointer) string2, NULL,
                                            XmCHARSET_TEXT, "TAG2");
    s3 = XmStringGenerate ((XtPointer) string3, NULL,
                                            XmCHARSET_TEXT, "TAG3");

    /* concatenate the 3 strings on top of each other, but we can only
    ** do two at a time. So do s1 and s2 onto tmp and then do s3.
    */
    tmp = XmStringConcatAndFree (s1, s2);
    text = XmStringConcatAndFree (tmp, s3);
    XtSetArg (args[0], XmNlabelString, text);
    label = XmCreateLabel (toplevel, "widget_name", args, 1);
    XtManageChild (label);

    XmStringFree (text);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}
```

The output of this program is shown in Figure 25-7.

Figure 25-7 Output of multi_font program

The XmNrenderTable resource is specified using three renditions, each with a distinct tag. We create each string using XmStringGenerate() with the appropriate text and tag. Then we concatenate the strings using XmStringConcatAndFree(), two at a time until we have a single compound string that contains all the text. XmStringConcatAndFree() does not work like strcat() in C. The Motif function creates a new compound string that is composed of the two existing strings, rather than appending one string to the other string.

It is possible to specify compound string resource values, such as the XmNlabelString resource of the Label widget, in a resource file as normal C strings. Motif provides a resource converter that converts the character string into a compound string. However, this resource converter does not allow you to specify rendition tags in the character string. If you need font changes within a compound string, you need to create the compound strings explicitly in your application as we have done in Example 25-3.

Manipulating Compound Strings

Most C programmers are used to dealing with functions such as strcpy(), strcmp(), and strcat() to copy, compare, and modify strings. However, these functions do not work with compound strings, as they are not based on a byte-per-character format, and they may have NULL characters as well as other types of information embedded in them. In order to perform these common tasks, you can either convert the compound string into a character string, or you can use the compound string manipulation functions provided by Motif. The method you choose depends largely on the complexity of the compound strings you have and/or the complexity of the manipulation you need to do.

Compound String Functions

Motif provides a number of functions that allow you to treat compound strings in much the same way that you treat C-style character arrays. The toolkit provides the following routines:

```
Boolean XmStringByteCompare (XmString string1, XmString string2)
Boolean XmStringCompare (XmString string1, XmString string2)
XmString XmStringConcat (XmString string1, XmString string2)
XmString XmStringConcatAndFree (XmString string1, XmString string2)
XmString XmStringCopy (XmString string)
Boolean XmStringEmpty (XmString string)
Boolean XmStringHasSubstring (XmString string, XmString substring)
Boolean XmStringIsVoid (XmString string)
int XmStringLength (XmString string)
```

Both XmStringCompare() and XmStringByteCompare() compare two compound strings, *string1*, and *string2*. XmStringCompare() simply checks if the strings have the same text components, directions, and separators; it returns True if they do. This routine is simpler and more frequently used than XmStringByteCompare(), which performs a byte-by-byte comparison of the two compound strings. If each string uses the same rendition tags, has the same direction, and contains the same embedded char string internally, the function returns True. The mapping between rendition tags and fonts does not happen until a compound string is rendered by a widget, so whether or not the same rendition tag actually maps to two different fonts does not affect the results of this function.

XmStringConcat() and XmStringConcatAndFree() can be used to concatenate compound strings. Both of these routines create a new compound string and copy the concatenation of *string1* and *string2* into the newly allocated string. With XmStringConcat(), the original strings are preserved, but with XmStringConcatAndFree() the original parameter strings are internally freed. In both cases, you are responsible for freeing the compound string returned by the routines using XmStringFree()*.

You can copy a compound string using XmStringCopy(), which copies the parameter *string* into a newly-allocated compound string.†

XmStringHasSubstring() determines whether or not a compound *string* contains a particular *substring*. For this function, *substring* must be a single-segment compound string. If its text is completely contained within any single segment of *string*, the function returns True. The two strings must use the same rendition tags for the routine to return True.

To get the length of a compound string, use XmStringLength(). This function returns the number of bytes in the compound *string* including all tags, direction indicators, and separators. If the structure of *string* is invalid, the routine returns zero. This function cannot be used to get the length of the text represented by the compound string; it is not the same as strlen()).

You can determine whether or not a compound string contains any segments using XmStringEmpty(). This function returns True if there are no segments in the specified *string* and False otherwise. If the routine is passed NULL, it returns True. The function XmStringIsVoid() is similar, except that it checks only for text, separator, and tab components‡. Therefore testing a compound string which contains only direction indicators using XmStringEmpty() will return False, whereas XmStringIsVoid() will return True.

---

* XmStringNConcat() is deprecated from Motif 2.0.

† XmStringNCopy() is deprecated from Motif 2.0.

‡ XmStringIsVoid() is only available in Motif 2.0 and later.

Compound String Retrieval

You can retrieve a compound string from a Motif widget using XtVaGetValues(). However, the way XtVaGetValues() is used for compound string resources is different than how it is used for most other resources. The value returned by XtVaGetValues() for a compound string resource is a copy of the internal data, so the compound string must be freed by the application, as shown in the following code fragment:

```
XmString         str;
extern Widget    pushbutton;
char             *text;

XtVaGetValues (pushbutton, XmNlabelString, &str, NULL);
...
/* do whatever you want with the compound string */
...
XmStringFree (str); /* must free compound strings from GetValues */
```

To avoid memory leaks in your application, you must remember to free any compound strings that you retrieve from a widget using XtVaGetValues(). You free a compound string using the routine XmStringFree().

Compound String Conversion

If the Motif routines described in the previous section are inadequate for your needs, you can convert compound strings back into C strings and manipulate them using the conventional C functions. This process can be simple or complicated depending on the complexity of the compound string to be converted.

In Motif 1.2, there were two basic methods of conversion. The simplest method was to use the routine XmStringGetLtoR(), which is sufficient if the compound string has a single tag associated with it, and has a left-to-right orientation. The more complex method is to walk along the compound string one segment at a time. This technique is rather low level, and requires the creation of a type, XmStringContext, that is used to identify and maintain the position within the compound string being scanned. This is described below.

In Motif 2.0 and later, XmStringGetLtoR() is deprecated. In its place is the notion of a Parse Table, whereby compound strings can be converted using a table-driven description of the requisite conversion process. This is covered in the *Parse Tables* Section later in this chapter.

The XmStringContext

To cycle through a compound string, you need to use the following sequence of operations:

1. Initialize a string context for the compound string using XmStringInitContext().

2. Iterate through the string by calling XmStringGetNextTriple() to get the type, length, and value associated with each segment*.
3. Free the string context using XmStringFreeContext().

XmStringInitContext() initializes a string context that allows an application to read the contents of a compound string segment by segment. This routine takes the following form:

```
Boolean XmStringInitContext (XmStringContext *context, XmString string)
```

The function allocates a new XmStringContext type and sets the pointer that is passed by the calling function in the *context* parameter to this data. If the allocation is successful and the compound string is valid, the function returns True.

Once the string context has been initialized, the contents of the string can be scanned using XmStringGetNextTriple():

```
XmStringComponentType XmStringGetNextTriple ( XmStringContext    context,
                                              unsigned int       *length,
                                              XtPointer          *value)
```

The routine does not take an XmString parameter because the *context* parameter is used to keep track of the compound string. The function reads the next segment; the values for *length*, *value* are filled in, and the function returns the type of the new segment. The XmStringComponentType parameter is an enumerated type, and has the following possible values:†

```
XmSTRING_COMPONENT_CHARSET
XmSTRING_COMPONENT_TEXT
XmSTRING_COMPONENT_LOCALE_TEXT
XmSTRING_COMPONENT_WIDECHAR_TEXT
XmSTRING_COMPONENT_DIRECTION
XmSTRING_COMPONENT_SEPARATOR
XmSTRING_COMPONENT_TAB
XmSTRING_COMPONENT_LAYOUT_PUSH
XmSTRING_COMPONENT_LAYOUT_POP
XmSTRING_COMPONENT_RENDITION_BEGIN
XmSTRING_COMPONENT_RENDITION_END
XmSTRING_COMPONENT_UNKNOWN
XmSTRING_COMPONENT_END
```

When you are through scanning the compound string, you need to free the string context using XmStringFreeContext(), which takes the following form:

```
void XmStringFreeContext (XmStringContext context)
```

* XmStringGetNextComponent() is deprecated as of Motif 2.0. XmStringGetNextTriple() is only available in Motif 2.0 or later.

† For backwards compatibility, the values XmSTRING_COMPONENT_TAG and XmSTRING_COMPONENT_FONTLIST_ELEMENT_TAG are mapped onto XmSTRING_COMPONENT_CHARSET. The Layout push/pop, Rendition push/pop, and Tab component values are only available from Motif 2.0 onwards.

Example 25-4 shows a routine that uses these functions to print a compound string used as the label for a widget.

Example 25-4 The PrintLabel() routine

```
void PrintLabel (Widget widget)
{
    XmString                str;
    XmStringContext         context;
    char                    *text, *p, buf[256];
    XtPointer               data;
    unsigned int            length;
    XmStringComponentType   type;

    XtVaGetValues (widget, XmNlabelString, &str, NULL);

    if (!XmStringInitContext (&context, str)) {
        /* compound strings from GetValues still need to be freed! */
        XmStringFree (str);
        XtWarning ("Can't convert compound string.");
        return;
    }

    /* p keeps a running pointer through buf as text is read */
    p = buf;
    while ((type = XmStringGetNextTriple (context, &length, &data))
                                            != XmSTRING_COMPONENT_END) {
        switch (type) {
            case XmSTRING_COMPONENT_TEXT :
                text = (char *) data;
                (void) strcpy (p, text);
                p += length;
                XtFree ((char *) data);
                break;
            case XmSTRING_COMPONENT_TAB :
                *p++ = '\t';
                *p = 0;
                break;
            case XmSTRING_COMPONENT_SEPARATOR :
                *p++ = '\n';
                *p = 0;
                break;
        }
    }

    XmStringFreeContext (context);
    XmStringFree (str);

    printf ("Compound string:\n%s\n", buf);
}
```

Parse Tables

From Motif 2.0, strings can be converted to and from compound strings by means of table-driven mappings. A Parse Table consists of Parse Mapping objects. What a Parse Mapping defines is a small piece of the conversion process in the form of a match pattern, and a substitution method. Whenever a sequence of input corresponds to the match pattern it is replaced using the substitution specified in the mapping object.

To be more concrete, a string containing newline characters can be mapped to a compound string by supplying a parse mapping which has a match pattern of the newline character, and a substitution pattern that is a compound string consisting of a segment of type `XmSTRING_COMPONENT_SEPARATOR`. The internal implementation of the toolkit convenience function `XmStringGenerate`() does precisely this: it uses the lower level parse mapping routines, supplying a default parse table that has mappings for newlines and tabs.

You dont need to specify a large Parse Table for most operations. The default mechanisms convert text directly into textual compound string components without the need for the programmer to specify Parse Mappings on a character-by-character basis for all of the input. Typically, you only need to create a Parse Mapping when you wish to handle a particular input character specially.

For many conversions, the default internal mappings are entirely sufficient, and so `XmStringGenerate`() is more than adequate for most tasks: it hides the lower level details of the table-driven parsing process from the programmer. There are however cases where a bespoke converter makes sense. Consider reading a file where the separation between fields is not by tab characters. A typical example of this is the UNIX `/etc/passwd` file, where fields are separated by colons. If we wanted to read lines of this file into a List widget, where each field is in a distinct column, we have to define a parse mapping which converts colons into compound string tab components because the default internal parse tables do not handle this specific conversion: they simply treat the colon like any other character, so if you use the default mechanisms the colon will appear untranslated in the converted compound string.

Thee two most important procedures for performing table-driven string to compound string conversion (and vice-versa) are `XmStringParseText`(), and `XmStringUnparse`(). `XmStringParseText`() converts a single or multi-byte character stream into a compound string; `XmStringUnparse`() converts a compound string back into the single or multi-byte character representation.

Where the conversion involves arrays of strings or compound strings, the routines `XmStringTableParseStringArray`() and `XmStringTableUnparse`() are provided. These are in fact simple convenience routines which do little more than call

XmStringParseText() and XmStringUnparse() iteratively across the array of strings or compound strings.

Examples of all of the above routines will be given in due course. But first, we must define exactly what Parse Mappings and Parse Tables are, and how to create them.

The XmParseMapping Object

A Parse Mapping object is a pseudo-widget; although not a true widget, it is an object which has resources and a resource-style interface. A mapping is created through the routine XmParseMappingCreate(), which takes the following form:

```
XmParseMapping XmParseMappingCreate (Arg *argList, Cardinal argCount)
```

The *argList* parameter is an array of parse mapping resources, *argCount* being the number of resources in the array. This is directly analogous to the Arg array passed as a parameter to standard widget creation routines. The function returns an opaque handle, an XmParseMapping, used to designate the object created. To create a Parse Table, we simply create an array of these, one for each particular piece of bespoke conversion, and pass the array to the XmStringParseText() or XmStringUnparse() routines as required. An XmParseTable is nothing more than an array of XmParseMapping objects.

Once created, the resources associated with an XmParseMapping object can be modified using the XmParseMappingSetValues() routine. This routine is defined as follows:

```
void XmParseMappingSetValues ( XmParseMapping     map,
                               Arg                *argList,
                               Cardinal           argCount)
```

Conversely, we can retrieve the values associated with an XmParseMapping object using the XmParseMappingGetValues() routine:

```
void XmParseMappingGetValues ( XmParseMapping     map,
                               Arg                *argList,
                               Cardinal           argCount)
```

When we have finished using the XmParseMapping object, we free it using XmParseMappingFree():

```
void XmParseMappingFree (XmParseMapping map)
```

XmParseMapping Resources

The most important attributes of an XmParseMapping object are the XmNpattern, XmNpatternType, XmNsubstitute, and the XmNinvokeParseProc resources. XmNpattern represents the input associated with this XmParseMapping object: it is the data against which the input stream is matched. You specify this to the parse mapping object in the form of a pointer to a character (multi-byte character or wide character, depending on what it is you are trying to convert to a compound string), even though any

given parse mapping object concerns itself with converting a single piece of input and not a sequence. If the current input matches the XmNpattern resource, it is replaced in the output stream by either the value of the XmNsubstitute resource, or dynamically constructed by the procedure specified by the XmNinvokeParseProc resource. Which of the two methods is adopted depends upon the value of the XmNincludeStatus resource. If XmNincludeStatus is XmINSERT, the XmNsubstitute resource is used for the transformation; if XmNincludeStatus is XmINVOKE, the XmNinvokeParseProc procedure is used; lastly, if XmNincludeStatus is XmTERMINATE, the conversion process is terminated at that point in the input stream. The XmNinvokeParseProc resource is considered later in this chapter. Remember that the XmNpattern resource requires a pointer to an array as its value, although only the first item in the array is used when performing matching. XmNpattern can specify more than just character arrays: we can parse multi-byte or wide-character input as well. The type of input is specified using the XmNpatternType resource, and it takes the following values:

```
XmCHARSET_TEXT          XmWIDECHAR_TEXT          XmMULTIBYTE_TEXT
```

To make this clear, the following code fragment creates a parse mapping for converting colon characters in a simple char * input stream into compound string tab components. Since it is only the colon that we are treating specially, we only need one parse mapping object to convert the entire input to a compound string: everything except the colon is converted using default internal algorithms:

```
XmParseMapping     map;
XmString           tab;
Arg                args[4];
Cardinal           n = 0;

/* Match against the colon character */
/* Note we specify the string ":" not the character ':' */
XtSetArg (args[n], XmNpattern, ":"); n++;

/* XmNpattern contains a simple character array */
/* so we specify the pattern type as XmCHARSET_TEXT */
XtSetArg (args[n], XmNpatternType, XmCHARSET_TEXT); n++;

/* Convert to a tab compound string component */
tab = XmStringComponentCreate (XmSTRING_COMPONENT_TAB, 0, NULL);
XtSetArg (args[n], XmNsubstitute, tab); n++;

/* Use the XmNsubstitute resource for conversion */
/* Not the XmNinvokeParseProc resource */
XtSetArg (args[n], XmNincludeStatus, XmINSERT); n++;

/* Create the Parse Mapping object */
map = XmParseMappingCreate (args, n);
```

The way the parse system works is as follows: each character in the input stream is matched in turn against the XmNpattern resources for each of the mappings in a given

XmParseTable. Where there is a match, that mapping is used to perform the translation, either by directly replacing the input with the XmNsubstitute value, or by calling the XmNinvokeParseProc routine to provide the output value. The input stream is advanced by one character (wide-character or multi-byte character, depending on the input stream type), and the process reiterates until either the end of the input is reached, or some parse mapping object specifies XmTERMINATE as the required match action. The concatenation of each chunk of output is performed automatically by the parse process: the parse mapping objects only concern themselves with supplying the individual chunks for a given piece of input.

It is important to stress that we only need to provide mapping objects for special processing, because the default action in the absence of a supplied mapping for some input is a direct conversion to the output. For example, if we wanted to convert lines of /etc/passwd into a compound string, we only need to supply a mapping that specifies the colon character is a logical tab separator. Everything else gets converted implicitly on a simple character-by-character basis.

Converting to Compound Strings

Example 25-5 illustrates a simple string to compound string conversion. It loads lines read from the /etc/passwd file into a multi-column List widget. For details of TabLists used to provide the multi-column layout, you are referred to Chapter 24, *Render Tables*.

Example 25-5 The parse_file.c program

```
/* parse_file.c: converts a file into
** a multi-column list format. Assumes that
** the file is colon-separated fields.
*/

#include <stdio.h>
#include <Xm/Xm.h>
#include <Xm/List.h>

/* construct an array of compound strings
** from loading a file using colon as the field separator.
**
** A more generic routine would pass the field separator in.
*/
XmString *load_file (Widget list, char *file, int *count)
{
    XmParseMapping     map[2];
    FILE               *fptr;
    char               buffer[256];
    Arg                args[8];
    char               *cptr;
    XmString           tab;
    XmString           *xms_array = (XmString *) 0;
    int                xms_count = 0;
    int                n;
```

```
    *count = 0;

    if ((fptr = fopen (file, "r")) == (FILE *) 0) {
        return NULL;
    }

    /* Map colons to tabs */
    n = 0;
    tab = XmStringComponentCreate (XmSTRING_COMPONENT_TAB, 0, NULL);
    XtSetArg (args[n], XmNpattern, ":");                n++;
    XtSetArg (args[n], XmNpatternType, XmCHARSET_TEXT); n++;
    XtSetArg (args[n], XmNsubstitute, tab);             n++;
    XtSetArg (args[n], XmNincludeStatus, XmINSERT);     n++;
    map[0] = XmParseMappingCreate (args, n);

    /* Throw away newlines by terminating the parse for this line */
    n = 0;
    XtSetArg (args[n], XmNpattern, "\n");               n++;
    XtSetArg (args[n], XmNpatternType, XmCHARSET_TEXT); n++;
    XtSetArg (args[n], XmNincludeStatus, XmTERMINATE);  n++;
    map[1] = XmParseMappingCreate (args, n);

    while ((cptr = fgets (buffer, 255, fptr)) != (char *) 0) {
        xms_array = (XmString *) XtRealloc ((char *) xms_array,
                                (xms_count + 1) * sizeof (XmString));

        xms_array[xms_count] = XmStringParseText (cptr, NULL, NULL,
                                        XmCHARSET_TEXT, map, 2, NULL);

        xms_count++;
    }

    (void) fclose (fptr);

    XmParseMappingFree (map[0]);
    XmParseMappingFree (map[1]);

    *count = xms_count;

    return xms_array;
}

main (int argc, char *argv[])
{
    Widget          toplevel, list;
    XtAppContext    app;
    Arg             args[8];
    XmTabList       tlist = NULL;
    XmRenderTable   rtable;
    XmRendition     rendition;
    XmString        *data = (XmString *) 0;
    int             lines = 0;
    int             n, i;
```

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

    n = 0;
    XtSetArg (args[n], XmNvisibleItemCount, 10); n++;
    /* For TabList separation calculations */
    XtSetArg (args[n], XmNunitType, XmINCHES);   n++;
    list = XmCreateScrolledList (toplevel, "list", args, n);
    XtManageChild (list);

    /* Load the data from file as an array of compound strings */
    data = load_file (list, ((argc > 1) ? argv[1] : "/etc/passwd"), &lines);

    XtVaSetValues (list, XmNitems, data, XmNitemCount, lines, NULL);

    /* Use the toolkit to propose a tab list for the items */
    /* This isn't ideal but will do for this example */
    tlist = XmStringTableProposeTablist (data, lines, list,
                                         (float) 0.1, XmRELATIVE);

    /* Create a render table for the List using the tab list */
    n = 0;
    XtSetArg (args[n], XmNtabList, tlist);                          n++;
    XtSetArg (args[n], XmNfontName, "-*-courier-*-r-*--*-120-*"); n++;
    XtSetArg (args[n], XmNfontType, XmFONT_IS_FONT);                n++;
    rendition = XmRenditionCreate (list, XmFONTLIST_DEFAULT_TAG, args, n);
    rtable = XmRenderTableAddRenditions (NULL, &rendition, 1,
                                         XmMERGE_REPLACE);
    XtVaSetValues (list, XmNrenderTable, rtable, NULL);

    /* Free up the temporarily allocated memory */
    XmRenditionFree (rendition);
    XmRenderTableFree (rtable);
    XmTabListFree (tlist);

    for (i = 0; i < lines; i++)
        XmStringFree (data[i]);

    XtFree ((char *) data);

    /* Display the interface */
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}
```

Sample output from the program is given in Figure 25-8.

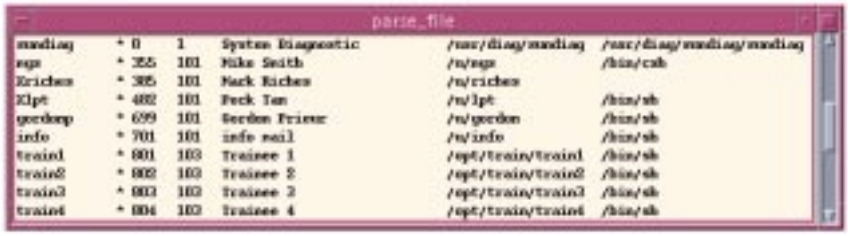


Figure 25-8 Output of the parse_file program

The interesting part of the program is the routine `load_file`(). This parses a file into an array of compound strings by constructing two `XmParseMapping` objects. The first object converts colons into tab components:

```
/* Map colons to tabs */
n = 0;
tab = XmStringComponentCreate (XmSTRING_COMPONENT_TAB, 0, NULL);
XtSetArg (args[n], XmNpattern, ":");                n++;
XtSetArg (args[n], XmNpatternType, XmCHARSET_TEXT); n++;
XtSetArg (args[n], XmNsubstitute, tab);             n++;
XtSetArg (args[n], XmNincludeStatus, XmINSERT);     n++;
map[0] = XmParseMappingCreate (args, n);
```

The second `XmParseMapping` object is used to throw away the trailing newline character returned by the `fgets`() system call. We could just as easily have truncated the returned string from `fgets`() to remove the newline, but it was more interesting to demonstrate how to terminate parsing of an input sequence using an `XmParseMapping` object. Here we simply specify the `XmNincludeStatus` as `XmTERMINATE` to throw the newline away:

```
/* Throw away newlines by terminating the parse for this line */
n = 0;
XtSetArg (args[n], XmNpattern, "\n");               n++;
XtSetArg (args[n], XmNpatternType, XmCHARSET_TEXT); n++;
XtSetArg (args[n], XmNincludeStatus, XmTERMINATE);  n++;
map[1] = XmParseMappingCreate (args, n);
```

Once the parse table is assembled, we only need to call the right routine to convert the input. This is the function `XmStringParseText`(), which takes the following form:

```
XmString XmStringParseText ( XtPointer        input,
                             XtPointer        *input_end,
                             XmStringTag      tag,
                             XmTextType       input_type,
                             XmParseTable     parse_table,
                             Cardinal         parse_table_size,
                             XtPointer        client_data)
```

The data to be converted to a compound string is given by *`input`*: this can be an ordinary C string as in Example 25-5, or a wide-character or multi-byte array. Which of these various input types is the case must be specified through the *`input_type`* parameter,

which can be one of XmCHARSET_TEXT, XmWIDECHAR_TEXT, or XmMULTIBYTE_TEXT. The *input_end* parameter specifies a point within the input where parsing is to terminate. If NULL, parsing continues to the end of the input unless terminated by an XmParseMapping object. If parsing does terminate before the end of the *input*, and *input_end* is not NULL, the routine resets the *input_end* pointer to where parsing actually finished. The *tag* parameter is used to name the created compound string: if the value is NULL, the compound string is created with the default tag _MOTIF_DEFAULT_LOCALE. The *parse_table* and *parse_table_count* parameters identify the array of XmParseMapping objects used to control the conversion process. If *parse_table* is NULL, a default internal parse table is used which simply converts newlines to compound string separator components, and tabs to tab components (and thus is equivalent to XmStringGenerate()). The *client_data* parameter is used to pass application-specific data to any parse procedures within the *parse_table*. In particular, the client_data is passed to any XmNinvokeParseProc routines specified for the XmParseMapping objects. The XmNinvokeParseProc routines are discussed in the *Parse Procedures* Section later in this chapter.

Another routine for converting strings to compound strings is XmStringTableParseStringArray(). This works using an array of input strings rather than a single input. We could have used this routine in Example 25-5 if we loaded the contents of the file into an array of C strings before performing the conversion process, instead of converting each line as it is encountered. The convenience routine XmStringTableParseStringArray() is defined as follows:

```
XmStringTable
XmStringTableParseStringArray ( XtPointer        *input_array,
                                Cardinal         input_array_count,
                                XmStringTag      tag,
                                XmTextType       input_type,
                                XmParseTable     parse_table,
                                Cardinal         parse_table_size,
                                XtPointer        client_data)
```

The parameters to XmStringTableParseStringArray() are similar to those of XmStringParseText(), except that the data to be converted is specified using the *input_array* and *input_array_count* parameters. The main difference between the two routines is that XmStringTableParseStringArray() does not have an *input_end* parameter. Example 25-6 is a reworking of the load_file() routine to use XmStringTableParseStringArray().

Example 25-6 The load_file_array() routine.

```
/* construct an array of compound strings
** from loading a file using colon as the field separator.
**
** A more generic routine would pass the field separator in.
*/
```

```
XmString *load_file_array (Widget list, char *file, int *count)
{
    XmParseMapping     map[2];
    FILE               *fptr;
    char               buffer[256];
    Arg                args[8];
    char               *cptr;
    XmString           tab;
    XmString           *xms_array = (XmString *) 0;
    int                lines = 0;
    char               **data = (char **) 0;
    int                n;

    *count = 0;

    if ((fptr = fopen (file, "r")) == (FILE *) 0) {
        return NULL;
    }

    /* Map colons to tabs */
    n = 0;
    tab = XmStringComponentCreate (XmSTRING_COMPONENT_TAB, 0, NULL);
    XtSetArg (args[n], XmNpattern, ":"); n++;
    XtSetArg (args[n], XmNpatternType, XmCHARSET_TEXT); n++;
    XtSetArg (args[n], XmNsubstitute, tab); n++;
    XtSetArg (args[n], XmNincludeStatus, XmINSERT); n++;
    map[0] = XmParseMappingCreate (args, n);

    /* Throw away newlines by terminating the parse for this line */
    n = 0;
    XtSetArg (args[n], XmNpattern, "\n"); n++;
    XtSetArg (args[n], XmNpatternType, XmCHARSET_TEXT); n++;
    XtSetArg (args[n], XmNincludeStatus, XmTERMINATE); n++;
    map[1] = XmParseMappingCreate (args, n);

    while ((cptr = fgets (buffer, 255, fptr)) != (char *) 0) {
        data = (char **) XtRealloc ((char *) data,
                                    (lines + 1) * sizeof (char *));
        data[lines] = XtMalloc ((unsigned) strlen (cptr) + 1);
        (void) strcpy (data[lines], cptr);

        lines++;
    }

    (void) fclose (fptr);

    xms_array = XmStringTableParseStringArray ((XtPointer *) data, lines, NULL,
                                    XmCHARSET_TEXT, map, 2, NULL);

    for (n = 0; n < lines; n++) {
        XtFree (data[n]);
    }

    XtFree ((char *) data);
```

```
    XmParseMappingFree (map[0]);
    XmParseMappingFree (map[1]);

    *count = lines;

    return xms_array;
}
```

Converting From Compound Strings

Converting compound strings back into a character array, whether to ordinary C or wide/multi-byte strings, is very similar to the process for converting strings to compound strings. We construct an XmParseTable with appropriate bespoke mappings if the toolkit default maps are insufficient, then call the relevant parsing procedure. In this case, we call either XmStringUnparse() or XmStringTableUnparse(), depending on whether it is a simple compound string or an array we need to convert. XmStringUnparse() is defined as follows:

```
XtPointer XmStringUnparse ( XmString         input,
                            XmStringTag      tag,
                            XmTextType       tag_type,
                            XmTextType       output_type,
                            XmParseTable     parse_table,
                            Cardinal         parse_table_count,
                            XmParseModel     parse_model)
```

The compound string to be converted is specified by *input*. How we want the compound string to be converted depends upon the *output_type* parameter. If we want an ordinary C string, we specify *output_type* as XmCHARSET_TEXT. For wide character text or multi-byte array, specify XmWIDECHAR_TEXT or XmMULTIBYTE_TEXT respectively. The routine returns a generic pointer: cast the result to the appropriate data type depending on the value of *output_type*. For a normal C style string, the return value of XmStringUnparse() should be simply cast to char *. XmStringUnparse() returns dynamically allocated memory, and thus the return value should be freed when no longer in use. Only those compound string components which match the designated *tag* are converted: if *tag* is NULL, all components of *input* are converted. *tag_type* specifies the representation of *tag*, which could be in ordinary, wide character, or multi-byte format. *tag_type* takes the same range of values as *output_type* described above. The *parse_table* and *parse_table_count* parameters specify the parse mappings for controlling the conversion process. If *parse_table* is NULL, the default internal parse table is used, which converts compound string separators to newlines and tab components to the tab character. The *parse_model* parameter specifies how non-textual compound string components are to be converted. The possible values are:

```
XmOUTPUT_ALL                XmOUTPUT_BEGINNING
XmOUTPUT_END                XmOUTPUT_BOTH
XmOUTPUT_BETWEEN
```

The value XmOUTPUT_ALL attempts to convert everything. The value XmOUTPUT_BETWEEN will convert a non-text component if and only if it appears between two text components. This can be used as a filter to map multiple adjoining tab components to a single tab character, for instance. XmOUTPUT_END only converts non-text components if they follow a textual one, so this could be used to strip tab components off the front. XmOUTPUT_BEGINNING converts non-text if it precedes text. XmOUTPUT_BOTH is a combination of XmOUTPUT_BEGINNING and XmOUTPUT_END.

As an example, consider the reverse of Example 25-5, where the contents of a multi-column List is output to file, using the colon character as a field separator. The routine to perform this task is given in Example 25-7:

Example 25-7 The dump_file() routine.

```
/* Dump the contents of a List to file,
** using colon as the field separator.
**
** A more generic routine would pass the field separator in.
*/
void dump_file (Widget list, char *file)
{
    XmParseMapping     map;
    FILE               *fptr;
    Arg                args[8];
    XmString           tab;
    XmString           *xms_array = (XmString *) 0;
    int                xms_count = 0;
    char               *output;
    int                n, i;

    if ((fptr = fopen (file, "w")) == (FILE *) 0) {
        return;
    }

    XtVaGetValues (list, XmNitems, &xms_array,
                                    XmNitemCount, &xms_count, NULL);

    /* Map tabs to colons */
    n = 0;
    tab = XmStringComponentCreate (XmSTRING_COMPONENT_TAB, 0, NULL);
    XtSetArg (args[n], XmNpattern, ":"); n++;
    XtSetArg (args[n], XmNpatternType, XmCHARSET_TEXT); n++;
    XtSetArg (args[n], XmNsubstitute, tab); n++;
    XtSetArg (args[n], XmNincludeStatus, XmINSERT); n++;
    map = XmParseMappingCreate (args, n);

    for (i = 0; i < xms_count; i++) {
        output = (char *) XmStringUnparse (xms_array[i], NULL,
                                        XmCHARSET_TEXT, XmCHARSET_TEXT,
                                        &map, 1, XmOUTPUT_ALL);
        (void) fprintf (fptr, "%s\n", output);
    }
```

```
    (void) fclose (fptr);

    XmParseMappingFree (map);
}
```

The important thing to note is that for converting compound strings to strings, an XmParseMapping is set up exactly as though we are converting to a compound string. That is, XmNpattern is the C string we get from the conversion, XmNsubstitute is the compound string segment to match against. Since we are substituting a compound string for a C string, we might reasonably expect XmNsubstitute to contain the required C string value, but this is not the case. XmNsubstitute always contains a compound string, XmNpattern contains the equivalent character, wide character or multi-byte value, irrespective of the direction of conversion.

The alternative when converting an array of compound strings is to use XmStringTableUnparse(), which takes the following form:

```
XtPointer *XmStringTableUnparse ( XmStringTable    table,
                                  Cardinal         table_count,
                                  XmStringTag      tag,
                                  XmTextType       tag_type,
                                  XmTextType       output_type,
                                  XmParseTable     parse_table,
                                  Cardinal         parse_table_count,
                                  XmParseModel     parse_model)
```

This routine differs from XmStringUnparse() only in that it takes XmStringTable and counter parameters, and returns an array. As for XmStringUnparse(), XmStringTableUnparse() returns allocated memory: you need to free each of the elements in the returned array, as well as the array itself. We could equally have written the dump_file() routine to use XmStringTableUnparse() as follows:

```
char **output;

output = (char **) XmStringTableUnparse (xms_array, xms_count,
                                         NULL, XmCHARSET_TEXT, XmCHARSET_TEXT,
                                         &map, 1, XmOUTPUT_ALL);
...
for (i = 0; i < xms_count; i++) {
    (void) fprintf (fptr, "%s\n", output[i]);
    XtFree (output[i]);
}

XtFree ((char *) output);
```

Parse Procedures

The problems with an XmParseMapping as described so far are that the mapping between the input and output sequences are static, the parse state as far as the application is

concerned is context free, and there is no opportunity to dynamically move the current input pointer around if we decide to skip some sequence of data. For example, we might want to skip a particular field when converting a formatted line read from a file because we don't want to show it to the user. For this kind of task, the simple XmNsubstitute resource is insufficient. We need an XmParseProc. The specification of an XmParseProc is as follows:

```
typedef XmIncludeStatus (*XmParseProc) ( XtPointer        *in_out,
                                         XtPointer        text_end,
                                         XmTextType       text_type,
                                         XmString         locale_tag,
                                         XmParseMapping   entry,
                                         int              pattern_length,
                                         XmString         *str_include,
                                         XtPointer        call_data)
```

The *in_out* parameter points to the current location within the input stream when the procedure is invoked. The procedure can move the pointer to reset the location where parsing is to continue after the routine finishes. *text_end* initially points to the end of the in_out string, but again the procedure can move the location to indicate where parsing is to continue from after the mapping has been applied to the current input. The *text_type* parameter is the type of the input stream, and is one of XmCHARSET_TEXT, XmWIDECHAR_TEXT, XmMULTIBYTE_TEXT. The *locale_tag* parameter specifies the tag to be used in creating the result of the mapping. *entry* is the XmParseMapping associated with the current matched input. *pattern_length* is the number of bytes remaining to be parsed at the address specified by *in_out*. If the XmParseProc wishes to insert a compound string into the output at this juncture, it returns it at the *str_include* address. Lastly, the application can arrange to pass data to the routine through the *client_data* parameter, thus providing whatever context the function needs to perform its task. The call_data is specified as a parameter to the convenience routine (XmStringParseText() or XmStringTableParseStringArray()) which is currently controlling the parsing process.

As an example, consider converting a UNIX /etc/passwd file to an array of compound strings, except that the password field is to be hidden. As before, we map colon characters to tab components, except that this time we skip the second field in each line of the file. We add a second parse mapping object to handle newlines - not because we want to map newlines to anything, but because it is a convenient place to reset the field counter as the end of the line is reached. The address of the field counter is passed as client_data to the parse routines so that they can inspect and manipulate the value. Example 25-8 shows how this can be done using XmParseProc routines.

Example 25-8 The load_filtered_passwd() routine.

```
/* load_filtered_passwd(): converts the /etc/passwd file into
** a multi-column list format, skipping the password field.
*/
```

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

/* reset_field(): used to reset the field context
** when a newline is encountered in the input
*/
XmIncludeStatus reset_field ( XtPointer          *in_out,
                              XtPointer          text_end,
                              XmTextType         text_type,
                              XmStringTag        locale_tag,
                              XmParseMapping     parse_mapping,
                              int                pattern_length,
                              XmString           *str_include,
                              XtPointer          call_data)
{
    /* client data is a pointer to the field counter */
    int *field_counter = (int *) call_data;

    /* A newline encountered.
    **
    ** Trivial: we reset the field counter for this line
    ** and terminate the parse sequence
    */

    *field_counter = 0;

    return XmTERMINATE;
}

/* filter_field(): throws away the second (password) field
** and maps colon characters to tab components.
*/
XmIncludeStatus filter_field ( XtPointer          *in_out,
                               XtPointer          text_end,
                               XmTextType         text_type,
                               XmStringTag        locale_tag,
                               XmParseMapping     parse_mapping,
                               int                pattern_length,
                               XmString           *str_include,
                               XtPointer          call_data)
{
    /* client data is a pointer to the field counter */
    int *field_counter = (int *) call_data;
    char *cptr = (char *) *in_out;

    /* Skip this colon */
    cptr += pattern_length;

    /* If this is the first colon
    ** then skip the input until after the second.
    ** Otherwise, we return a TAB component
    */
```

```
    if (*field_counter == 0) {
        /* Skip over the next colon */
        while (*cptr != ':') cptr++;
            cptr++;
    }

    *str_include = XmStringComponentCreate (XmSTRING_COMPONENT_TAB,
                                            0, NULL);

    *in_out = (XtPointer) cptr;
    *field_counter = *field_counter + 1;

    return XmINSERT;
}

XmString *load_filtered_passwd (Widget list, char *file, int *count)
{
    XmParseMapping     map[2];
    FILE               *fptr;
    char               buffer[256];
    Arg                args[8];
    char               *cptr;
    XmString           *xms_array = (XmString *) 0;
    int                xms_count = 0;
    /* Used as client data to the XmParseProc routines */
    int                field_count = 0;
    int                n;

    *count = 0;

    if ((fptr = fopen (file, "r")) == (FILE *) 0) {
        return NULL;
    }

    /* Set up an XmParseProc to handle colons */
    n = 0;
    XtSetArg (args[n], XmNpattern, ":");                  n++;
    XtSetArg (args[n], XmNpatternType, XmCHARSET_TEXT);   n++;
    XtSetArg (args[n], XmNincludeStatus, XmINVOKE);       n++;
    XtSetArg (args[n], XmNinvokeParseProc, filter_field); n++;
    map[0] = XmParseMappingCreate (args, n);

    /* Set up an XmParseProc to handle newlines */
    n = 0;
    XtSetArg (args[n], XmNpattern, "\n");                 n++;
    XtSetArg (args[n], XmNpatternType, XmCHARSET_TEXT);   n++;
    XtSetArg (args[n], XmNincludeStatus, XmINVOKE);       n++;
    XtSetArg (args[n], XmNinvokeParseProc, reset_field); n++;
    map[1] = XmParseMappingCreate (args, n);

    while ((cptr = fgets (buffer, 255, fptr)) != (char *) 0) {
        xms_array = (XmString *) XtRealloc ((char *) xms_array,
                            (xms_count + 1) * sizeof (XmString));
```

```
            xms_array[xms_count] = XmStringParseText (cptr,
                                    NULL, NULL, XmCHARSET_TEXT,
                                    map, 2, &field_count);

            xms_count++;
        }

        (void) fclose (fptr);

        XmParseMappingFree (map[0]);
        XmParseMappingFree (map[1]);

        *count = xms_count;

        return xms_array;
    }
```

Figure 25-9 shows the result of replacing the `load_file`() routine of Example 25-5 with the `load_filtered_passwd`() function:

Figure 25-9 Output of the load_filtered_passwd() routine

Rendering Compound Strings

Motif always renders compound strings automatically within its widgets, so you should never find yourself in a situation where you need to render a compound string manually. However, if you are writing your own widget, you may need to incorporate the same type of functionality. Motif provides three functions that render compound strings:

```
XmStringDraw()
XmStringDrawImage()
XmStringDrawUnderline()
```

From Motif 1.2, all of these routines use the X11R5 text output routines when necessary, to ensure that the text is rendered correctly for the current locale.

The most basic rendering function is `XmStringDraw()`, which takes the following form*:

```
void XmStringDraw ( Display          *display,
                    Window           window,
                    XmRenderTable    renderTable,
```

```
XmString          string,
GC                gc,
Position          x,
Position          y,
Dimension         width,
unsigned char     alignment,
unsigned char     layout_direction,
XRectangle        *clip)
```

As you can see, the function requires a great deal of information to actually render the string. If you are rendering into a widget, you can specify the *display* and *window* using XtWindow() and XtDisplay(). Since a gadget does not have a window, you must use XtWindowOfObject() with a gadget. The *renderTable* parameter can be constructed using any of the functions described in Chapter 24, *Render Tables*, or you can retrieve a render table from a widget using XtVaGetValues().

The function also requires a graphics context (GC) so that certain rendering attributes such as color can be applied. A graphics context is generally not available through a widget, so you have to get one at the Xlib level. If you are writing your own widget, you can probably use a GC that is cached by Xt and returned by XtGetGC() (see Volume 4, *Xlib Programming Manual*). Also, if you are writing your own widget, you may want to consider exposing the GC to the programmer in the form of a resource.

The *x*, *y*, and *width* parameters specify the coordinates and width of the rectangle that contains the compound string. The *width* parameter is used only for alignment purposes. There is no height parameter because the render table may specify fonts that are unknown in size and whose heights are too variable. The *clip* parameter defines the drawing boundary; you can pass NULL to indicate that the rendering should not be clipped.

The *alignment* parameter can be set to one of the following values:

```
XmALIGNMENT_BEGINNING     XmALIGNMENT_CENTER     XmALIGNMENT_END
```

The value identifies the justification for the text. The effect of the value is modified by the value of the *layout_direction* parameter, which can be set to either XmSTRING_DIRECTION_L_TO_R or XmSTRING_DIRECTION_R_TO_L.

The function XmStringDrawImage() is to XmStringDraw() as XDrawString() is to XDrawImageString().The difference is that the image routines overwrite the background even in places where the font does not set bits in the character image, while the other routines only render foreground pixels.

The XmStringDrawUnderline() routine takes the same parameters as XmStringDraw() with one addition. The last parameter specifies the portion of the

* In Motif 1.2, XmStringDraw() takes an XmFontList parameter. In Motif 2.0, the XmFontList is obsolete, and has been replaced with an XmRenderTable. For backwards compatibility, the XmFontList type is implemented as a skeleton XmRenderTable.

compound string that should be underlined. A compound string can be wholly or partially underlined depending on whether the last parameter specifies the entire compound string or only a substring of the *string* parameter.

It may be necessary to get dimensional information about a compound string in order to know where to place it within the window when it is drawn. You may also want this data to determine the optimal or desired width and height of a widget in case you have to provide a geometry callback method. When a call to XtQueryGeometry() is made, a widget that contains compound strings may need to tell the calling function the dimensions it needs to render its compound strings adequately. Motif provides the following routines to help you determine compound string dimensions:

```
Dimension   XmStringBaseline (XmRenderTable renderTable, XmString string)
void        XmStringExtent (XmRenderTable renderTable, XmString string,
                            Dimension *width, Dimension *height)
Dimension   XmStringHeight (XmRenderTable renderTable, XmString string)
Dimension   XmStringWidth (XmRenderTable renderTable, XmString string)
```

Each of these functions takes *renderTable* (XmRenderTable) and *string* (XmString) parameters. The render table is dependent on the widget associated with the string, but there is no requirement that you must use a string that is associated with a widget. If you just want to get the dimensions of a particular compound string rendered using an arbitrary font or font set, you can create a render table manually, as described in Chapter 24, *Render Tables*.

XmStringBaseline() returns the number of pixels between the top of the character box and the baseline of the first line of text in the compound string. XmStringWidth() and XmStringHeight() return the width and height, respectively, for the specified compound string. XmStringExtent() takes two additional parameters, *width* and *height*. These arguments return the width and height in pixels of the smallest rectangle that encloses the compound string specified in *string*.

Summary

Compound strings can be useful for creating multi-line or multi-font text for widgets such as Labels, PushButtons, and ToggleButtons. Compound strings were also designed to help in making internationalized applications, but this functionality has basically been made obsolete by the addition of internationalization features in X11R5. Since Motif applications have to use compound strings to display most textual data, the trick to developing an internationalized application is to use compound strings without interfering with lower-level X internationalization functionality.

The best practice is to specify compound string and rendition resources in resource files, so that you can have a separate file for each language that is supported by your application. If you have to create compound strings in an application, you should use

`XmStringCreateLocalized()` or specify the `XmFONTLIST_DEFAULT_TAG` rendition tag to ensure that the strings are interpreted and rendered in the current locale.

In this chapter:
- *Handling Signals in X11R5*
- *Handling Signals in X11R6*
- *Summary*

26

Signal Handling

This chapter describes the techniques which should be adopted in order to handle UNIX signals within an X-based application. It is not a lesson in signal handling per se: if you are unsure what a signal is or how to handle one in an ordinary non-X-based application, you should consult your operating system or programming language manuals.

In X11R5, handling signals safely could be problematic. UNIX signals are delivered asynchronously in the context of the currently running process. On receipt of a signal, your application branches immediately to any handlers which have been set up to process the signal concerned, without any consideration to the application state or context. That is, regardless of whatever state the application is in, a new function context is pushed onto the process stack in order to immediately handle the signal. This scheme of operations can severely interfere with the transmission and processing of X protocol messages between the X server and the application client, because the signal could potentially be delivered right in the middle of an X call which is manipulating the event queue. Any attempt by the signal handler to call further X routines in these circumstances might garbage any messages which are in progress. Although the probability of this interference is small, because signals can arise from a variety of reasons which in many cases are from events external to the application, mission critical applications have to guard against the possibility by incorporating carefully constructed code to process potentially dangerous signals safely and cleanly. This is particularly important because the default action associated with some signal types causes the termination of the application, and thus we have no option but to install our own handlers to override the system default.

The solution to the problem is to ensure that the processing of received signals is delayed until such time as the application knows that it is safe to do so*. In practical terms, this means that we rewrite our X event loop to take into account the possible receipt of a signal, and we only process the signal when we know that the event queue is stable. This usually involves two signal handling routines. The first routine is installed as the normal signal

* It should be noted that BSD-style UNIX systems do provide a system call that effectively suspends signal delivery, but it would be too costly to invoke this routine for each Xlib call. Furthermore, it is considered inappropriate for X, a windowing system that is designed to be independent of the operating system, to adopt this system-specific solution.

handler, but it does nothing more than set a flag on receipt of the signal. The second routine performs the actual signal processing; we call this at a later date when it is safe to do so.

In X11R6, the task is much simplified because the toolkit has been rewritten in order to handle signals safely. There are new toolkit procedures which we can call that are specifically designed to ensure that the receipt and handling of a signal does not interfere with the X queue processing.

In the discussion which follows, we present both X11R5- and X11R6-style signal handling. In the case of X11R5, the issue is complicated by the fact that an application can decide to handle events in two different ways, either using the lower level Xlib mechanisms, or through the higher level event procedures provided in the X toolkit.

Handling Signals in X11R5

Handling Signals using Xlib

An application that uses Xlib gets events from the server using a function like `XNextEvent()`. This function reads the next event in the queue and fills an `XEvent` data structure that describes various things about the event, such as the window associated with it, the time it took place, the event type, and so on. When the function returns, the event has been delivered and it is up to the application to decide what to do next. The following code fragment demonstrates a simplified view of Xlib event handling:

```
void sigchld_handler(int);

main_event_loop ()
{
    ...
    signal (SIGCHLD, sigchld_handler);

    while (1) {
        XNextEvent (display, &event);
        switch (event.type) {
            case ConfigureNotify: /*...*/ break;
            case Expose:          /*...*/ break;
            case ButtonPress:     /*...*/ break;
            case EnterWindow:     /*...*/ break;
            case LeaveWindow:     /*...*/ break;
            case MapNotify:       /*...*/ break;
            ...
        }
    }
}
```

If the operating system decides to deliver a `SIGCHLD` signal, the signal can arrive at any time, possibly inside any of the case statements or even inside the call to `XNextEvent()`. The signal handler for the signal is called automatically by the operating system. If the

signal handler makes any Xlib calls, you have no way of knowing if it is doing so at a time when another Xlib call is being sent to the X server. The solution is to have the signal handler do nothing but set a flag to indicate that the signal has been delivered. Then, just before the call to `XNextEvent()`, the event loop can check the flag to determine whether or not to call another function that actually processes the signal. This new design is shown in the following code fragment:

```
static int sigchld_delivered;
void sigchld_handler(...), real_sigchld_handler(int);

main_event_loop()
{
    ...
    signal (SIGCHLD, real_sigchld_handler);

    while (1) {
        /* it's safe to handle signals that may have been delivered */
        if (sigchld_delivered > 0) {
            /* add other parameters as necessary */
            sigchld_handler (SIGCHLD);
            sigchld_delivered--;
        }
        XNextEvent (display, &event);
        switch (event.type) {
            case ConfigureNotify: /*...*/ break;
            case Expose:          /*...*/ break;
            case ButtonPress:     /*...*/ break;
            case EnterWindow:     /*...*/ break;
            case LeaveWindow:     /*...*/ break;
            case MapNotify:       /*...*/ break;
            ...
        }
    }
}
```

All that `real_sigchld_handler()` does is increment the `sigchld_delivered` flag, as shown in the following fragment:

```
/* additional parameters differ between BSD and SYSV */
void real_sigchld_handler (int sig)
{
    sigchld_delivered++;
}
```

The actual `sigchld_handler()` routine can do whatever it needs to do, including call Xlib routines, since it is only called when it is safe to do so. You should note that `XNextEvent()` waits until it reads an event from the X server before it returns, so handling the signal may take a long time if the program is waiting for the user to do something.

These code fragments demonstrate the general design for handling signals in a rudimentary way. In a real application, the actual signal handler would probably need access to all of the parameters passed to the original signal handling function. One example of this situation

would be a signal handler that displays the values of all its parameters in a dialog box. You can't change anything on the display using the original signal handler because it would require making Xlib calls, so you have to save the parameters until the real signal handler is called. To save the parameters, you could define a data structure that contains fields for all of the parameters. The original signal handler could allocate a new structure and fill it in each time a signal is delivered.* When the real signal handler is called, it can access the data structure and create a dialog using the appropriate Xlib calls.

Handling Signals in Xt

Since this is a book on Motif and Motif is based on Xt, the next step is to find a solution that is appropriate for Xt-based applications. In Xt, you typically don't read events directly from the X server using `XNextEvent()` and then branch on the event type to decide what to do next. Instead, Xt provides `XtAppMainLoop()`; the code for this function is below:

```
void XtAppMainLoop (XtAppContext app_context)
{
    XEvent event;

    for (;;) {
        XtAppNextEvent (app_context, &event);
        XtDispatchEvent (&event);
    }
}
```

Since the event processing loop is internal to the Xt toolkit, we don't have the opportunity to insert a check to see if any signals have been delivered, as we did with Xlib. There are various ways to handle this problem. We could write our own event processing loop and include code that tests for the delivery of a signal. One problem with this solution is that it bypasses a standard library routine. We want to ensure upwards compatibility with future versions of Xt, and if we write our own routine, we risk losing any functionality that might be introduced later.

Even though it is unlikely that `XtAppMainLoop()` will change in the future, we should find another way to solve the problem. Clearly, the desired effect is to get Xt to notify us just before it's going to call `XNextEvent()`, since this is the window of opportunity where it is safe for a signal handler to make Xlib or Xt calls. It just so happens that Xt provides two methods that do what we want: work procedures and timers.

A work procedure is a function that is called by Xt when it does not have any events to process. Although an application can register multiple work procedures, the procedures are processed one at a time, with the most recent one being invoked first. We can solve the signal handler problem using a work procedure because most applications spend a fair bit of time waiting for the user to generate events. In the signal handler, we register a work

* As we will discuss later, there can also be problems with memory allocation in a signal handler.

procedure using XtAppAddWorkProc(). When the application is idle, Xt invokes the work procedure, which does the real work of handling the signal. The following code fragment uses this approach:

```
XtAppContext app;
static void real_reset(int);
static Boolean reset(XtPointer);

main (int argc, char *argv[])
{
    ...
    signal (SIGCHLD, real_reset);
    ...
}

/* reset() -- a program died... */
static void real_reset (int unused)
{
    int pid, i;

    #ifdef    SYSV
        int status;
    #else  /* SYSV */
        union wait status;
    #endif /* SYSV */

    if ((pid = wait (&status)) == -1)
        /* an error of some kind (fork probably failed); ignore it */
        return;

    (void) XtAppAddWorkProc (app, reset, NULL);
}

static Boolean reset (XtPointer client_data)
{
    /* handle anything Xt/Xlib-related that needs to be done now */
    ...
    return True; /* remove the work procedure from the list */
}
```

This example assumes that the application forks off a new process at some point. When the child eventually exits, the parent is sent a SIGCHLD signal, at which point the application branches directly to the real_reset() signal handler. This routine reaps the child using wait() and then adds a work procedure using XtAppAddWorkProc(). (The function normally returns a work procedure ID, but we're not interested in it here.) When Xt does not have any events to process, it calls reset(). This routine can perform any other tasks necessary for handling the signal, such as calling Xlib routines, popping up dialogs, or anything it likes.

If the application is waiting for events when it receives the signal, the work procedure is invoked almost immediately after the actual signal handler. However, if the application is

in a callback routine handling an event, the work procedure is not called until control is passed back to the event loop. While it's true that there may be some delay between the time that the signal is delivered and the time that it is actually processed, the delay is usually small enough that an application doesn't need to worry about it. If timing is critical, you can always set a global signal flag when the signal is received, and then test that variable in critical sections of your code to see if the signal has been delivered.

An Example

The signal handling problem can also be solved with a timer, using the same approach as with a work procedure. Example 26-1 demonstrates the use of a timer in a more realistic application.*The program displays an array of DrawnButtons that start application programs. While an application is running, the associated button is insensitive, so that the user can only run one instance of the application. When the application exits, the button is reactivated, so that the user can select it again.

Example 26-1 The app_box.c program

```
/* app_box.c -- make an array of DrawnButtons that, when activated,
** executes a program. When the program is running, the drawn button
** associated with the program is insensitive. When the program dies,
** reactivate the button so the user can select it again.
*/

#include <Xm/DrawnB.h>
#include <Xm/RowColumn.h>
#include <signal.h>

#ifndef   SYSV
#include <sys/wait.h>
#else  /* SYSV */
#define SIGCHLD SIGCLD
#endif /* SYSV */

#define MAIL_PROG "/bin/mail"

typedef struct {
    Widget    drawn_w;
    char      *pixmap_file;
    char      *exec_argv[6]; /* 6 is arbitrary, but big enough */
    int       pid;
} ExecItem;

ExecItem prog_list[] = {
    { NULL, "terminal", { "xterm", NULL }, 0 },
    { NULL, "flagup", { "xterm", "-e", MAIL_PROG, NULL }, 0 },
    { NULL, "calculator", { "xcalc", NULL }, 0 },
```

* In X11R6, the routine XtVaAppInitialize() is deprecated, and should be replaced with XtVaOpenApplication().

```
    { NULL, "xlogo64", { "foo", NULL }, 0 }
};

XtApp Context app;/* application context for the whole program */
GC    gc;         /* used to render pixmaps in the widgets */

void reset (int);
void reset_btn (XtPointer, XtIntervalId *);
void redraw_button (Widget, XtPointer, XtPointer);
void exec_prog (Widget, XtPointer, XtPointer);

main (int argc, char *argv[])
{
    Widget  toplevel, rowcol;
    Pixmap  pixmap;
    Pixel   fg, bg;
    Arg     args[8];
    int     i, n;

    /* we want to be notified when child programs die */
    (void) signal (SIGCHLD, reset);

    XtSetLanguageProc (NULL, NULL, NULL);

    /* Since this is an X11R5 example...
    ** For X11R6, use XtVaOpenApplication()
    */
    toplevel = XtVaAppInitialize (&app, "Demos", NULL, 0, &argc, argv, NULL,
                                  NULL);

    n = 0;
    XtSetArg (args[n], XmNorientation, XmHORIZONTAL); n++;
    rowcol = XmCreateRowColumn (toplevel, "rowcol", args, n);

    /* get the foreground and background colors of the rowcol
    ** so the gc (DrawnButtons) will use them to render pixmaps.
    */
    XtVaGetValues (rowcol,
    XmNforeground, &fg,
    XmNbackground, &bg, NULL);
    gc = XCreateGC (XtDisplay (rowcol),
                    RootWindowOfScreen (XtScreen (rowcol)),
                    NULL, 0);
    XSetForeground (XtDisplay (rowcol), gc, fg);
    XSetBackground (XtDisplay (rowcol), gc, bg);

    for (i = 0; i < XtNumber (prog_list); i++) {
        /* the pixmap is taken from the name given in the structure */
        pixmap = XmGetPixmap (XtScreen (rowcol),
                              prog_list[i].pixmap_file, fg, bg);
        /* Create a drawn button 64x64 (arbitrary, but sufficient)
        ** shadowType has no effect till pushButtonEnabled is false.
        */
        n = 0;
```

```
        XtSetArg (args[n], XmNwidth, 64);                       n++;
        XtSetArg (args[n], XmNheight, 64);                      n++;
        XtSetArg (args[n], XmNpushButtonEnabled, True);         n++;
        XtSetArg (args[n], XmNshadowType, XmSHADOW_ETCHED_OUT); n++;
        prog_list[i].drawn_w = XmCreateDrawnButton (rowcol,
                                                    "dbutton",
                                                    args, n);
        XtManageChild (prog_list[i].drawn_w);

        /* if this button is selected, execute the program */
        XtAddCallback (prog_list[i].drawn_w, XmNactivateCallback,
                       exec_prog, (XtPointer) &prog_list[i]);
        /* when the resize and expose events come, redraw pixmap */
        XtAddCallback (prog_list[i].drawn_w, XmNexposeCallback,
                       redraw_button, (XtPointer) pixmap);
        XtAddCallback (prog_list[i].drawn_w, XmNresizeCallback,
                       redraw_button, (XtPointer) pixmap);
    }

    XtManageChild (rowcol);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

/* redraw_button() -- draws the pixmap into its DrawnButton
** using the global GC. Get the width and height of the pixmap
** being used so we can either center it in the button or clip it.
*/
void redraw_button (Widget button, XtPointer client_data,
XtPointer call_data)
{
    Pixmap       pixmap = (Pixmap) client_data;
    int          unused, destx, desty;
    unsigned int srcx, srcy, pix_w, pix_h;
    int          drawsize, border;
    Dimension    bdr_w, w_width, w_height;
    short        hlthick, shthick;
    Window       root;
    XmDrawnButtonCallbackStruct *cbs =
                                (XmDrawnButtonCallbackStruct *) call_data;

    /* get width and height of the pixmap. don't use srcx and root */
    XGetGeometry (XtDisplay (button), pixmap, &root, &unused,
                  &unused, &pix_w, &pix_h, &srcx, &srcx);
    /* get the values of all the resources that affect the entire
    ** geometry of the button.
    */
    XtVaGetValues (button,
                   XmNwidth, &w_width,
                   XmNheight, &w_height,
                   XmNborderWidth, &bdr_w,
                   XmNhighlightThickness, &hlthick,
                   XmNshadowThickness, &shthick,
                   NULL);
```

```
    /* calculate available drawing area, width first */
    border = bdr_w + hlthick + shthick;
    /* if window is bigger than pixmap, center it; else clip pixmap */
    drawsize = w_width - 2 * border;
    if (drawsize > pix_w) {
        srcx = 0;
        destx = (drawsize - pix_w) / 2 + border;
    }
    else {
        srcx = (pix_w - drawsize) / 2;
        pix_w = drawsize;
        destx = border;
    }
    drawsize = w_height - 2 * border;
    if (drawsize > pix_h) {
        srcy = 0;
        desty = (drawsize - pix_h) / 2 + border;
    }
    else {
        srcy = (pix_h - drawsize) / 2;
        pix_h = drawsize;
        desty = border;
    }

    XCopyArea (XtDisplay (button), pixmap, cbs->window, gc, srcx, srcy, pix_w,
                                            pix_h, destx, desty);
}

/* exec_proc() -- the button has been pressed; fork() and call
** execvp() to start up the program. If the fork or the execvp
** fails (program not found?), the sigchld catcher will get it
** and clean up. If the program is successful, set the button's
** sensitivity to False (to prevent the user from exec'ing again)
** and set pushButtonEnabled to False to allow shadowType to work.
*/
void exec_prog (Widget drawn_w, XtPointer client_data,
                                        XtPointer call_data)
{
    ExecItem *program = (ExecItem *) client_data;
    XmDrawnButtonCallbackStruct *cbs =
                                (XmDrawnButtonCallbackStruct *) call_data;

    switch (program->pid = fork ()) {
        case 0:  /* child */
            execvp (program->exec_argv[0], program->exec_argv);
            perror (program->exec_argv[0]); /* command not found? */
            _exit (255);
        case -1:
            printf ("fork() failed.\n");
    }

    /* The child is off executing program... parent continues */
    if (program->pid > 0) {
        XtVaSetValues (drawn_w, XmNpushButtonEnabled, False, NULL);
```

```
            XtSetSensitive (drawn_w, False);
        }
    }

    /* reset() -- a program died, so find out which one it was and
    ** reset its corresponding DrawnButton widget so it can be reselected
    */
    void reset (int unused)
    {
        int         pid, i;
        #ifdef    SYSV
            int         status;
        #else  /* SYSV */
            union wait  status;
        #endif /* SYSV */

        if ((pid = wait (&status)) != -1) {
            for (i = 0; i < XtNumber (prog_list); i++)
                if (prog_list[i].pid == pid) {
                    /* program died -- now reset item. But not here! */
                    XtAppAddTimeOut (app, (unsigned long) 0, reset_btn,
                                     (XtPointer) prog_list[i].drawn_w);
                    break;
                }
        }

        (void) signal (SIGCHLD, reset);
    }

    /* reset_btn() -- reset the sensitivity and pushButtonEnabled resources
    ** on the drawn button. This cannot be done within the signal
    ** handler or we might step on an X protocol packet since signals are
    ** asynchronous. This function is safe because it's called from a timer
    */
    void reset_btn (XtPointer closure, XtIntervalId *unused)
    {
        Widget drawn_w = (Widget) closure;

        XtVaSetValues (drawn_w, XmNpushButtonEnabled, True, NULL);
        XtSetSensitive (drawn_w, True);
        XmUpdateDisplay (drawn_w);
    }
```

The output of the program is shown in Figure 26-1.

Figure 26-1 Output of the app_box program

The program in Example 26-1 is almost identical in design to the code fragment that used a work procedure, but it is more like something you might actually write. The program uses DrawnButtons to represent different application programs. The idea is that when a button is pressed, the program corresponding to the image drawn on the button is run. The button turns insensitive for as long as the application is alive. When the user exits the program, the button's state is restored so the user can select it again.

Each button has a data structure associated with it that specifies the file that contains the icon bitmap, an `argv` that represents the program to be run, the process ID associated with the program's execution, and a handle to the button itself. The callback routine for each button spawns a new process, sets the button to insensitive, and immediately returns control to the main event loop. The process ID is saved in the button's data structure. When the external process terminates, a `SIGCHLD` signal is sent to the main program and the button is reset.

As a general note, it is crucial that you understand that the new process does not attempt to interact with the widgets in its parent application or read events associated with the same display connection as its parent process. Even though the child has access to the same data structures as the parent, it cannot use its parent's connection to the X server because multiple processes cannot share an X server connection. If a child process intends to interact with the X server, it must close its existing connection and open a new one.

In our application, we play it safe by running a completely new application using `execvp()`. This system call executes a program provided it can be found in the user's `PATH`, so we don't need to specify full pathnames to the applications. If the program cannot be found for whatever reason, the child process dies immediately and the `reset()` signal handler is called by the operating system.

The `reset()` signal handler is called whenever a child process dies. At this point, the child needs to be reaped and the state of the button needs to be reset. The `wait()` system call is used to reap the child; this routine can be called from within `reset()` because it doesn't make any Xlib calls. However, we cannot reset the button's state by calling `XtVaSetValues()` and `XtSetSensitive()` because these routines would ultimately result in Xlib calls. Therefore, rather than actually resetting the button in `reset()`, we call `XtAppAddTimeOut()` to install a timer routine. This Xt call is safe in a signal handler because it does not make any calls to Xlib; the timer is handled entirely on the client side.

`XtAppAddTimeOut()` registers a timer procedure that is called after a specified amount of time. Xt's main event processing loop takes care of calling the timer routine after the appropriate time interval. Since we have specified an interval of `0` for the `reset_btn()` timer, the routine is called immediately after the signal is received and control is passed back to the main event loop. The `reset_btn()` routine handles restoring the state of the DrawnButton, so that the user can run the associated application again.

In terms of signal handling, there is really one main difference between using a work procedure and using an interval timer. The work procedure is called as soon as the application is idle and waiting for input, while the timer is called after a specified interval.

Additional Issues

There are several loose ends that we need to address. One issue involves the way timers are implemented. You may be thinking, "Isn't a timer another signal in UNIX?" While the answer is yes, what is important is that Xt-timers are not implemented using UNIX signals, but instead using a feature of the `select()` system call. In this context, `select()` is used to determine if the X server is sending events to the application (although this function does not actually read any events). The last parameter to `select()` is a time interval that specifies how long the routine waits before returning whether there is anything to read. Setting this time interval allows Xt to implement what appears to be a timer. As long as there are events to read from the server, however, the timer is inactive, which is why a timer in Xt can only be set in terms of an interval, rather than as a real-time value. It is also why you should never rely on the accuracy of these timers.

Timers are not implemented using UNIX signals for the same reasons that we did not call `XtVaSetValues()` from within the `SIGCHLD` signal handler. It is also for this reason that you should not use UNIX-based functions such as `sleep()` or `setitimer()` to modify widgets or make Xlib calls. We don't mean to imply that you should not use these functions at all; it's just that the same restrictions apply to UNIX timers as they do to other UNIX signals. If you need to do any X or Xt-related function calls, don't do it from a signal handler. You should install a zero-length interval timeout function using `XtAppAddTimeOut()` and, when the toolkit invokes your function, call whatever X routines are necessary.Timers of this type are used frequently with clock programs and text widgets. In the case of a clock, the timer advances the second hand, while for a text widget, it causes the insertion cursor to flash.

Another loose end that needs to be tied up involves System V's handling of signals. In most modern versions of UNIX (derived from BSD UNIX), when a signal is delivered to an application, any system call that might be going on is interrupted, the signal handler is called, and when it returns, the system call is allowed to continue. For example, if you are reading in the text of a file using `read()` and a signal is sent to the application, the `read()` is suspended while the signal handler is called. After your signal handler returns, the `read()` is restarted and it returns the actual number of bytes read as if no signal had ever occurred. Under System V, all system calls are interrupted and return an error (with errno set to `EINTR`). In this case, all of the data read by the `read()` call is lost.

This situation is a problem in X because `read()` is used to read events from the X server. If `read()` fails because a signal is delivered, then the protocol that was being sent by the server is lost, as would be anything we were sending to the server, since the same is true for calls to `write()`. There really isn't anything you can do about this problem, except,

perhaps, for upgrading to a more modern version of UNIX. This problem does not exist with SVR4 or Solaris.

Even system calls in BSD-derived UNIX systems may have problems. If, for example, you call `read()` from a signal handler that interrupted another `read()`, you still might not get what you expected because `read()` is not *re-entrant*. A function that is re-entrant is one that can be called at any time, even while the function is already being executed.

We're pretty safe with the advice we've given so far, with one exception: calling `XtAppAddTimeOut()` or `XtAppAddWorkProc()` eventually requires the allocation of memory to add the new timer or work procedure to the respective list. If your application happens to be allocating memory when a signal is delivered and you try to add a timer or a work procedure, you could make another call to `alloc()`, which is the lowest-level routine that allocates memory from the system. Unless your version of UNIX has a re-entrant memory allocation system call, your memory stack may be corrupted.*There really isn't anything that you can do about these problems, and there are no official specifications anywhere in the X documents that even address these issues, so the best tactic is to minimize the exposure using timers or work procedures as described here.

Handling Signals in X11R6

In X11R6, three new functions are added in order to handle signals safely in the context of the X Toolkit Intrinsics event queue mechanisms.

XtAppAddSignal

Firstly, `XtAppAddSignal`() registers a handler with the toolkit which is to be invoked at the appropriate safe point in the Xt event mechanisms; the handler is permitted to call further X routines, in the knowledge that it is safe to do so at that juncture. `XtAppAddSignal`() returns an opaque handle, an `XtSignalId`, used to keep track of the registered handler. The full signature of the routine is as follows:

```
XtSignalId XtAppAddSignal ( XtAppContext          app,
                            XtSignalCallbackProc  handler,
                            XtPointer             client_data)
```

The *`handler`* parameter is the application-specific routine invoked in response to a signal. As usual, *`client_data`* is any application-specific data which you want your handler to be passed when invoked. The returned XtSignalId should be stored for subsequent later use. An `XtSignalCallbackProc` routine is defined as follows:

```
typedef void (*XtSignalCallbackProc) (XtPointer closure, XtSignalId *id)
```

* The GNU version of `malloc()` is re-entrant, so it is safe from this problem.

The application *handler* should be registered after the usual operating system signal() call. Note that the purpose of *handler* is not to catch the signal, but to perform any processing that is required in the knowledge that it is Xt-event-safe. A standard signal handling routine is still required to catch the signal itself. The following code fragment outlines the steps:

```
void my_safe_handler (XtPointer, XtSignalId *);
void my_signal_catcher ();

XtSignalId my_signal_id;
XtAppContext app;
...
/* standard operating system signal() call */
signal (SIGCHLD, my_signal_catcher);

/* register a handler to process the signal safely */
my_signal_id = XtAppAddSignal (app, my_safe_handler, NULL);
...
```

XtNoticeSignal

If we are still using the standard operating system signal() routines to catch a signal, and a separate application handler to perform safe processing, what then, you may ask, is the connecting piece of logic which makes the system work? The answer is the routine XtNoticeSignal(). We call this inside our normal operating system signal handling routine to inform Xt that the signal has arrived, secure in the knowledge that this Xt routine is the only intrinsics function we are guaranteed as to its signal safety. XtNoticeSignal() takes as a parameter the XtSignalId of the handler we wish to invoke in response to the current signal delivery. XtNoticeSignal() therefore has the following form:

```
void XtNoticeSignal (XtSignalId)
```

Following on from the previous example, we might sketch our signal catching routine as follows:

```
void my_signal_catcher (int signo)
{
    /* Perform non-X processing */
    ...

    /* Inform the intrinsics of signal arrival */
    XtNoticeSignal (my_signal_id);
}
```

XtNoticeSignal() therefore replaces the various calls to XtAppAddWorkProc() or XtAppAddTimeOut() within the signal catching routine, as recommended for the X11R5 examples. XtNoticeSignal() simply arranges to call the handler associated with the parameter XtSignalId at a safe point in the Xt event processing loop.

XtRemoveSignal

We can disassociate a handler from the X Intrinsics at any time by calling XtRemoveSignal(). We would do this if we are no longer interested in handling the given signal, or if we wanted to change the handler. The routine is passed as a parameter the XtSignalId returned from a previous call to XtAppAddSignal(). XtRemoveSignal() has the following prototype:

```
void XtRemoveSignal (XtSignalId)
```

The source of the signal must be disabled before calling XtRemoveSignal() in order to prevent possible race conditions. The following code fragment outlines the correct scheme of operations:

```
XtSignalId my_signal_id;
...
/* Disable the signal source */
signal (SIGCHLD, SIG_IGN);

/* Unregister the Intrinsics signal handler */
XtRemoveSignal (my_signal_id);
```

An Example

The following fragments of code represent the changes required to modify Example 26-1 to use the X11R6 signal notification mechanisms.* We add a new global variable for the XtSignalId returned by XtAppAddSignal(), we modify reset() to call XtNoticeSignal(), and we change reset_btn() to an XtSignalCallbackProc.

```
.....
XtAppContext  app;          /* application context for the whole program */
GC            gc;           /* used to render pixmaps in the widgets */
XtSignalId    sign_id;      /* X11R6 Signal Registration handle */
ExecItem      *reap = NULL; /* the item which has just terminated */
....
/* reset_btn is now an XtSignalCallbackProc */
void reset_btn (XtPointer, XtSignalId *);
...

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

    /* we want to be notified when child programs die */
    (void) signal (SIGCHLD, reset);

    XtSetLanguageProc (NULL, NULL, NULL);
    /* Now that we are using X11R6...*/
```

* XtVaOpenApplication(), the SessionShell widget class, XtAppAddSignal(), and XtNoticeSignal() are only available in X11R6. XtVaAppInitialize() is considered deprecated in X11R6.

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

    /* Register our safe signal handler with the Intrinsics */
    /* Pass the address of the reap pointer as client data */
    sign_id = XtAppAddSignal (app, reset_btn, (XtPointer) &reap);

    /* The rest of main() is exactly as before */
    n = 0;
    XtSetArg (args[n], XmNorientation, XmHORIZONTAL); n++;
    rowcol = XmCreateRowColumn (toplevel, "rowcol", args, n);

    ...
    XtAppMainLoop (app);
}

/* reset() -- a program died, so find out which one it was and
** reset its corresponding DrawnButton widget so it can be reselected
**
** The difference between this and the X11R5 example is the replacing
** of XtAppAddTimeOut() with XtNoticeSignal().
*/
void reset (int unused)
{
    int pid, i;
    #ifdef    SYSV
        int      status;
    #else  /* SYSV */
        union    wait status;
    #endif /* SYSV */

    reap = (ExecItem *) 0;

    /* Basically the same loop as X11R5, except for XtNoticeSignal()
    ** instead of the XtAppAddTimeOut() call
    */
    if ((pid = wait (&status)) != -1) {
        for (i = 0; i < XtNumber (prog_list); i++)
            if (prog_list[i].pid == pid) {
                /* program died -- now reset item. But not here! */
                /* Set up the client data for our signal procedure */
                reap = &prog_list[i];
                /* Inform Xt of signal arrival */
                XtNoticeSignal (sign_id);
                break;
            }
    }

    (void) signal (SIGCHLD, reset);
}

/* reset_btn() -- reset the sensitivity and pushButtonEnabled resources
** on the drawn button.
**
```

```
** For X11R6, we have simply changed the signature to that
** of an XtSignalCallbackProc, and pass a pointer to the
** reap ExecItem as client data, instead of a Widget.
*/
void reset_btn (XtPointer closure, XtSignalId *id)
{
    ExecItem **reap_ptr = (ExecItem **) closure;
    Widget     drawn_w  = (*reap_ptr)->drawn_w;

    XtVaSetValues (drawn_w, XmNpushButtonEnabled, True, NULL);
    XtSetSensitive (drawn_w, True);
    XmUpdateDisplay (drawn_w);
}
```

Summary

Up until the release of X11R6, the official advice of the X Consortium was that you should not mix signals with X applications. However, there are cases where you must choose the lesser of two evils. The need for signal handling exists and cannot simply be ignored. In X11R6, Xt has support for registering signal handlers, so this problem is no longer a critical issue as far as the support offered by the X Toolkit is concerned. In an X11R5 environment, the approaches given in this chapter should serve you well most of the time.

The most important lesson to learn from this chapter may well be that UNIX signals are potentially dangerous to X applications, or indeed any sort of program that relies on a client-server protocol. They can also be a problem for system calls in an extremely sensitive or real-time environment. The issue is not X specific; X just happens to be an environment where the issue arises. Whenever the operating system can interrupt the client side (or the server side, for that matter), you should be prepared to consider those cases where the client-server protocol may be breached. Using the X11R6 signal notification scheme, the toolkit takes upon itself the task of ensuring X protocol integrity.

In this chapter:
- *Help Systems*
- *Working Dialogs*
- *Dynamic Message Symbols*
- *Summary*

27

Advanced Dialog Programming

This chapter describes some Motif features that have not been described, or at least not completely, in earlier chapters. The topics, which all deal with dialogs, include the creation of multi-stage help systems, the development of WorkingDialogs that allow the user to interrupt long-running tasks, and a method for dynamically changing the pixmaps displayed in a dialog.

In one sense, this chapter isn't about dialogs at all, but about various aspects of X programming that become most evident when working with dialogs. Here we address some issues involved in creating multi-stage help systems, we show you how to create a WorkingDialogs that allows the user to interrupt a long-running task, and we describe a method for dynamically changing the pixmaps that are displayed in a dialog. These topics explore some of the most interesting problems in this book.

These topics take us deeper into the lower layers of X than anything we've discussed so far in this book. You should have a good basic understanding of X event-processing, as implemented both in Xlib and Xt. Otherwise, be prepared to refer frequently to Volume 1, *Xlib Programming Manual*, and Volume 4, *X Toolkit Intrinsics Programming Manual*, when faced with references to lower-level functions.

Help Systems

The *Motif Style Guide* doesn't have much to say about how help is presented to the user, although it does discuss the ways in which the user can request help from an application. The user can request help by selecting the *Help* button in a dialog box, by choosing help items from the *Help* menu in the MenuBar, or by pressing the HELP or F1 key on the keyboard. Help information should be presented clearly, so that it is accessible and beneficial to users. You should also maintain consistency in a help system, so that the user can become familiar with the style of help that you provide.

The easiest and most straightforward method of presenting help information involves creating an InformationDialog with the necessary text displayed as the

XmNmessageString. Example 27-1 demonstrates how to display a help dialog when the user presses the *Help* button in another dialog box.*

Example 27-1The simple_help.c program

```
/* simple_help.c -- create a PushButton that posts a dialog box
** that entices the user to press the help button. The callback
** for this button displays a new dialog that gives help.
*/

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

main (int argc, char *argv[])
{
    Widget         toplevel, button;
    XtAppContext   app;
    XmString       label;
    Arg            args[2];
    void           pushed(Widget, XtPointer, XtPointer);

    XtSetLanguageProc (NULL, NULL, NULL);
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv, NULL,
                                    sessionShellWidgetClass, NULL);
    label = XmStringCreateLocalized ("Push Me");
    XtSetArg (args[0], XmNlabelString, label);
    button = XmCreatePushButton (toplevel, "button",args, 1);
    XtAddCallback (button, XmNactivateCallback, pushed,
                           "You probably need help for this item.");
    XmStringFree (label);
    XtManageChild (button);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

#define HELP_TEXT "This is the help information.\nNow press 'OK'"

/* pushed() -- the callback routine for the main app's pushbutton. */
void pushed (Widget w, XtPointer client_data, XtPointer call_data)
{
    char      *text = (char *) client_data;
    Widget    dialog;
    XmString  t = XmStringCreateLocalized (text);
    Arg       args[5];
    int       n;
    void      help_callback(Widget, XtPointer, XtPointer);
    void      help_done(Widget, XtPointer, XtPointer);

    n = 0;
```

* XtVaAppInitialize() is considered deprecated in X11R6. XmStringGetLtoR() and XmMessageBoxGetChild() are deprecated in Motif 2.0 and later. XmStringGenerate() is only available from Motif 2.0 onwards.

```
    XtSetArg (args[n], XmNautoUnmanage, False); n++;
    XtSetArg (args[n], XmNmessageString, t);    n++;
    dialog = XmCreateMessageDialog (XtParent (w), "notice", args, n);
    XmStringFree (t);
    XtUnmanageChild (XtNameToWidget (dialog, “Cancel”));
    XtAddCallback (dialog, XmNokCallback, help_done, NULL);
    XtAddCallback (dialog, XmNhelpCallback, help_callback, HELP_TEXT);

    /* This also pops up the DialogShell parent */
    XtManageChild (dialog);
}

/* help_callback() -- callback routine for the Help button in the
** original dialog box that displays an InformationDialog based on the
** help_text parameter.
*/

void help_callback (Widget parent, XtPointer client_data, XtPointer call_data)
{
    char       *help_text = (char *) client_data;
    Widget     dialog;
    XmString   text;
    void       help_done(Widget, XtPointer, XtPointer);
    Arg        args[5];
    int        n;

    n = 0;
    text = XmStringGenerate ((XtPointer) help_text,
                             XmFONTLIST_DEFAULT_TAG,
                             XmCHARSET_TEXT,
                             NULL);
    XtSetArg (args[n], XmNmessageString, text); n++;
    XtSetArg (args[n], XmNautoUnmanage, False); n++;
    dialog = XmCreateInformationDialog (parent, "help", args, n);
    XmStringFree (text);

    XtUnmanageChild (XtNameToWidget (dialog, “Cancel”));
    XtSetSensitive (XtNameToWidget (dialog, “Help”), False);
    /* the OK button will call help_done() below */
    XtAddCallback (dialog, XmNokCallback, help_done, NULL);
    /* display the help text */
    XtManageChild (dialog);
}

/* help_done() -- called when user presses "OK" in dialogs. */
void help_done (Widget dialog, XtPointer client_data, XtPointer call_data)
{
    XtDestroyWidget (XtParent (dialog));
}
```

The main window contains a PushButton that posts a simple MessageDialog. This dialog, as you can tell from Figure 27-1, contains a *Help* button, that pops up an InformationDialog. This dialog is intended to provide help text for the user.

Figure 27-1 Output of simple_help.c

The callback routine for the *Help* button is installed using the `XmNhelpCallback`. This routine pops up an InformationDialog that contains some predefined text. Obviously, this text is for demonstration purposes only. We used `XmStringGenerate()` to display the text, instead of `XmStringCreateLocalized()`, since the help message contains newline characters. See Chapter 25, *Compound Strings*, for more information on how you can use compound strings.

The `XmNhelpCallback` resource serves as the callback for any widget that wishes to provide help information; every Motif widget has an `XmNhelpCallback` resource associated with it. Whenever the user presses the HELP key on the keyboard (if one exists and the X server is set up correctly* the `XmNhelpCallback` is invoked for the widget that has the keyboard focus. The F1 key also serves as a help key for compatibility with Microsoft Windows and to compensate for any computer that may not have a HELP key.†

If a widget does not have an `XmNhelpCallback` function installed, Motif climbs the widget tree, searching for the nearest ancestor that has a help callback. If you assign help callbacks to widgets, were commend that you provide specific help information for individual interface components, such as PushButtons, Lists, and Text widgets, and more general information for manager widgets. It is possible to design an elaborate context-

* Sun workstations do not necessarily generate the proper event when the HELP key is pressed, and your mileage may vary for other computers.

† The F1 key works by default, but it may be remapped to perform another function in the user's.*mwmrc* file.

sensitive help system for an application by installing help callback routines for the widgets in the interface and providing relevant help information throughout the hierarchy.

Although *simple_help.c* is rather contrived, we can use it to examine the different actions the user might take within a help system. You can think of the *Push Me* button as any widget in an application on which the user might want help. When the button is activated, the user is presented with a MessageDialog that undoubtedly requires help. The user can select the *Help* button or press the F1 or HELP keys to access the help information. Since the InformationDialog is modeless, as it should be, the user can either close the InformationDialog or the original MessageDialog.

Since the InformationDialog is a child of the MessageDialog, if the MessageDialog is destroyed, the InformationDialog is also destroyed. Similarly, if the MessageDialog is unmapped, so is the InformationDialog. In general, when you display an InformationDialog, you should remove it if the user unmanages, destroys, or otherwise disables the dialog from which it was posted because if the help dialog remains posted, it could confuse the user. By making the InformationDialog the child of the original dialog, you can let the parent-child interaction handle this behavior.

Multi-level Help

Developing a help system may involve providing multiple levels of help information. If the user has already posted an InformationDialog, it is possible to display an additional dialog if the user requests help in the original dialog. However, multiple help windows can confuse the user, so they should be avoided. A better solution is to display the new help text in the same InformationDialog, so that all of the help information is displayed in the same place. Example 27-2 shows new `help_callback()` and `help_done()` routines that implement this technique.*

Example 27-2 Sample routines to create a single help dialog

```
#define MAX_HELP_STAGES 3

char *help_text[3][5] = {
    {
        "You have reached the first stage of the help system.",
        "If you need additional help, select the 'More Help' button.",
        "You may exit help at any time by pressing 'Done'.",
        NULL
    },
    {
        "This is the second stage of the help system. There is",
        "more help available. Press 'More Help' to see more.",
        "Press 'Previous' to return to the previous help message,",
```

* `XmStringCreateLtoR`() and `XmMessageBoxGetChild`() are deprecated from Motif 2.0. `XmStringGenerate`() is only available from Motif 2.0 onwards.

```
        "or press 'Done' to exit the help system.",
        NULL
    },
    {
        "This is the last help message you will see on this topic.",
        "You may either press 'Previous' to return to the previous",
        "help level, or press 'Done' to exit the help system.",
        NULL
    }
};

/* help_callback() -- callback routine for the Help button in the
** original dialog box. The routine also serves as its own help
** callback for displaying multiple levels of help messages.
*/
void help_callback (Widget parent, XtPointer client_data, XtPointer call_data)
{
    static Widget    dialog = (Widget) 0; /* prevent multiple dialogs */
    XmString         text;
    char             buf[BUFSIZ], *p;
    static int       index;
    int              i;
    void             help_done(Widget, XtPointer, XtPointer);
    int              index_incr = (int) client_data;

    if (dialog && index_incr == 0) {
        /* user pressed Help button in MesageDialog again. We're
        ** already up, so just make sure we're visible and return.
        */
        XtManageChild (dialog);
        XMapRaised (XtDisplay (dialog), XtWindow (XtParent (dialog)));
        return;
    }

    if (dialog)
        index += index_incr; /* more/previous help; change index */
    else {
        /* We're not up, so create new Help Dialog */
        Arg args[5];
        int n;
        /* Action area button labels. */
        XmString done = XmStringCreateLocalized ("Done");
        XmString cancel = XmStringCreateLocalized ("Previous");
        XmString more = XmStringCreateLocalized ("More Help");
        n = 0;
        XtSetArg (args[n], XmNautoUnmanage, False);       n++;
        XtSetArg (args[n], XmNokLabelString, done);       n++;
        XtSetArg (args[n], XmNcancelLabelString, cancel); n++;
        XtSetArg (args[n], XmNhelpLabelString, more);     n++;
        dialog = XmCreateInformationDialog (parent, "help", args, n);
        /* pass help_done() the address of "dialog" so it can reset */
        XtAddCallback (dialog, XmNokCallback, help_done, (XtPointer) &dialog);
        /* if more/previous help, recall ourselves with increment */
```

```
        XtAddCallback (dialog, XmNcancelCallback, help_callback,
                                (XtPointer) -1);
        XtAddCallback (dialog, XmNhelpCallback, help_callback, (XtPointer) 1);
        /* If our parent dies, we must reset "dialog" to NULL! */
        XtAddCallback (dialog, XmNdestroyCallback, help_done,
                                (XtPointer) &dialog);
        XmStringFree (done);   /* once dialog is created, these */
        XmStringFree (cancel); /* strings are no longer needed. */
        XmStringFree (more);
        /* initialize index--needed for each new help stuff */
        index = 0;
    }

    /* concatenate help text into a single string with newlines */
    for (p = buf, i = 0; help_text[index][i]; i++) {
        p += strlen (strcpy (p, help_text[index][i]));
        *p++ = '\n';
        *p = 0;
    }
    text = XmStringGenerate ((XtPointer) buf,
                             XmFONTLIST_DEFAULT_TAG,
                             XmCHARSET_TEXT,
                             NULL);
    XtVaSetValues (dialog, XmNmessageString, text, NULL);
    XmStringFree (text); /* after set-values, free unneeded memory */
    /* If no previous help msg, set "Previous" to insensitive. */
    XtSetSensitive (XtNameToWidget (dialog, "Cancel"), index > 0);
    /* If no more help, set "More Help" insensitive. */
    XtSetSensitive (XtNameToWidget (dialog, "Help"),
                             index < MAX_HELP_STAGES - 1);
    /* display the dialog */
    XtManageChild (dialog);
}

/* help_done () -- callback used to set the dialog pointer
** to NULL so it can't be referenced again by help_callback().
** This function is called from the Done button in the help dialog.
** It is also our XmNdestroyCallback, so reset our dialog_ptr to NULL.
*/
void help_done (Widget dialog, XtPointer client_data, XtPointer call_data)
{
    Widget *dialog_ptr;

    if (!client_data) {
        /* destroy original MessageDialog */
        XtDestroyWidget (XtParent (dialog));
        return;
    }

    dialog_ptr = (Widget *) client_data;
    if (!*dialog_ptr)
        /* prevent unnecessarily destroying twice */
        return;
```

```
        /* this might call ourselves. */
        XtDestroyWidget (XtParent (dialog));

        *dialog_ptr = NULL;
    }
```

In our help system, each level has a new help string that needs to be displayed. All of the help text is displayed in the same InformationDialog. The dialog for the first level of help is shown in Figure 27-2.

Figure 27-2 Displaying multiple levels of help text

The `help_callback` routine addresses several problems that arise when dealing with the added complexity of a multi-level help system. Since many dialogs may be trying to pop up the same InformationDialog, the routine uses a static variable for the dialog to prevent multiple instances of the dialog. This variable allows the routine to keep track of when the dialog is active and when it is dormant.

The routine is conceptually recursive, in that it is used as the callback routine for the buttons in the help dialog. The `client_data` is used as an index into the `help_text` array. When this parameter is `0`, the routine was called by the original MessageDialog. Otherwise, the routine was invoked as a result of the user pressing the *Previous* button or the *More Help* button. In this case, the `index` is changed so that the help text changes.

If the InformationDialog has already been created and the user presses the *Help* button anyway, the dialog is remapped and raised to the top of the screen using `XMapRaised()`. If the parent dialog is unmapped or destroyed, the InformationDialog is also unmapped or destroyed. In order to maintain the correct state information, we install an `XmNdestroyCallback` to monitor the destruction of the InformationDialog. When the dialog is destroyed, we need to reset the handle to the dialog to `NULL` so that we cannot reference the destroyed dialog again from `help_callback()` the next time help is requested.

All of our help text is fairly short, but if you need to provide help text that longer, you may want to use a ScrolledText object in your help dialog. With a ScrolledText object, you can display text of an arbitrary length without worrying about screen real estate. This technique is explained in Chapter 7, *Custom Dialogs*.

Context-sensitive Help

Although the user can access the help system by using the HELP or F1 keys, this interface is somewhat cumbersome and it doesn't work for widgets like Labels that do not process input events.You can provide a more intuitive interface that allows the user to point-and-click directly on a widget to obtain help. The *Motif Style Guide* refers to this style of help as *context-sensitive* help.

Context-sensitive help is make possible by the `XmTrackingEvent()` routine, which takes the following form:

```
Widget XmTrackingEvent ( Widget      widget,
                         Cursor      cursor,
                         Boolean     confine_to,
                         XEvent      *event)
```

The routine invokes a server-grab on the pointer, changes the pointer shape to that specified by the *cursor* parameter, and waits until the user presses a mouse button. The routine returns the widget on which the user pressed the button. If the *confine_to* parameter is `True`, the cursor is confined to the window of the specified *widget*. This window is also used as the owner of the pointer grab. The *event* parameter returns the actual event performed by the user.

An application usually provides context-sensitive help through an item on the *Help* menu. Example 27-3 shows the `query_for_help()` callback routine that could be used for such a menu item.

Example 27-3 The query_for_help() routine

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

Widget toplevel;

void query_for_help (Widget widget, XtPointer client_data, XtPointer call_data)
{
    Cursor                cursor;
    Display               *display;
    Widget                help_widget;
    XmAnyCallbackStruct   *cbs, *newcbs;
    XEvent                event;

    cbs = (XmAnyCallbackStruct *) call_data;
    display = XtDisplay (toplevel);
    cursor = XCreateFontCursor (display, XC_hand2);
    help_widget = XmTrackingEvent (toplevel, cursor, True, &event);

    while (help_widget != NULL) {
        if (XtHasCallbacks (help_widget, XmNhelpCallback) ==
                                                XtCallbackHasSome)
        {
            newcbs->reason = XmCR_HELP;
```

```
                newcbs->event = &event;
                XtCallCallbacks (help_widget, XmNhelpCallback,
                                                    (XtPointer) newcbs);
                help_widget = NULL;
            }
            else
                help_widget = XtParent (help_widget);
        }

        XFreeCursor (display, cursor);
    }
```

When the user selects the menu item for context-sensitive help, `query_for_help()` is invoked. This routine calls `XmTrackingEvent()` to allow the user to specify a widget on which to see help information. The `confine_to` parameter is set to `True`, so the pointer is constrained to the window of the `toplevel` widget. We use `toplevel` so that the user can select any component in the entire application.

`XmTrackingEvent()` changes the pointer to the specified cursor to provide visual feedback that the application is in a new state. Since the user is expected to click on a object, the routine uses the `XC_hand2` glyph that shows a pointing hand. The `cursor` is created using `XCreateFontCursor()`. See Volume 1, *Xlib Programming Manual*, for more information.

If the user clicks on any valid widget within the application, `XmTrackingEvent()` returns the ID for that widget. The widget itself is not activated and it does not receive any events that indicate that anything has happened at all. If the user does not click on a valid widget, the function returns `NULL`. If `XmTrackingEvent()` returns a widget ID, we use `XtCallCallbacks()` to activate the `XmNhelpCallback` for the widget. If the widget does not have a help callback, `query_for_help()` climbs the widget tree looking for an ancestor widget with a help callback.

While the `confine_to` flag makes `XmTrackingEvent()` useful for constraining mouse movement, you should use this feature with caution. Once the cursor is confined to the window, the server grab is not released until the user presses the mouse button. We also advise caution if you are using a debugger while working with this function. If the debugger stops at a breakpoint while the function is invoked, you will have to log in remotely and kill the debugger process to release the pointer grab. If you kill the process, you will have to shut down the computer.

Working Dialogs

The Motif WorkingDialog is used to inform the user that an application is busy processing, so that it doesn't have the time to handle other actions the user may take. For example, if your application is busy trying to figure out the complete value of *pi*, the user is probably going to have to wait for the application to respond to her next action. The delay occurs

because the application code has control, rather than Xt. When Xt has control, it processes events and dispatches them to the appropriate widgets in the application. If a widget has a callback installed for an event, Xt returns control to the application. While the application has control, there is no way for the window system to service any requests the user may happen to make.

In the meantime, the application is faced with the dilemma of how it is going to process events that happen in the interim. While your application is busy number-crunching, the user is frantically pounding on the *Stop* button and hoping that the application will figure out that she really didn't want it to figure out the complete value of *pi*, but instead to print out the recipe for *cherry pie*.

What the application needs to do is to find a way to do the necessary work for callback routine and process events at the same time. The solution is conceptually simple: the application should periodically check to see if there are any events in the input queue, and if there are, process and dispatch them. The implementation of this solution, on the other hand, is quite a different story. There are a number of different approaches you can take, depending on the nature of the work you are trying to do. Let's examine four of the options:

- If the task can be broken down into tiny chunks, you can set up *work procedures* that are invoked automatically by Xt when there are no events on the event queue. Since events are very infrequent in terms of processor time, this type of processing goes quite quickly. This technique works best for tasks that are not critical to the application; the tasks can be done in the background and not interfere with the normal event-processing loop. To minimize the effect on system performance, you should be sure to break the task into small components time-wise.
- You can set timer event handlers to go off periodically using `XtAppAddTimeOut()`. As each timer fires, another chunk of work is done before control is returned to Xt. While this method is similar to using work procedures, the time intervals may be more in tune with the type of processing you are doing. Timers are typically used when the work being done is synchronous with the system clock or some other regular interval. However, timers are not associated directly with the system clock, so a task should not rely on their accuracy.
- You can choose to maintain control and use Xlib and Xt functions to process events yourself. In this case, your application checks for events in the queue and processes them. This technique is appropriate for applications that need to perform complex operations, as it is possible to handle sophisticated looping constructs, process recursively, or manage complex state information.
- You can simply choose to ignore events entirely. In this case, it is best to set the cursor to a stopwatch or hour glass shape, and/or post a message that indicates that the user must wait. This solution is sometimes the only one available if the task is dependent on some outside entity. Examples include device driver communication (printer, disk

drives), network communications (NFS), inter-process activity (forks and pipes), or anything that puts the application in a state where it has no control over the object with which it is communicating.

You can mix and match some of these techniques. Say the user wants to send a large PostScript file to a laser printer. When she clicks on the *Print* button, you can post a WorkingDialog that reports that the file is being printed and the user must wait. Additionally, you could provide an option that allows the user to send the file to the printer in the background. In this case, you can send the file to the printer in small chunks using work procedures.

The four methods fall into two basic categories:

- Xt maintains control, processes events as normal, and periodically calls application routines
- The application takes control, performs the necessary tasks, and periodically calls Xlib functions to check the event queue

Work procedures and timers return control to Xt and allow it to process events as normal. In turn, Xt gives control back to the application for short intervals every now and then. When the application maintains control, it can query and process X events whenever it wants. While this process is more complicated, it does make it easier for the application to control its own processing.

In all four situations, you can decide whether or not to display a WorkingDialog. If you want to give the user the ability to terminate the work in progress, you can provide a *Stop* button in the dialog. Otherwise, you can simply display the dialog for informational purposes. If you do not want the user to interact with other windows in the application while the WorkingDialog is being displayed, you can make the dialog modal as described in Chapter 5, *Introduction to Dialogs*.

Using Work Procedures

Work procedures in Xt are extremely simple in design. They are typically used by applications that can process tasks in the background. When a work procedure is used in conjunction with a WorkingDialog, the application can provide feedback on the status of the task.Say the user wants to load a large bitmap into a window. The nature of your application requires you to load the file from disk into client-side memory, perform some bitmap manipulation, and then send the bitmap to the X server to be loaded into a pixmap. If you suspect that this task might take a long time and you want to allow the user to interrupt it, you can use work procedures and a WorkingDialog.

Unfortunately, demonstrating such a task is difficult, due to its extremely complex nature. The bitmap loading operation requires a great deal of image-handling code that is a distraction from the issue at hand, which is installing a work procedure. To get around this

problem, we present a short, abstract program that demonstrates the use of a work procedure. In Example 27-4, we represent a time-consuming task by counting from 0 to 20000.*

Example 27-4 The working.c program

```
/* working.c -- represent a complicated, time-consuming task by
** counting from 0 to 20000 and provide feedback to the user about
** how far we are in the process. The user may terminate the process
** at any time by selecting the Stop button in the WorkingDialog.
** This demonstrates how a WorkingDialog can be used to allow the
** user to interrupt lengthy procedures.
*/

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

#define MAXNUM 20000

void done(Widget, XtPointer, XtPointer);
/* Global variables */
static int i = 0;
static XtWorkProcId work_id;

main (int argc, char *argv[])
{
    XtAppContext   app;
    Widget         toplevel, button;
    XmString       label;
    Arg            args[2];
    int            n;
    void           pushed(Widget, XtPointer, XtPointer);

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

    label = XmStringCreateLocalized ("Press To Start A Long Task");
    n = 0;
    XtSetArg (args[n], XmNlabelString, label); n++;
    button = XmCreatePushButton (toplevel, "button", args, n);
    XtAddCallback (button, XmNactivateCallback, pushed, (XtPointer) app);
    XmStringFree (label);
    XtManageChild (button);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

/* pushed() -- the callback routine for the main app's pushbutton.*/
void pushed (Widget w, XtPointer client_data, XtPointer call_data)
```

* `XtVaAppInitialize()` is considered deprecated in X11R6. `XmMessageBoxGetChild()` is deprecated from Motif 2.0.

```
{
    XtAppContext     app = (XtAppContext) client_data;
    Widget           dialog;
    XmString         stop_txt;
    Arg              args[5];
    int              n;
    Boolean          count(XtPointer);

    /* Create the dialog -- the "cancel" button says "Stop" */
    n = 0;
    stop_txt = XmStringCreateLocalized ("Stop");
    XtSetArg (args[n], XmNcancelLabelString, stop_txt); n++;
    dialog = XmCreateWorkingDialog (w, "working", args, n);
    XmStringFree (stop_txt);

    work_id = XtAppAddWorkProc (app, count, dialog);
    XtUnmanageChild (XtNameToWidget (dialog, “OK”));
    XtUnmanageChild (XtNameToWidget (dialog, “Help”));

    /* Use cancel button to stop counting. True = remove work proc */
    XtAddCallback (dialog, XmNcancelCallback, done, (XtPointer) True);
    XtManageChild (dialog);
}

/* count() -- work procedure that counts to MAXNUM. When we get there,
** change the "Stop" button to say "Done".
*/
Boolean count (XtPointer client_data)
{
    Widget       dialog = (Widget) client_data;
    char         buf[64];
    XmString     str, button;
    Boolean      finished = False;
    /* If we printed every number, the flicker is too fast to read.
    ** Therefore, just print every 1000 ticks for smoother feedback.
    */
    if (++i % 1000 != 0)
        return finished;
    /* display where we are in the counter. */
    sprintf (buf, "Counter: %d", i);
    str = XmStringCreateLocalized (buf);
    XtVaSetValues (dialog, XmNmessageString, str, NULL);
    XmStringFree (str);
    if (i == MAXNUM) {
        i = 0;
        finished = True;
        button = XmStringCreateLocalized ("Done");
        XtVaSetValues (dialog, XmNcancelLabelString, button, NULL);
        XmStringFree (button);
        XtRemoveCallback (dialog, XmNcancelCallback, done, (XtPointer) True);
        XtAddCallback (dialog, XmNcancelCallback, done, (XtPointer) False);
        XMapRaised (XtDisplay (dialog), XtWindow (XtParent (dialog)));
    }
    /* Return either True, meaning we're done and remove the work proc,
```

```
    ** or False, meaning continue working by calling this function.
    */
    return finished;
}

/* done () -- user pressed "Stop" or "Done" in WorkingDialog. */
void done (Widget dialog, XtPointer client_data, XtPointer call_data)
{
    Boolean remove_work_proc = (Boolean) client_data;
    if (remove_work_proc) {
        i = 0;
        XtRemoveWorkProc (work_id);
    }
    XtDestroyWidget (XtParent (dialog));
}
```

The main application simply displays a button. When the user presses the button, the application starts counting and displays a WorkingDialog. The user can press *Stop* at any time during the process. If the user allows the application to finish counting, the button is changed from *Stop* to *Done*. Figure 27-3 shows both states of the WorkingDialog.

Figure 27-3 Output of the working program

This program is designed to demonstrate how a work procedure and a WorkingDialog can interact. The callback for the button in the application creates a WorkingDialog using `XmCreateWorkingDialog()`. The callback routine also installs a work procedure using `XtAppAddWorkProc()`. This function takes the following form:

```
XtWorkProcId XtAppAddWorkProc ( XtAppContext app_context,
                                XtWorkProc   proc,
                                XtPointer    client_data)
```

The WorkingDialog is used as the client data for the `count()` work procedure, so that the procedure can update the dialog. To allow the user to interrupt the counting operation, we install done`()` as the `XmNcancelCallback` resource. If the user presses the *Stop* button, this routine is invoked. The routine stops the counting operation by removing the work procedure using `XtRemoveWorkProc()`.

During the counting operation, Xt calls the work procedure when there are no events that need to be processed. The work procedure increments the global counter variable, `i`. Each time `i` reaches an increment of `1000`, the `XmNmessageString` for the WorkingDialog is

updated to inform the user about the progress of the operation. The work procedure returns `True` when the task is complete, which causes Xt to remove the procedure from the list of work procedures being called. When `count()` returns `False`, Xt continues to call the routine when the application is idle.

If the user allows the task to complete, the work procedure changes the action button to say *Done* and removes the `XmNcancelCallback`. The procedure then reinstalls the callback in order to change the client data from `True` to `False`. The client data must be set to `False` so that `done()` does not try to remove the work procedure. Since the work procedure returns `True` in this case, Xt removes the procedure for us.

The work procedure also calls `XMapRaised()` to ensure that the dialog is visible when the operation completes. The user must explicitly press the *Done* button to remove the dialog. Another approach is to call `XtDestroyWidget()` to remove the dialog when the processing is done. In this case, the user is not notified that the operation has finished, but she also does not have to respond to the dialog.

An application can install multiple work procedures, but Xt only processes one procedure at a time. The last work procedure installed has the highest priority, so it is the first one called, except if one work procedure installs another work procedure. In this case, the new procedure has a lower priority than the current one.

As you can see from running the program in Example 27-4, work procedures are called extremely frequently. In any real application, however, the task that is being performed is going to be more sophisticated and time-consuming than our example here. It is important that the operations you perform in a work procedure do not take too much time, or response time will suffer. A work procedure should return frequently enough to allow Xt to process user events, so that the operation of the entire application flows smoothly.

Using Timers

Using timers to process a task is very similar to using work procedures. Timers are not called as frequently as work procedures, so Xt can wait longer for user events to be generated and processed when the application uses timers. An application can add a timer using `XtAppAddTimeOut()`, which takes the following form:

```
XtIntervalId XtAppAddTimeOut ( XtAppContext          app_context,
                               unsigned long         interval,
                               XtTimerCallbackProc   proc,
                               XtPointer             client_data)
```

The *`interval`* parameter specifies how long Xt waits before invoking the timer specified by *`proc`*. The main difference between using a timer and a work procedure is that a timer is called once and then automatically unregistered. To have a timer called at a regular interval, an application must call `XtAppAddTimeOut()` again from within the timer callback. With this exception, using timers is similar to using work procedures, so we aren't

going to present a separate example here. See Chapter 12, *Labels and Buttons*, for some examples that use timers in various contexts.

Processing Events

If your application needs to start a lengthy process that is difficult to break into small pieces, you probably don't want to return control to Xt. In this case, you never lose control of your own processing loop, but you need to check for X events that need to be processed every once in a while. This technique is more convenient than work procedures for certain algorithms, since the application doesn't have to break out of its processing loop unless the user terminates the operation or the task completes naturally.

Processing events is somewhat complicated, but not because of the function calls involved or the design required to support the processing. The complications involve the decisions about which events you want to process, which you want to ignore, and which you want to put off handling until later. Say you are rendering a complicated graphic directly into a DrawingArea. While you are busy processing, you need to decide what to do if you get an incoming `ButtonPress`, `Expose`, or `ConfigureNotify` event, among others.In many cases, what you do depends on the widget or the window that receives the event.

When an application starts a lengthy task, it should post a WorkingDialog that displays an appropriate message. The WorkingDialog can also provide a *Stop* button to allow the user to terminate the task. During the operation, the user should not be interacting with other windows in the application. It is a good idea to change the cursor that is used in these windows, to make it clear that the windows will not respond to user input. When the operation is finished, the application needs to remove the WorkingDialog and reset the cursor.

If you are going to process events yourself, you probably want to write a routine that checks the event queue for relevant events. This routine would process all of the important events, such as those that cause widgets to be repainted. The routine should also handle events for the *Stop* button in the WorkingDialog, so the user can terminate the task.

The program listed in Example 27-5 supports the requirements that we have laid out for an application that processes its own events.*

Example 27-5 The busy.c program

```
/* busy.c -- demonstrate how to use a WorkingDialog and to process
** only important events. e.g., those that may interrupt the
** task or to repaint widgets for exposure. Set up a simple shell
** and a widget that, when pressed, immediately goes into its own
** loop. Set a timeout cursor on the shell and pop up a WorkingDialog
```

* `XtVaAppInitialize()` is considered deprecated in X11R6. `XmMessageBoxGetChild`() is deprecated from Motif 2.0.

```
** Then enter loop and sleep for one second ten times, checking between
** each interval to see if the user clicked the Stop button or if
** any widgets need to be refreshed. Ignore all other events.
**
** main() and get_busy() are stubs that would be replaced by a real
** application; all other functions can be used as is.
*/

#include <Xm/MessageB.h>
#include <Xm/PushB.h>
#include <X11/cursorfont.h>

Widget     shell;
void       TimeoutCursors(Boolean, Boolean);
Boolean    CheckForInterrupt(void);

main (int argc, char *argv[])
{
    XtAppContext   app;
    Widget         button;
    XmString       label;
    Arg            args[2];
    void           get_busy(Widget, XtPointer, XtPointer);

    XtSetLanguageProc (NULL, NULL, NULL);
    shell = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv, NULL,
                                 sessionShellWidgetClass, NULL);
    label = XmStringCreateLocalized ("Press To Start A Long Task");
    XtSetArg (args[0], XmNlabelString, label);
    button = XmCreatePushButton (shell, "button", args, 1);
    XmStringFree (label);
    XtAddCallback (button, XmNactivateCallback, get_busy, NULL);
    XtManageChild (button);
    XtRealizeWidget (shell);
    XtAppMainLoop (app);
}

void get_busy (Widget widget, XtPointer client_data, XtPointer call_data)
{
    int n;

    TimeoutCursors (True, True);

    for (n = 0; n < 10; n++) {
        sleep ((unsigned) 1);
        if (CheckForInterrupt ()) {
            puts ("Interrupt!");
            break;
        }
    }
    if (n == 10)
        puts ("Done");
    TimeoutCursors (False, False);
}
```

```
/* The interesting part of the program -- extract and use at will */
static Boolean stopped; /* True when user wants to stop task */
static Widget dialog; /* WorkingDialog displayed */

/* TimeoutCursors() -- turns on the watch cursor over the application
** to provide feedback for the user that she's going to be waiting
** a while before she can interact with the application again.
*/
void TimeoutCursors (Boolean on, Boolean interruptible)
{
    static int           locked = False;
    static Cursor        cursor = (Cursor) 0;
    extern Widget        shell;
    XSetWindowAttributes attrs;
    Display              *dpy = XtDisplay (shell);
    XEvent               event;
    Arg                  args[5];
    int                  n;
    XmString             str;
    void                 stop(Widget, XtPointer, XtPointer);
    /* "locked" keeps track if we've already called the function.
    ** This allows recursion and is necessary for most situations.
    */
    if (on)
        locked++;
    else
        locked--;
    if (locked > 1 || (locked == 1 && on == False))
        return;
    /* already locked and we're not unlocking */
    stopped = False;
    if (!cursor)
        cursor = XCreateFontCursor (dpy, XC_watch);
    /* if on is true, then turn on watch cursor, otherwise, return
    ** the shell's cursor to normal.
    */
    attrs.cursor = on ? cursor : None;
    /* change the main application shell's cursor to be the timeout
    ** cursor or to reset it to normal. If other shells exist in
    ** this application, they will have to be listed here in order
    ** for them to have timeout cursors too.
    */
    XChangeWindowAttributes (dpy, XtWindow (shell), CWCursor, &attrs);
    XFlush (dpy);
    if (on) {
        /* we're timing out, put up a WorkingDialog. If the process
        ** is interruptible, allow a "Stop" button. Otherwise, remove
        ** all actions so the user can't stop the processing.
        */
        n = 0;
        str = XmStringCreateLocalized ("Busy -- Please Wait.");
        XtSetArg (args[n], XmNmessageString, str); n++;
        dialog = XmCreateWorkingDialog (shell, "busy", args, n);
```

```
            XmStringFree (str);
            XtUnmanageChild (XtNameToWidget (dialog, "OK"));
            XtUnmanageChild (XtNameToWidget (dialog, "Help"));
            if (interruptible) {
                str = XmStringCreateLocalized ("Stop");
                XtVaSetValues (dialog, XmNcancelLabelString, str, NULL);
                XmStringFree (str);
                XtAddCallback (dialog, XmNcancelCallback, stop, NULL);
            }
            else
                XtUnmanageChild (XtNameToWidget (dialog, "Cancel"));
            XtManageChild (dialog);
        }
        else {
            /* get rid of all button and keyboard events that occurred
            ** during the time out. The user shouldn't have done anything
            ** during this time, so flush for button and keypress events.
            ** KeyRelease events are not discarded because accelerators
            ** require the corresponding release event before normal input
            ** can continue.
            */
            while (XCheckMaskEvent (dpy, ButtonPressMask | ButtonReleaseMask |
                ButtonMotionMask | PointerMotionMask | KeyPressMask, &event)) {
                /* do nothing */;
            }
            XtDestroyWidget (dialog);
        }
    }

    /* stop() -- user pressed the "Stop" button in dialog. */
    void stop (Widget dialog, XtPointer client_data, XtPointer call_data)
    {
        stopped = True;
    }

    /* CheckForInterrupt() -- check events in event queue and process
    ** the interesting ones.
    */
    Boolean CheckForInterrupt (void)
    {
        extern Widget  shell;
        Display        *dpy = XtDisplay (shell);
        Window         win = XtWindow (dialog);
        XEvent         event;
        /* Make sure all our requests get to the server */
        XFlush (dpy);
        /* Let Motif process all pending exposure events for us. */
        XmUpdateDisplay (shell);
        /* Check the event loop for events in the dialog ("Stop"?) */
        while (XCheckMaskEvent (dpy, ButtonPressMask | ButtonReleaseMask |
                ButtonMotionMask | PointerMotionMask | KeyPressMask, &event)) {
            /* got an "interesting" event. */
            if (event.xany.window == win)
                XtDispatchEvent (&event); /* it's in our dialog. */
```

```
            else
                /* uninteresting event--throw it away and sound bell */
                XBell (dpy, 50);
        }
        return stopped;
    }
```

This program is obviously for demonstration purposes only. To keep to the subject matter, we have made the main part of the program quite unrealistic and only use it to support the functions we are about to discuss. The application displays a single button that starts a "long task". The `get_busy()` callback routine is trivial, but it demonstrates the use of the `TimeoutCursors()` and `CheckForInterrupt()` routines.

The `TimeoutCursors()` routine is used to change the cursor for the main application shell and to post the WorkingDialog. The cursor is changed to a watch shape to give the user visual feedback that the main window is not responding to input. The routine uses the static variable `locked` to keep track of how many times it has been called with `on` set to `True`. The function does not reset the cursor and remove the WorkingDialog until a matching number of calls has been made with `on`set to `False`. This technique makes it possible for a low-level function in an application to call `TimeoutCursors()` at its beginning and end, without affecting higher-level loops that also call the function.

The routine stores the `XC_watch` cursor in the static `cursor` variable. The cursor is created using `XCreateFontCursor()`, which is why *<X11/cursorfont.h>* is included. `TimeoutCursors()` uses `XChangeWindowAttributes()` to change the cursor to the watch shape and to reset it to its normal shape when `on` is `False`. The cursor is modified for the window of the `shell` widget, which is the main window for the application. If your application uses multiple SessionShells or TopLevelShells, you will need to modify the function to change the cursor shape for all of the shells*.

At this point, we call `XFlush()` to make sure that all of our requests have been sent to the server. The `TimeoutCursors()` function may be called from deep within an application, so there may be a number of server requests that are waiting and we want to be sure that the server knows about them now. If the are turning off the timeout cursor, we may also need to read any resulting events.

Now we determine whether we are locking or unlocking the application. If we are locking it, we create and post a WorkingDialog. The dialog is created with a standard message. If the `interruptible` parameter is `True`, we provide a *Stop* button by changing the label of the *Cancel* button. We also add a callback routine for button, so that we can actually stop the task in progress.

We should note that an application does not necessarily have to post a WorkingDialog, as long as it changes the cursor. The watch cursor provides enough feedback to indicate that

* The ApplicationShell is deprecated in X11R6, where the SessionShell is to be preferred.

the application is in a busy state. The decision about whether or not to post a dialog really depends on the length of the task being performed. For relatively short tasks, it typically doesn't make sense to provide a WorkingDialog, as it takes some time to actually create and post the dialog.

Now the application is in a busy state. However, the user has yet to see anything; events need to be processed in order for the dialog to be mapped to the screen. At this point, the `CheckForInterrupt()` routine takes over. This routine handles `Expose` events by calling `XmUpdateDisplay()`. This Motif function processes all of the *Expose* events in the event queue by causing the server to flush these events for all of the windows on the display. This processing may cause redrawing event handlers to be called for various widgets. If you have installed your own exposure routines for any widgets, be sure that they are not too time consuming, or you may find yourself in a bind. You can check to see which windows are going to be repainted before it actually happens by using `XCheckMaskEvent()` to process `Expose` events.

After any possible repainting has occurred, we check for any button or keyboard events in the event queue. If one has been generated, we extract it from the input queue using `XCheckMaskEvent()`. The function takes the following form:

```
Bool XCheckMaskEvent ( Display *display,
                       long event_mask,
                       XEvent *event_return)
```

This Xlib function looks for events in the queue that match *`event_mask`*. If there is a matching event, the *`event_return`* parameter is filled in with the event and the routine returns `True`. Otherwise, the function returns `False` and we can return. The event is processed only if it occurred within the WorkingDialog window. Since the application is busy, events in other windows are not processed. If the user did something in the WorkingDialog, we process the event because she may have activated the *Stop* button. If the button is not provided for the dialog, it does not affect the code here.

You should be aware that `XCheckMaskEvent()` removes the event from the queue. If you choose not to process an event, you cannot stick it back in the queue. If you retrieve an event out of the queue and don't want to process it, you should set an application-defined variable or flag that notifies the application that it must eventually deal with the event. Another alternative is to save the event by allocating a new `XEvent` structure and copying the data. Then you dispatch the event later, when you are prepared to handle it.

We do not check for `KeyRelease` events in `CheckForInterrupt()` for a very important reason that concerns how the X Toolkit Intrinsics handles accelerators. Say your application has a menu item that initiates a long, complicated process. The callback function for this menu item calls both `TimeoutCursors()` and `CheckForInterrupt()`, just like the `get_busy()` routine. Let's say that ALT-X is the accelerator for the menu item. When the user types this key sequence, the callback routine for the menu item is activated by the `KeyPress` events. At this time, the `KeyRelease`

events associated with the accelerator are still in the event queue. If we checked for KeyRelease events in CheckForInterrupt(), the ones for the accelerator would get thrown away, since they did not occur in the WorkingDialog.

Throwing away these events is a problem because Xt uses an internal state machine to determine whether or not any particular sequence of keyboard events is an accelerator or a prefix for one. Since Xt would never get the accompanying KeyRelease events, it would think that the user is still entering a keyboard accelerator.Xt would not get out of that state until the matching events were given, with the result that no other keyboard events would work in the application until the user happened to type the same accelerator sequence. This situation is not a bug in Xt; Xt is simply doing what it must to handle acclerators. However, the situation does demonstrate the intricacies of handling events in X.

Getting back to CheckForInterrupt(), if the user presses the *Stop* in the dialog, the event is processed and the stop() callback routine is invoked. This routines imply sets the global variable stopped to True. By the time that CheckForInterrupt() is ready to return, stopped has been set, so the function returns True. If the WorkingDialog does not have a *Stop* button, the callback routine is not installed, so stopped is never set to True.

After the get_busy() routine finishes processing, it calls TimeoutCursors() again to unlock the application. When on is set to False, the routine uses XCheckMaskEvent() to look in the event queue for button and keyboard events. In this case, the events are thrown away, since the input is no longer useful. The routine also destroys theWorkingDialog. In one sense, TimeoutCursors() implements a kind of modality, similar to that discussed in Chapter 5, *Introduction to Dialogs*. However, modality alone cannot provide the functionality necessary to handle long-running tasks.

Updating the Display

As discussed earlier, XmUpdateDisplay() checks the event queue for all Expose events and processes them immediately. However, there are some circumstances under which the routine does not work as you might expect. For example, let's say that your application creates and posts a dialog that contains a DrawingArea widget. You call XSync() and XmUpdateDisplay() to make sure that the dialog is on the screen and fully exposed. After you call XClearWindow() to make sure the window is clear, you begin drawing. Unfortunately, you may find that nothing is drawn.

The problem is due to the redirection of events from the window manager and the way events are processed and queued. When a dialog is posted using XtManageChild() or XtPopup(), the toolkit calls XMapRaised() to raise the window to the top of the window stack. The call to XSync() sends the MapRequest event to the server, which redirects it to the window manager(e.g., *mwm*). A bottleneck can occur if the window manager is swapped out, which is a side effect of multi-tasking operating systems such as UNIX.

In this case, *mwm* may not react immediately to the redirection and can take an indeterminate amount of time to respond. The X server doesn't take this delay into account. It thinks that the event has been delivered properly, so your application believes that the window has been mapped. As a result, `XmUpdateDisplay()` doesn't get the `Expose` event that you were expecting and drawing does no good because the window still hasn't been mapped. When *mwm* gets around to mapping the window to the screen, the server generates the `Expose` event, but by now your application is off doing something else.

One solution to this problem is to change the design of your application so that it doesn't start drawing until the server actually generates the `Expose` events. In this case, you should post the dialog and immediately return control to the main event-processing loop (`XtAppMainLoop()`). If you have installed an event handler or a translation for the `Expose` event, the routine is called at the appropriate time. Another advantage to this design is that the drawing procedure is called any time an `Expose` event occurs, which ensures that the window is always up-to-date.

In Example 27-6, we show another solution. This solution should be used only if you need to create, pop up, or manage a dialog and then immediately draw into the window. The `ForceUpdate()` routine ensures that the specified widget is visible before it returns.

Example 27-6 The ForceUpdate() routine.

```
/* ForceUpdate() -- a superset of XmUpdateDisplay() that ensures
** that a window's contents are visible before returning.
** The monitoring of window states is necessary because an attempt to
** map a window is subject to the whim of the window manager, which can
** introduce a significant delay before the window is actually mapped
** and exposed. This function is intended to be called after XtPopup(),
** XtManageChild() or XMapRaised(). Don't use it in other situations
** as it may sit and process other unrelated events until the widget
** becomes visible.
*/

/* The parameter widget must be visible before the function returns */
void ForceUpdate (Widget w)
{
    Widget             diashell, topshell;
    Window             diawindow, topwindow;
    XtAppContext       cxt = XtWidgetToApplicationContext (w);
    Display            *dpy;
    XWindowAttributes  xwa;
    XEvent             event;
    /* Locate the shell we are interested in */
    for (diashell = w; !XtIsShell (diashell);
        diashell = XtParent (diashell));
    /* Locate its primary window's shell (which may be the same) */
    for (topshell = diashell; !XtIsTopLevelShell (topshell);
        topshell = XtParent (topshell));
    /* If the dialog shell (or its primary shell window) is
    ** not realized, don't bother... nothing can possibly happen.
```

```
        */
        if (XtIsRealized (diashell) && XtIsRealized (topshell)) {
            dpy = XtDisplay (topshell);
            diawindow = XtWindow (diashell);
            topwindow = XtWindow (topshell);

            /* Wait for the dialog to be mapped.
            ** It's guaranteed to become so
            */
            while (XGetWindowAttributes (dpy, diawindow, &xwa) &&
                                        xwa.map_state != IsViewable) {
                /*...if the primary is (or becomes) unviewable or
                ** unmapped, it's probably iconic, and nothing will happen.
                */
                if (XGetWindowAttributes (dpy, topwindow, &xwa) &&
                                        xwa.map_state != IsViewable)
                    break;
                /* we are guaranteed there will be an event of some kind. */
                XtAppNextEvent (cxt, &event);
                XtDispatchEvent (&event);
            }
        }
        /* The next XSync() will get an expose event. */
        XmUpdateDisplay (topshell);
    }
```

This routine makes sure that a dialog is visible by waiting for the window of the dialog to be mapped to the screen.

Avoiding Forks

Before we close out this section, there is one more method of executing tasks in the background that we should discuss. Beginning programmers tend to use library functions and system calls such as system(), popen(), fork(), and exec() to invoke external commands. Although these functions are perfectly reasonable, they can backfire quite easily on virtually any error condition. Recovering from these errors is the GUI programmer's nightmare, since there are so many different possible conditions to deal with.

The purpose of using these functions, of course, is to call another UNIX program and have it run concurrently with the main application. The system() and popen() functions fork a new process using the fork() system call. They also use some form of exec() so the new child process can invoke the external UNIX program. If the new process cannot fork, if there is something wrong with the external UNIX command, if there is a communications protocol error, or any one of a dozen other possible error conditions, there is noway for the external program to display an error message as a part of the main application.

It is unlikely that the external program would display a dialog box or any sort of reasonable user-interface element. It is illegal fora new process to use any of the widgets or windows in the main application because only one connection to the server is allowed per process. If

the child process wants to post a dialog, it must establish a new connection to the X server and create an entirely new widget tree, as it is a separate application. Since most system utilities do not have graphical user interface front ends, this scenario is very unlikely. It is also entirely unreasonable to have any expectations of the external process, especially since other solutions are much easier.

If a separate process is necessary in order to accomplish a particular task, setting up pipes between the child application and the parent is usually the best alternative. The `popen()` function uses this method superficially, but it is not the most elegant solution. The routine only handles forking the new process and setting up half of a two-way pipe. The `popen()` function is used in several places throughout the book; check the index for those uses.

To really handle external processes and pipes properly, an application should do the following:

1. The parent process calls `pipe()` to set up entry points for the expected child process' input and output channels. Two pipes for both input and output are usually needed.
2. The parent process calls `fork()` to spawn the new child process.
3. The child uses `dup2()` to redirect its own `stdin`, `stdout`, and `stderr` to the other ends of the pipes set up by the parent. The communication pipeline between the parent and the child is now established.
4. The parent calls `XtAppAddInput()` to tell Xt to monitor an additional file descriptor while it is waiting for input events from the X server.
5. The parent can read data (e.g., output, error conditions, etc.) sent by the child using `read()` on the appropriate pipe.
6. The parent can display the output from the pipe to a dialog, a ScrolledText object, or some other widget because it is still in connection with the X server.

If the parent calls `XtAppAddInput()`, Xt can see the data the child sends through the pipe and invoke the callback routine associated with the file descriptor. `XtAppAddInput()` takes the following form:

```
XtInputId XtAppAddInput ( XtAppContext          app_context,
                          int                   source,
                          XtPointer             mask,
                          XtInputCallbackProc   proc,
                          XtPointer             client_data)
```

The *source* parameter should be the side of the pipe that the parent uses to read data sent by the child process. The *proc* function is called when there is data to read on the pipe. When the function is called, the *client_data* is passed to the callback. For example, you can pass the process ID returned by `fork()`, so you can see if the process is still alive and read the data using `read()`.

This discussion is merely presented as an overview, since the implementation details are beyond the scope of this book. For example, UNIX signals cause problems in a number of ways. The parent process is sent signals when the child dies or its process state changes. The child is also sent signals that are delivered to the parent by the user or other outsideforces. Different forms of UNIX require that process groups be set up in different ways to avoid other problems with signals.

Another problem involves file descriptors that are set up as non-blocking files. If `read()` returns `0` with one of these descriptors, you may not know whether there is nothing to read or the end of the file has been reached, which means that the child process has terminated. Incidentally, `popen()` does not deal with any of these issues correctly, so building a new solution is the best thing to do in the long run.

You should really consult the programmer's guide for your UNIX system for more information on the techniques used to spawn new processes and communicate with them appropriately. Once you have a handle on those issues, it should be relatively easy to redirect text from file descriptors using the toolkit. For more information on `XtAppAddInput()`, including examples of how it can be used, see Volume 4, *X Toolkit Intrinsics Programming Manua*l.

Dynamic Message Symbols

The MessageDialog is used to display many different types of messages; the image in the dialog helps the user identify the purpose of the dialog. The pixmaps used by the standard MessageDialogs are predefined by the Motif toolkit. When you are using the standard dialogs, you typically change the dialog's type rather than its symbol, since changing its type effectively changes the symbol that it displays. However, you can change the MessageDialog's symbol to a customized image using the `XmNsymbolPixmap` resource.

The resource takes a pixmap value that must be created before the resource is set. When the resource is set, the pixmap is not copied by the dialog widget. If the dialog is destroyed, you should be sure to free the pixmap unless you are using it elsewhere. If you are going to destroy the dialog using `XtDestroyWidget()` directly, you should get the pixmap by calling `XtVaGetValues()`, so that you can free it. However, the dialog can also be destroyed automatically, so you should also specify an `XmNdestroyCallback` procedure that is called whenever the dialog is destroyed.

Example 27-7 shows an example of using a custom image in a standard MessageDialog. The program also demonstrates how the dialog should clean up after itself.*

* `XtVaAppInitialize()` is considered deprecated in X11R6. `XmStringCreateLtoR()` and `XmMessageBoxGetChild()` are deprecated from Motif 2.0.

Example 27-7 The warn_msg.c program

```
/* warn_msg.c -- display a very urgent warning message.
** Really catch the user's attention by flashing an urgent-
** looking pixmap every 250 milliseconds.
** The program demonstrates how to set the XmNsymbolPixmap
** resource, how to destroy the pixmap and how to use timers.
*/

#include <Xm/MessageB.h>
#include <Xm/PushB.h>
#include "bang0.symbol"
#include "bang1.symbol"

#define TEXT "Alert!\n\
The computer room is ON FIRE!\n\
All of your e-mail will be lost."

/* define the data structure we need to implement flashing effect */
typedef struct {
    XtIntervalId   id;
    int            which;
    Pixmap         pix1, pix2;
    Widget         dialog;
    XtAppContext   app;
} TimeOutClientData;

main (int argc, char *argv[])
{
    XtAppContext  app;
    Widget        toplevel, button;
    XmString      label;
    Arg           args[2];
    int           n;
    void          warning(Widget, XtPointer, XtPointer);

    XtSetLanguageProc (NULL, NULL, NULL);
    toplevel = XtVaOpenApplication (&app, "Demos", NULL, 0, &argc, argv, NULL,
                                    sessionShellWidgetClass, NULL);
    label = XmStringCreateLocalized ("Do Not Touch");
    n = 0;
    XtSetArg (args[n], XmNlabelString, label); n++;
    button = XmCreatePushButton (toplevel, "button", args, n);
    XtManageChild (button);
    XmStringFree (label);
    /* set up callback to popup warning */
    XtAddCallback (button, XmNactivateCallback, warning, NULL);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

/* warning() -- callback routine for the button. Create a message
** dialog and set the message string. Allocate an instance of
** the TimeOutClientData structure and set a timer to alternate
```

```
** between the two pixmaps. The data is passed to the timeout
** routine and the callback for when the user presses "OK".
*/
void warning (Widget parent, XtPointer client_data, XtPointer call_data)
{
    Widget             dialog;
    XtAppContext       app = XtWidgetToApplicationContext (parent);
    XmString           text;
    void               done(Widget, XtPointer, XtPointer);
    void               destroy_it(Widget, XtPointer, XtPointer);
    void               blink(XtPointer, XtIntervalId *);
    Display            *dpy = XtDisplay (parent);
    Screen             *screen = XtScreen (parent);
    Pixel              fg, bg;
    Arg                args[5];
    int                n, depth;
    TimeOutClientData  *data = XtNew (TimeOutClientData);

    /* Create the dialog */
    n = 0;
    XtSetArg (args[n], XmNdeleteResponse, XmDESTROY); n++;
    dialog = XmCreateMessageDialog (parent, "danger", args, n);
    XtUnmanageChild (XtNameToWidget (dialog, “Cancel”));
    XtUnmanageChild (XtNameToWidget (dialog, ”Help”));
    XtAddCallback (dialog, XmNokCallback, done, NULL);
    XtAddCallback (dialog, XmNdestroyCallback, destroy_it, data);
    /* now that dialog has been created, it's colors are initialized */
    XtVaGetValues (dialog, XmNforeground, &fg, XmNbackground, &bg, XmNdepth,
                           &depth, NULL);
    /* Create pixmaps that are going to be used as symbolPixmaps.
    ** Use the foreground and background colors of the dialog.
    */
    data->pix1 = XCreatePixmapFromBitmapData (dpy, XtWindow (parent),
                    bang0_bits, bang0_width, bang0_height, fg, bg, depth);
    data->pix2 = XCreatePixmapFromBitmapData (dpy, XtWindow (parent),
                    bang1_bits, bang1_width, bang1_height, fg, bg, depth);
    /* complete the timeout client data */
    data->dialog = dialog;
    data->app = app;
    /* Add the timeout for blinking effect */
    data->id = XtAppAddTimeOut (app, 1000L, blink, (XtPointer) data);
    /* display the help text and the appropriate pixmap */
    text = XmStringGenerate (TEXT, XmFONTLIST_DEFAULT_TAG,
    XmCHARSET_TEXT, NULL);
    XtVaSetValues (dialog, XmNmessageString, text, XmNsymbolPixmap,
                        data->pix2, NULL);
    XmStringFree (text);
    XtManageChild (dialog);
}

/* blink() -- visual blinking effect for dialog's symbol. Displays
** flashing ! symbol, restarts timer and saves timer id.
*/
void blink (XtPointer client_data, XtIntervalId *id)
```

```
{
    TimeOutClientData *data = (TimeOutClientData *) client_data;
    Pixmap            pixmap;

    data->id = XtAppAddTimeOut (data->app, 250L, blink, (XtPointer) data);
    data->which = !data->which;
    pixmap = (data->which ? data->pix1 : data->pix2);
    XtVaSetValues (data->dialog, XmNsymbolPixmap, pixmap, NULL);
}

/* done() -- called when user presses "OK" in dialog or
** if the user picked the Close button in system menu.
** Remove the timeout id stored in data, free pixmaps and
** make sure the widget is destroyed (which is only when
** the user presses the "OK" button.
*/
void done (Widget dialog, XtPointer client_data, XtPointer call_data)
{
    XtDestroyWidget (XtParent (dialog));
}

/* destroy_it() -- called when dialog is destroyed. Removes
** timer and frees allocated data.
*/
void destroy_it (Widget dialog, XtPointer client_data, XtPointer call_data)
{
    TimeOutClientData *data = (TimeOutClientData *) client_data;
    Pixmap symbol;
    XtRemoveTimeOut (data->id);
    XFreePixmap (XtDisplay (data->dialog), data->pix1);
    XFreePixmap (XtDisplay (data->dialog), data->pix2);
    XtFree ((char *) data);
}
```

The dialog is created in `warning()`, the callback routine for the PushButton in the main window. We create a simple MessageDialog that does not have a predefined symbol so we can specify a custom image. The dialog actually uses two symbols that are exchanged every 250 milliseconds by a timer callback routine. The output of this program is shown in Figure 27-4.

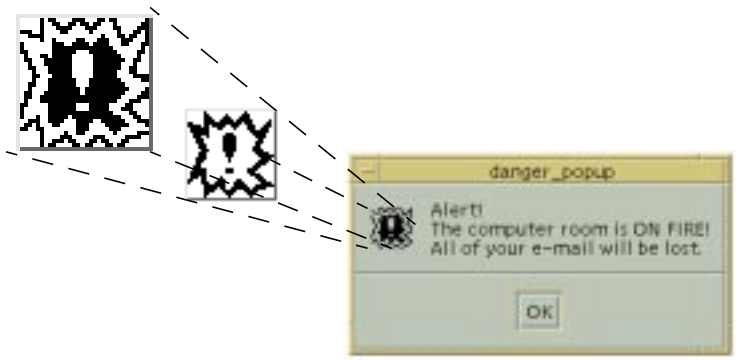


Figure 27-4 The output of the warn_msg program

To implement the flashing symbol, we must associate certain information with the dialog. Basically, we need to keep track of the two pixmaps and the timer routine. All of the information is placed in a single data structure, so we can pass the structure around as client data. We can also use multiple structure variables to store information about multiple dialogs. The `TimeOutClientData` is defined as follows:

```
typedef struct {
    XtIntervalId   id;
    int            which;
    Pixmap         pix1, pix2;
    Widget         dialog;
    XtAppContext   app;
} TimeOutClientData;
```

The `warning()` routine allocates a new instance of the structure using `XtNew()`, since it is going to create anew dialog and it needs a unique structure for the dialog. The routine uses `XmCreateMessageDialog()` to create the dialog. We unmanage the *Cancel* and *Help* buttons and specify a callback for the *OK* button. The `done()` callback simply calls `XtDestroyWidget()`, which causes the `XmNdestroyCallback` to be called. We also set the `XmNdeleteResponse` resource for the dialog to `XmDESTROY`. This setting causes the Motif toolkit to destroy the dialog if the user dismisses it using the *Close* button on the window menu,

Since we are not reusing the dialog or its data, we must be sure to free the pixmaps, release the timer, and free the allocated data structure when the dialog is destroyed. To be sure that these tasks take place, we install a callback function for the `XmNdestroyCallback` resource. The `destroy_it()` routine handles all of the cleanup for the dialog.

Before we create the pixmaps that are used in the dialog, we retrieve the dialog's foreground and background colors using `XtVaGetValues()` so that the new pixmaps can use the same colors. Once the colors are known, we can create the pixmaps and finish initializing the fields in the `TimeOutClientData` structure. The `dialog` field of the

structure points to the MessageDialog. We call `XtAppAddTimeOut()` to start the timer that controls the flashing effect and set the `id` field to the timer ID.

We perform a final bit of setup for the dialog by specifying the `XmNsymbolPixmap` and `XmNmessageString` resources. Once everything is set up, the function returns, Xt regains control, and normal event processing resumes. After the initial one-second interval times out, the blink`()` function is called. This routine adds another timeout for 250 milliseconds and switches the pixmaps displayed in the dialog. This loop continues until the user dismisses the dialog, at which time it is destroyed, the pixmaps are freed, the timer is removed, and the `TimeOutClientData` structure is freed.

Since we created a simple MessageDialog that does not have a predefined image, we did not have to get a handle to the `XmNsymbolPixmap` for the dialog and destroy it. However, if you decide to change the pixmap for one of the standard dialogs that has a predefined symbol, like the ErrorDialog, you should get its pixmap and free it. In this case, you should use `XmDestroyPixmap()` rather than `XFreePixmap()`. The Motif dialogs use `XmGetPixmap()` to create their images, so the pixmaps must be freed with the companion routine `XmDestroyPixmap()`. See Chapter 3, *Overview of the Motif Toolkit*, for a discussion on `XmGetPixmap()`.

Although changing the symbol pixmap in a dialog is quite simple, using the feature effectively requires a careful design to make sure that all of the pointers and data structures are destroyed appropriately. Being meticulous about cleaning up after destroyed widgets and other objects is sometimes a difficult task because of the many ways in which the user can destroy them. However, eliminating these possible memory leaks enables a program to run longer and more efficiently.

Summary

Developing a real application often involves a lot of work to get the details just right. Some of the most interesting problems in designing an interface cannot be solved by Motif alone. Motif provides the basic user interface, but you must make it work with your application. A solid understanding of the fundamentals of the X Window System and the X Toolkit Intrinsics makes it easier to fine-tune the interface for an application. This chapter has presented some solutions to common problems that require using both Xlib and Xt routines in conjunction with the Motif toolkit.

A

Additional Example Programs

This appendix provides some additional example programs that illustrate techniques not discussed in the body of the book.

This appendix contains a number of programs that provide more realistic examples of how the Motif toolkit is used. Most of the examples are also intended to encourage further investigation into other X-related topics, such as the use of app-defaults files, fallback resources, and command-line option parsing. Our discussion of the examples is fairly limited; see the comments in the code for explanations of various implementation details.

A Bitmap Display Utility

The *xshowbitmap* program is a useful utility for reviewing a group of bitmap files. The filenames for the bitmaps can be specified on the command line, sent through a pipe, or typed into `stdin`. All of the bitmaps are drawn into a pixmap, which is rendered into a DrawingArea widget. The DrawingArea is used as the work window for a ScrolledWindow, so that we can demonstrate application-defined scrolling for the Motif ScrolledWindow. The bitmaps are displayed in an equal number of rows and columns if possible. Alternatively, you can specify either the number of rows or the number of columns using the `-rows` or `-columns` command-line option, respectively.

The example in Example A-1 demonstrates the use of Xt mechanisms for adding command-line options and application-level resources in an application. For an explanation of these Xt features, see Volume 4. For details on the Xlib functions for reading and manipulating bitmaps, see Volume 1.*

Example A-1 The xshowbitmap.c program

```
/* Written by Dan Heller and Paula Ferguson.
** Copyright 1994, O'Reilly & Associates, Inc.
```

* `XtVaAppInitialize`() is considered deprecated in X11R6. The SessionShell and `XtVaOpenApplication`() are only available in X11R6.

```
**
** The X Consortium, and any party obtaining a copy of these files from
** the X Consortium, directly or indirectly, is granted, free of charge, a
** full and unrestricted irrevocable, world-wide, paid up, royalty-free,
** nonexclusive right and license to deal in this software and
** documentation files (the "Software"), including without limitation the
** rights to use, copy, modify, merge, publish, distribute, sublicense,
** and/or sell copies of the Software, and to permit persons who receive
** copies from any such party to do so.  This license includes without
** limitation a license to do the foregoing actions under any patents of
** the party supplying this software to the X Consortium.
*/

/* xshowbitmap.c -- displays a set of bitmaps specified on the command
** line, from a pipe, or typed into stdin.  Bitmaps must be specified
** as file names.
**
** Usage: xshowbitmap
** -s             sorts the bitmaps in order of size with largest first
** -v             verbose mode for when input is redirected to stdin
** -w             width of viewport window
** -h             height of viewport window
** -fg            foreground color
** -bg            background color
** -label         labels each bitmap with its corresponding filename
**                      (this is the default behavior)
** -nolabel       doesn't label each bitmap with its filename
** -grid N        line width for grid between bitmaps; defaults to 1
** -rows N        number of rows; cannot be used with -cols
** -cols N        number of columns; cannot be used with -rows
** -fn font       font for bitmap filenames
** -bw max-width  excludes bitmaps larger than this width; default is 64
** -bh max-height excludes bitmaps larger than this height; default is 64
** -              indicates to read from stdin; piping doesn't require
**                      the '-' argument
** [no arguments] reads from stdin
**
** Modified by A.J.Fountain, IST
** for Motif 2.1/X11R6, ANSI
**
** Example usage:
** xshowbitmaps /usr/X11R6/include/X11/bitmaps/*
*/

#include <stdio.h>
#include <X11/Xos.h>
#include <Xm/ScrolledW.h>
#include <Xm/DrawingA.h>
#include <Xm/ScrollBar.h>

#ifdef max
#undef max
#endif
#define max(a,b) ((int)(a)>(int)(b)?(int)(a):(int)(b))
```

```
#define min(a,b) ((int)(a)<(int)(b)?(int)(a):(int)(b))

typedef struct {
    char             *name;
    int              len;
    unsigned int     width, height;
    Pixmap           bitmap;
} Bitmap;

/* Resrcs is an object that contains global variables that we want the
** user to be able to initialize through resources or command line options.
** XtAppInitialize() initializes the fields in this data structure to values
** indicated by the XrmOptionsDescRec structure defined later.
*/
struct _resrcs {
    Boolean        sort;                    /* sort the bitmaps */
    Boolean        verbose;                 /* loading bitmaps verbosely */
    Boolean        label_bitmap;            /* whether to label bitmaps */
    int            max_width, max_height;   /* largest allowable bitmap */
    unsigned int   grid;                    /* line width between bitmaps */
    Pixel          fg, bg;                  /* colors of bitmaps */
    XFontStruct    *font;                   /* font for bitmap labels */
    Dimension      view_width, view_height; /* initial clip window size */
    int            rows, cols;              /* forcefully set #rows/cols */
} Resrcs;

/* .Xdefaults or app-defaults resources.  The last field in each structure
** is used as the default values for the field in the Resrcs struct above.
*/
static XtResource resources[] = {
    { “sort”, “Sort”, XmRBoolean, sizeof (Boolean),
                XtOffsetOf (struct _resrcs, sort), XmRImmediate, False },
    { “verbose”, “Verbose”, XmRBoolean, sizeof (Boolean),
                XtOffsetOf (struct _resrcs,verbose), XmRImmediate, False },
    { “labelBitmap”, “LabelBitmap”, XmRBoolean, sizeof (Boolean),
                XtOffsetOf (struct _resrcs, label_bitmap), XmRImmediate,
                (char *) True },
    { “grid”, “Grid”, XmRInt, sizeof (int), XtOffsetOf (struct _resrcs, grid),
                XmRImmediate, (char *) 1 },
    { “bitmapWidth”, “BitmapWidth”, XmRInt, sizeof (int),
                XtOffsetOf (struct _resrcs, max_width), XmRImmediate,
                (char *) 64 },
    { “bitmapHeight”, “BitmapHeight”, XmRInt, sizeof (int),
                XtOffsetOf (struct _resrcs, max_height), XmRImmediate,
                (char *) 64 },
    { XmNfont, XmCFont, XmRFontStruct, sizeof (XFontStruct *),
                 XtOffsetOf (struct _resrcs, font), XmRString, XtDefaultFont },
    { XmNforeground, XmCForeground, XmRPixel, sizeof (Pixel),
                XtOffsetOf (struct _resrcs, fg), XmRString,
                XtDefaultForeground },
    { XmNbackground, XmCBackground, XmRPixel, sizeof (Pixel),
                XtOffsetOf (struct _resrcs, bg), XmRString,
                XtDefaultBackground },
```

```
    { "view-width", "View-width", XmRDimension, sizeof (Dimension),
            XtOffsetOf (struct _resrcs, view_width), XmRImmediate,
            (char *) 500 },
    { "view-height", "View-height", XmRDimension, sizeof (Dimension),
            XtOffsetOf (struct _resrcs, view_height), XmRImmediate,
            (char *) 300 },
    { "rows", "Rows", XmRInt, sizeof (int), XtOffsetOf (struct _resrcs, rows),
            XmRImmediate, 0 },
    { "cols", "Cols", XmRInt, sizeof (int), XtOffsetOf (struct _resrcs, cols),
            XmRImmediate, 0 },
};

/* If the following command line args (1st field) are found, set the
** associated resource values (2nd field) to the given value (4th field).
*/
static XrmOptionDescRec options[] = {
    { "-sort", "sort", XrmoptionNoArg, "True" },
    { "-v", "verbose", XrmoptionNoArg, "True" },
    { "-fn", "font", XrmoptionSepArg, NULL },
    { "-fg", "foreground", XrmoptionSepArg, NULL },
    { "-bg", "background", XrmoptionSepArg, NULL },
    { "-w", "view-width", XrmoptionSepArg, NULL },
    { "-h", "view-height", XrmoptionSepArg, NULL },
    { "-rows", "rows", XrmoptionSepArg, NULL },
    { "-cols", "cols", XrmoptionSepArg, NULL },
    { "-bw", "bitmapWidth", XrmoptionSepArg, NULL },
    { "-bh", "bitmapHeight", XrmoptionSepArg, NULL },
    { "-bitmap_width", "bitmapWidth", XrmoptionSepArg, NULL },
    { "-bitmap_height", "bitmapHeight", XrmoptionSepArg, NULL },
    { "-label", "labelBitmap", XrmoptionNoArg, "True" },
    { "-nolabel", "labelBitmap", XrmoptionNoArg, "False" },
    { "-grid", "grid", XrmoptionSepArg, NULL },
};

/* size_cmp() -- used by qsort to sort bitmaps into alphabetical order
** This is used when the "sort" resource is true or when -sort is given.
*/
int size_cmp(Bitmap *b1, Bitmap *b2)
{
    int n = (int) (b1->width * b1->height) - (int) (b2->width * b2->height);
    if (n)
        return n;
    return strcmp (b1->name, b2->name);
}

/* int_sqrt() -- get the integer square root of n.  Used to put the
** bitmaps in an equal number of rows and colums.
*/
int int_sqrt(register int n)
{
    register int i, s = 0, t;
    for (i = 15; i >= 0; i--) {
        t = (s | (1L << i));
        if (t * t <= n)
```

```
            s = t;
    }
    return s;
}

/* global variables that are not changable thru resources or command
** line options.
*/
Widget          drawing_a, vsb, hsb;
Pixmap          pixmap; /* used the as image for Label widget */
GC              gc;
Display         *dpy;
unsigned int    cell_width, cell_height;
unsigned int    pix_hoffset, pix_voffset, sw_hoffset, sw_voffset;

void redraw(Window);

main(int argc, char *argv[])
{
    extern char     *strcpy();
    XtAppContext    app;
    Widget          toplevel, scrolled_w;
    Bitmap          *list = (Bitmap *) NULL;
    char            buf[128], *p;
    Arg             args[12];
    int             n;
    XFontStruct     *font;
    int             istty = isatty(0), redirect = !istty, i = 0, total = 0;
    unsigned int    bitmap_error;
    int             j, k;
    void            scrolled(Widget, XtPointer, XtPointer);
    void            expose_resize(Widget, XtPointer, XtPointer);

    XtSetLanguageProc (NULL, NULL, NULL);
    toplevel = XtVaOpenApplication (&app, "XShowbitmap", options,
                                XtNumber (options), &argc, argv, NULL,
                                sessionShellWidgetClass, NULL);
    dpy = XtDisplay (toplevel);

    XtGetApplicationResources (toplevel, &Resrcs,
                                resources, XtNumber (resources), NULL, 0);

    if (Resrcs.rows && Resrcs.cols)
        XtWarning ("You can't specify both rows *and* columns.");

    font = Resrcs.font;

    /* check to see if we have to load the bitmaps from stdin */
    if (!argv[1] || !strcmp(argv[1], "-")) {
        printf ("Loading bitmap names from standard input. ");
        if (istty) {
            puts ("End with EOF or .");
            redirect++;
        }
```

```
    else
        puts ("Use -v to view bitmap names being loaded.");
}
else if (!istty && strcmp(argv[1], "-")) {
    printf ("%s: either use pipes or specify bitmap names.\n", argv[0]);
    exit (1);
}

/* Now, load the bitmap file names */
while (*++argv || redirect) {
    if (!redirect)
        /* this may appear at the end of a list of filenames */
        if (!strcmp (*argv, "-"))
            redirect++; /* switch to stdin prompting */
        else
            (void) strcpy (buf, *argv);
    if (redirect) {
        if (istty)
            printf ("Bitmap file: "), fflush(stdout);
        if (!fgets (buf, sizeof buf - 1, stdin) || !strcmp (buf, ".\n"))
            break;
        buf[strlen (buf) - 1] = 0; /* plug a null at the newline */
    }
    if (!buf[0])
        continue;
    if (Resrcs.verbose)
        printf ("Loading \"%s\"...", buf), fflush(stdout);
    if (i == total) {
        total += 10; /* allocate bitmap structures in groups of 10 */
        if (!(list = (Bitmap *) XtRealloc ((char *) list,
                                    total * sizeof (Bitmap))))
            XtError ("Not enough memory for bitmap data");
    }
    if ((bitmap_error = XReadBitmapFile (dpy, DefaultRootWindow(dpy),
                                buf, &list[i].width, &list[i].height,
                                &list[i].bitmap,
                                &j, &k)) == BitmapSuccess) {
        if (p = rindex (buf, '/'))
            p++;
        else
            p = buf;
        if (Resrcs.max_height && list[i].height > Resrcs.max_height ||
                Resrcs.max_width && list[i].width > Resrcs.max_width) {
            printf ("%s: bitmap too big\n", p);
            XFreePixmap (dpy, list[i].bitmap);
            continue;
        }
        list[i].len = strlen (p);
        list[i].name = strcpy (XtMalloc (list[i].len + 1), p);
        if (Resrcs.verbose)
            printf ("size: %dx%d\n", list[i].width, list[i].height);
        i++;
    }
    else {
```

```
            printf ("Couldn't load bitmap: ");
            if (!istty && !Resrcs.verbose)
                printf("\"%s\": ", buf);
            switch (bitmap_error) {
                case BitmapOpenFailed  : puts ("open failed.");
                                         break;
                case BitmapFileInvalid : puts ("bad file format.");
                                         break;
                case BitmapNoMemory    : puts ("not enough memory.");
                                         break;
            }
        }
    }
    if ((total = i) == 0) {
        puts ("couldn't load any bitmaps.");
        exit (1);
    }
    printf ("Total bitmaps loaded: %d\n", total);
    if (Resrcs.sort) {
        printf ("Sorting bitmaps...");
        fflush (stdout);
        qsort (list, total, sizeof (Bitmap), size_cmp);
        putchar ('\n');
    }

    /* calculate size for pixmap by getting the dimensions of each bitmap. */
    printf ("Calculating sizes for pixmap...");
    fflush (stdout);
    for (i = 0; i < total; i++) {
        if (list[i].width > cell_width)
            cell_width = list[i].width;
        if (list[i].height > cell_height)
            cell_height = list[i].height;
        if (Resrcs.label_bitmap && (j = XTextWidth (font, list[i].name,
                                                    list[i].len)) > cell_width)
            cell_width = j;
    }

    /* Compensate for vertical font height if label_bitmap is true.
    ** Add value of grid line weight and a 6 pixel padding for aesthetics.
    */
    cell_height += Resrcs.grid + 6 + Resrcs.label_bitmap *
                                        (font->ascent + font->descent);
    cell_width += Resrcs.grid + 6;

    /* if user didn't specify row/column layout figure it out ourselves.
    ** optimize layout by making it "square".
    */
    if (!Resrcs.rows && !Resrcs.cols) {
        Resrcs.cols = int_sqrt (total);
        Resrcs.rows = (total + Resrcs.cols - 1) / Resrcs.cols;
    }
    else if (Resrcs.rows)
        /* user specified rows -- figure out columns */
```

```
    Resrcs.cols = (total + Resrcs.rows - 1) / Resrcs.rows;
else
    /* user specified cols -- figure out rows */
    Resrcs.rows = (total + Resrcs.cols - 1) / Resrcs.cols;

printf ("Creating pixmap area of size %dx%d (%d rows, %d cols)\n",
                    Resrcs.cols * cell_width, Resrcs.rows * cell_height,
                    Resrcs.rows, Resrcs.cols);

if (!(pixmap = XCreatePixmap (dpy, DefaultRootWindow(dpy),
                    Resrcs.cols * cell_width,
                    Resrcs.rows * cell_height,
                    DefaultDepthOfScreen (XtScreen (toplevel)))))
    XtError ("Can't Create pixmap.");

if (!(gc = XCreateGC (dpy, pixmap, NULL, 0)))
    XtError ("Can't create gc.");
XSetForeground (dpy, gc, Resrcs.bg); /* init GC's foreground to bg */
XFillRectangle (dpy, pixmap, gc, 0, 0,
Resrcs.cols * cell_width, Resrcs.rows * cell_height);
XSetForeground (dpy, gc, Resrcs.fg);
XSetBackground (dpy, gc, Resrcs.bg);
XSetFont (dpy, gc, font->fid);
if (Resrcs.grid) {
    if (Resrcs.grid != 1)
        /* Line weight of 1 is faster when left as 0 (the default) */
        XSetLineAttributes (dpy, gc, Resrcs.grid, 0, 0, 0);
    for (j = 0; j <= Resrcs.rows * cell_height; j += cell_height)
        XDrawLine (dpy, pixmap, gc, 0, j, Resrcs.cols * cell_width, j);
    for (j = 0; j <= Resrcs.cols * cell_width; j += cell_width)
        XDrawLine (dpy, pixmap, gc, j, 0, j,
    Resrcs.rows * cell_height);
}

/* Draw each of the bitmaps into the big picture */
for (i = 0; i < total; i++) {
    int x = cell_width * (i % Resrcs.cols);
    int y = cell_height * (i / Resrcs.cols);
    if (Resrcs.label_bitmap)
        XDrawString (dpy, pixmap, gc,
                    x + 5 + Resrcs.grid / 2,
                    y + font->ascent + Resrcs.grid / 2,
                    list[i].name, list[i].len);
    if (DefaultDepthOfScreen (XtScreen (toplevel)) > 1)
        XCopyPlane (dpy, list[i].bitmap, pixmap, gc,
                    0, 0, list[i].width, list[i].height,
                    x + 5 + Resrcs.grid / 2,
                    y + font->ascent + font->descent +
                    Resrcs.grid / 2, 1L);
    else
        XCopyArea (dpy, list[i].bitmap, pixmap, gc,
                    0, 0, list[i].width, list[i].height,
                    x + 5 + Resrcs.grid / 2,
                    y + font->ascent + font->descent + Resrcs.grid / 2);
```

```
                    XFreePixmap (dpy, list[i].bitmap);
    XtFree (list[i].name);
}
XtFree ((char *) list);

/* Now we get into the Motif stuff */

/* Create automatic Scrolled Window */
n = 0;
XtSetArg (args[n], XmNscrollingPolicy, XmAPPLICATION_DEFINED); n++;
XtSetArg (args[n], XmNvisualPolicy, XmVARIABLE);               n++;
XtSetArg (args[n], XmNshadowThickness, 0);                     n++;
scrolled_w = XmCreateScrolledWindow (toplevel, "scrolled_w", args, n) ;

/* Create a drawing area as a child of the ScrolledWindow.
** The DA's size is initialized (arbitrarily) to view_width and
** view_height. The ScrolledWindow will expand to this size.
*/
n = 0;
XtSetArg (args[n], XmNwidth, Resrcs.view_width);   n++;
XtSetArg (args[n], XmNheight, Resrcs.view_height); n++;
drawing_a = XmCreateDrawingArea (scrolled_w, "drawing_a", args, n) ;

XtAddCallback (drawing_a, XmNexposeCallback, expose_resize, NULL);
XtAddCallback (drawing_a, XmNresizeCallback, expose_resize, NULL);
XtManageChild (drawing_a) ;

/* Application-defined ScrolledWindows won't create their own
** ScrollBars. So, we create them ourselves as children of the
** ScrolledWindow widget. The vertical ScrollBar's maximum size is
** the number of rows that exist (in unit values). The horizontal
** ScrollBar's maximum width is represented by the number of columns.
*/
n = 0;
j = min (Resrcs.view_height / cell_height, Resrcs.rows);
XtSetArg (args[n], XmNorientation, XmVERTICAL) ; n++;
XtSetArg (args[n], XmNmaximum, Resrcs.rows) ;    n++;
XtSetArg (args[n], XmNsliderSize, j) ;           n++;
vsb = XmCreateScrollBar (scrolled_w, "vsb", args, n) ;
XtManageChild (vsb) ;

if (Resrcs.view_height / cell_height > Resrcs.rows)
    sw_voffset = (Resrcs.view_height - Resrcs.rows * cell_height) / 2;

n = 0;
j = min (Resrcs.view_width / cell_width, Resrcs.cols);
XtSetArg (args[n], XmNorientation, XmHORIZONTAL) ; n++;
XtSetArg (args[n], XmNmaximum, Resrcs.cols) ;      n++;
XtSetArg (args[n], XmNsliderSize, j) ;             n++;
hsb = XmCreateScrollBar (scrolled_w, "hsb", args, n);
XtManageChild (hsb);

if (Resrcs.view_width / cell_width > Resrcs.cols)
    sw_hoffset = (Resrcs.view_width - Resrcs.cols * cell_width) / 2;
```

```
    /* Allow the ScrolledWindow to initialize itself accordingly...*/
    XtVaSetValues (scrolled_w,
                   XmNhorizontalScrollBar, hsb,
                   XmNverticalScrollBar, vsb,
                   XmNworkWindow, drawing_a,
                   NULL);

    XtAddCallback (vsb, XmNvalueChangedCallback, scrolled,
                   (XtPointer) XmVERTICAL);
    XtAddCallback (hsb, XmNvalueChangedCallback, scrolled,
                   (XtPointer) XmHORIZONTAL);
    XtAddCallback (vsb, XmNdragCallback, scrolled,
                   (XtPointer) XmVERTICAL);
    XtAddCallback (hsb, XmNdragCallback, scrolled,
                   (XtPointer) XmHORIZONTAL);

    XtManageChild (scrolled_w);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

/* scrolled() -- react to scrolling actions; cbs->value is ScrollBar's
** new position.
*/
void scrolled(Widget scrollbar, XtPointer client_data, XtPointer call_data)
{
    int orientation = (int) client_data;
    XmScrollBarCallbackStruct *cbs =
                           (XmScrollBarCallbackStruct *) call_data;

    if (orientation == XmVERTICAL)
        pix_voffset = cbs->value * cell_height;
    else
        pix_hoffset = cbs->value * cell_width;
    redraw (XtWindow (drawing_a));
}

/* expose_resize() -- handles both expose and resize (configure) events.
** For XmCR_EXPOSE, just call redraw() and return. For resizing,
** we must calculate the new size of the viewable area and possibly
** reposition the pixmap's display and position offset. Since we
** are also responsible for the ScrollBars, adjust them accordingly.
*/
void expose_resize(Widget drawing_a, XtPointer client_data,
                                            XtPointer call_data)
{
    XmDrawingAreaCallbackStruct *cbs =
                           (XmDrawingAreaCallbackStruct *) call_data;
    Dimension view_width, view_height, oldw, oldh;
    int do_clear = 0;

    if (cbs->reason == XmCR_EXPOSE) {
        redraw (cbs->window);
```

```
    return;
}
oldw = Resrcs.view_width;
oldh = Resrcs.view_height;

/* Unfortunately, the cbs->event field is NULL, we have to have
** get the size of the drawing area manually.
*/
XtVaGetValues (drawing_a,
                    XmNwidth, &Resrcs.view_width,
                    XmNheight, &Resrcs.view_height,
                    NULL);

/* Get the size of the viewable area in "units lengths" where
** each unit is the cell size for each dimension. This prevents
** rounding error for the {vert,horiz}_start values later.
*/
view_width = Resrcs.view_width / cell_width;
view_height = Resrcs.view_height / cell_height;

/* When the user resizes the frame bigger, expose events are generated,
** so that's not a problem, since the expose handler will repaint the
** whole viewport. However, when the window resizes smaller, then no
** expose event is generated. In this case, the window does not need
** to be redisplayed if the old viewport was smaller than the pixmap.
** (The existing image is still valid--no redisplay is necessary.)
** The window WILL need to be redisplayed if:
** 1) new view size is larger than pixmap (pixmap needs to be centered).
** 2) new view size is smaller than pixmap, but the OLD view size was
**   larger than pixmap.
*/
if ( (int) view_height >= Resrcs.rows) {
    /* The height of the viewport is taller than the pixmap, so set
    * pix_voffset = 0, so the top origin of the pixmap is shown,
    * and the pixmap is centered vertically in viewport.
    */
    pix_voffset = 0;
    sw_voffset = (Resrcs.view_height - Resrcs.rows * cell_height) / 2;
    /* Case 1 above */
    do_clear = 1;
    /* scrollbar is maximum size */
    view_height = Resrcs.rows;
}
else {
    /* Pixmap is larger than viewport, so viewport will be completely
    ** redrawn on the redisplay. (So, we don't need to clear window.)
    ** Make sure upper side has origin of a cell (bitmap).
    */
    pix_voffset = min (pix_voffset,
                    (Resrcs.rows-view_height) * cell_height);
    sw_voffset = 0; /* no centering is done */
    /* Case 2 above */
    if (oldh > Resrcs.rows * cell_height)
        do_clear = 1;
```

```
    }
    XtVaSetValues (vsb,
                   XmNsliderSize, max (view_height, 1),
                   XmNvalue, pix_voffset / cell_height,
                   XmNpageIncrement, max (view_height - 1, 1),
                   NULL);

    /* identical to vertical case above */
    if ( (int) view_width >= Resrcs.cols) {
        /* The width of the viewport is wider than the pixmap, so set
        ** pix_hoffset = 0, so the left origin of the pixmap is shown,
        ** and the pixmap is centered horizontally in viewport.
        */
        pix_hoffset = 0;
        sw_hoffset = (Resrcs.view_width - Resrcs.cols * cell_width) / 2;
        /* Case 1 above */
        do_clear = 1;
        /* scrollbar is maximum size */
        view_width = Resrcs.cols;
    }
    else {
        /* Pixmap is larger than viewport, so viewport will be completely
        ** redrawn on the redisplay. (So, we don't need to clear window.)
        ** Make sure left side has origin of a cell (bitmap).
        */
        pix_hoffset = min (pix_hoffset,
                    (Resrcs.cols - view_width) * cell_width);
        sw_hoffset = 0;
        /* Case 2 above */
        if (oldw > Resrcs.cols * cell_width)
            do_clear = 1;
    }
    XtVaSetValues (hsb,
                   XmNsliderSize, max (view_width, 1),
                   XmNvalue, pix_hoffset / cell_width,
                   XmNpageIncrement, max (view_width - 1, 1),
                   NULL);

    if (do_clear)
        /* XClearWindow() doesn't generate an ExposeEvent */
        XClearArea (dpy, cbs->window, 0, 0, 0, 0, True);
}

void redraw(Window window)
{
    XCopyArea (dpy, pixmap, window, gc, pix_hoffset, pix_voffset,
            Resrcs.view_width, Resrcs.view_height, sw_hoffset, sw_voffset);
}
```

The output of the example is shown in Figure A-1

.

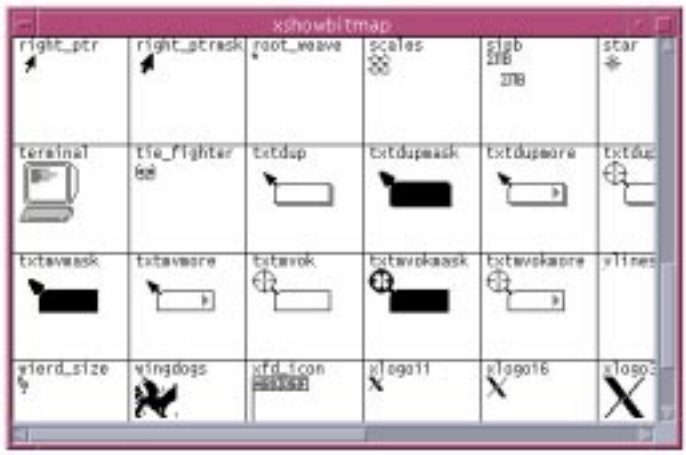


Figure A-1 Output of the xshowbitmap program

A Memo Calendar

The *xmemo* program creates a main application window that contains a calendar and a list of months. Selecting a month changes the calendar, while selecting a day causes that date to become activated. When a date is activated, the application displays another window that contains a Text widget. The Text widget could be used to keep a memo for that day if you were to add code to save and retrieve the contents of the memo. If you select the same day a second time, the window is popped down. Figure A-2 shows the output of the program.

The program shown in Example A-2 demonstrates a number of very subtle quirks about X and Motif programming. What separates simple programs from sophisticated ones is how well you get around quirks like the ones demonstrated in this example. For example, the way the dates in the calendar are handled is not as simple as it might appear. Unlike the *xcal* example in Chapter 12, *Labels and Buttons*, which used a single Label widget as the calendar, here each date in a month is a separate PushButton widget. To give the appearance that the calendar is a single flat area, the `XmNShadowThickness` of each PushButton is initialized to `0`. When a date is selected, the shadow thickness for that PushButton is reset to `2` (the default) to provide visual feedback that there is a memo associated with it.*

* `XtVaAppInitialize`() is considered deprecated in X11R6. The SessionShell and `XtVaOpenApplication`() are only available in X11R6. `XmStringGenerate`() is only available in Motif 2.0 and later.

Example A-2 The xmemo.c program

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

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

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

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

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

/* These only take effect if the app-defaults file is not found */
```

```
String fallback_resources[] = {
    "*bold.fontName: -*-courier-bold-r-*--18-*",
    "*bold.fontType: FONT_IS_FONT",
    "*medium.fontName: -*-courier-medium-r-*--18-*",
    "*medium.fontType: FONT_IS_FONT",
    "*XmPushButton*.renderTable: bold",
    "*XmLabelGadget*.renderTable: medium",
    NULL
};

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

    XtSetLanguageProc (NULL, NULL, NULL);

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

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

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

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

```
        for (month = 0; month < XtNumber (months); month++)
            XmStringFree (strs[month]);
        XtAddCallback (list_w, XmNbrowseSelectionCallback, set_month, NULL);
        XtManageChild (list_w);
    }

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

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

    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

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

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

    if (month >= 0 && months[month].form)
        XtUnmanageChild (months[month].form); /* unmanage last month */
    month = list_cbs->item_position - 1; /* set new month */
    sprintf (text, "%s %d", months[month].name, year);
    xms = XmStringGenerate ((XtPointer) text, NULL, XmCHARSET_TEXT, NULL);
    XtVaSetValues (month_label, XmNlabelString, xms, NULL);
    XmStringFree(xms);
    if (months[month].form) {
```

```
    /* it's already been created -- just manage and return */
    XtManageChild (months[month].form);
    return;
}

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

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

/* We are creating a whole bunch of PushButtons, but not all of
* them have dates associated with them. The buttons that have
* dates get the number sprintf'ed into it. All others get two blanks.
*/
for (day = i = 0; i < 6; i++) {
    for (j = 0; j < 7; j++, m += (j > m && --tot > 0)) {
        char *name;
        if (j != m || tot < 1)
            name = " ";
        else {
            sprintf(text, "%2d", ++day);
            name = text;
        }
        n = 0;
        /* this is where we will hold the dialog later. */
        XtSetArg (args[n], XmNuserData, NULL);                      n++;
        XtSetArg (args[n], XmNsensitive, (j % 7 == m && tot > 0)); n++;
        XtSetArg (args[n], XmNshadowThickness, 0);                  n++;
        months[month].dates[i][j] = XmCreatePushButton (months[month].
                                        form, name, args, n);
        XtManageChild (months[month].dates[i][j]);
        XtAddCallback (months[month].dates[i][j], XmNactivateCallback,
                                        date_dialog,(XtPointer) day);
    }
    m = 0;
}
XtManageChild (months[month].form);

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

```
    ** have a visible border and make it appear preselected, so, we have
    ** to make it appear invisible. If it is invisible then it cannot be
    ** selected, but it just so happens that the last 5 days in
    ** the month will never have selectable dates, so we can use any one
    ** of those. To make the button invisible, we need to unmap the
    ** widget. We can't simply unmanage it or the parent won't consider
    ** its size, which defeats the whole purpose. We can't create the
    ** widget and then unmap it because it has not been realized, so it
    ** does not have a window yet. We don't want to realize and manage
    ** the entire application just to realize this one widget, so we
    ** set XmNmappedWhenManaged to False along with the shadow thickness
    ** being set to 2. Now the RowColumn is the right size.
    */
    XtVaSetValues (months[month].dates[5][6],
                                XmNshadowThickness, 2,
                                XmNmappedWhenManaged, False,
                                NULL);
}

/* date_dialog() -- when a date is selected, this function is called.
** Create a dialog (toplevel shell) that contains a multiline text
** widget for memos about this date.
*/
void date_dialog(Widget w, XtPointer client_data, XtPointer call_data)
{
    int                 date = (int) client_data;
    Widget              dialog;
    XWindowAttributes   xwa;

    /* the dialog is stored in the PushButton's XmNuserData */
    XtVaGetValues (w, XmNuserData, &dialog, NULL);
    if (!dialog) {
        /* it doesn't exist yet, create it. */
        char   buf[32];
        Arg    args[5];
        int    n, n_pos, *list;

        /* get the month that was selected -- we just need it for its name */
        if (!XmListGetSelectedPos (list_w, &list, &n_pos))
            return;
        sprintf (buf, "%s %d %d", months[list[0]-1].name, date, year);
        XtFree ((char *) list);
        dialog = XtVaCreatePopupShell ("popup", topLevelShellWidgetClass,
                                    XtParent (w),
                                    XmNtitle,  buf,
                                    XmNallowShellResize, True,
                                    XmNdeleteResponse,  XmUNMAP,
                                    NULL);
        n = 0;
        XtSetArg (args[n], XmNrows,   10);                  n++;
        XtSetArg (args[n], XmNcolumns, 40);                 n++;
        XtSetArg (args[n], XmNeditMode, XmMULTI_LINE_EDIT); n++;
        XtManageChild (XmCreateScrolledText (dialog, "text", args, n));
        /* set the shadow thickness to 2 so user knows there is a memo
```

```
        ** attached to this date.
        */
        XtVaSetValues (w,
                       XmNuserData, dialog,
                       XmNshadowThickness, 2,
                       NULL);
    }
    /* See if the dialog is realized and is visible. If so, pop it down */
    if (XtIsRealized (dialog) &&
                  XGetWindowAttributes (XtDisplay (dialog),
                  XtWindow (dialog), &xwa) &&
                  xwa.map_state == IsViewable) {
        XtPopdown (dialog);
    }
    else
        XtPopup (dialog, XtGrabNone);
}

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

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

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

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

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

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

    year -= 1900;
    days_ctr = ((long)year * 365L) + ((year + 3) / 4);
    days_ctr += mtbl[month - 1] + day + 6;
    if (month > 2 && (year % 4 == 0))
    days_ctr++;
    return (int) (days_ctr % 7L);
}
```

The output from the xmemo program is given in Figure A-2.

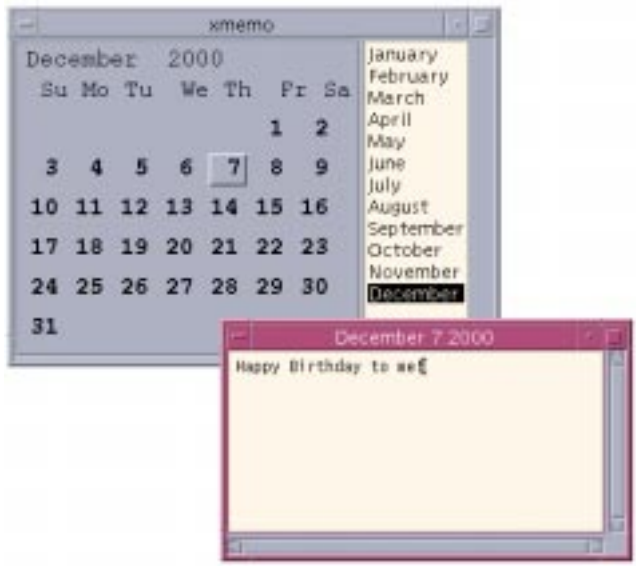


Figure A-2 Output of the xmemo program

Index

SYMBOLS

.mwmrc file 650
_MOTIF_WM_MESSAGES property 667

NUMERICS

3-D widgets 57, 238, 271

A

accelerators 882
    menu 591, 608
action areas 219–227
    and dialogs 78
    buttons 133, 206, 227
    creating 222
    generalizing 221
    in dialogs 131
action routines
    and text widgets 527
actions 14
add_item callback routine 442
alignment
    widget 257, 395
animated widgets 699, 700, 755
ANSI
    compilers 24, 36
    function declarations 36
ANSI-C 19
app-defaults file 13
application class 21
application context 21
application manager
    constructing 430
application-defined scrolling 321
    and main windows 102
    example 342–356, 893
ApplicationShell
    and dialogs 204, 230
    creating 204
    versus DialogShell 152
Apply button
    managing 177
argument lists
    and setting resources 29
arm callbacks 400
    and ToggleButton 412
ArrowButton 387, 421–427
arrows
    and scrollbars 331
    directional 421
    for traversing widgets 358
atoms 638, 694
    and types in ICCCM 700
    protocol 651
attachments
    default 257
    example 257
    Form 243
    offsets 250–253
    position 253, 255
    resources 217
automatic scrolling
    example 325
automatic~scrolling 319–321, 324
auxiliary area 584

B

background tasks
executing 885–887
backing store
and redrawing a DrawingArea 368
backspace key
remapping 34
backspacing
and Text 575
binary searches
and lists 442
bindings 33
Boolean state
ToggleButton 406
borders
3-D 57
color of 83
window manager 648
bugs
event field 578
PanedWindow 216
Scale 502
translation tables 372
window manager 646
Xt and varargs 24
BulletinBoard 237–243
and dialog shells 237
and dialogs 130
and font lists 238
and geometry management 239–243
and translation tables 241
creating 238
resizing 241
button clicks
maximum time between 556
multiple 403
buttons 387–431
and menus 591
ArrowButton (see ArrowButton)
CascadeButton (see CascadeButton)
dialog
default 143
keyboard focus 145
pre-defined 132
setting 136
sizes 145
DrawnButton (see DrawnButton)
in action areas 133, 206, 227
multi-colored 85
PushButton; (see PushButton)
radio buttons 48
tear-off 614
ToggleButton (see ToggleButton)

C

C strings
and menu accelerator text 609
converting to compound strings 805, 807
caching
and pixmaps 80
callback reasons 503
XmAddWMProtocolCallback() 653
(see also callbacks)
callback resources (see XmN entries for individual callback resources)
callback routines
CascadeButton 105
dialog 147, 220
FileSelectionDialog 186
List 454–460
scrollbar 338
SimpleMenu 108
callback structures
XmCommandCallbackStruct 181
XmDragDropFinishCallbackStruct 736
XmDragProcCallbackStruct 754
XmDropProcCallbackStruct 746
XmDropSiteEnterCallbackStruct 734
XmFileSelectionBoxCallbackStruct 186
XmRowColumnCallbackStruct 269
XmSelectionBoxCallbackStruct 176
XmTextVerifyCallbackStruct 569
callbacks
and CascadeButtons 105
and clipboards 687
and DrawingArea 363
arm 400, 412
CommandDialog 180
cursor movement 578
dialog 147, 220
example 147, 149
setting 136
disarm 400, 412

FileSelectionDialog 186
List 454–460
lists 13
popdown 141
popup 141
protocol 653
PushButton 400
reasons; (see callback reasons)
resources; (see callback resources)
routines; (see callback routines)
RowColumn 269
Scale 503
scrollbar 338
SimpleMenu 108
structures 108, 109
Text 565, 567–580
ToggleButton 412
cancelling a drag operation 699
CascadeButton 69, 387
and callback routines 105
and menu items 611
and MenuBars 103–106
and menus 589
and option menus 601
cascading menus 42, 589, 598
example 631
case sensitivity
and mnemonics 607
cell_height 354
cell_width 354
character sets 580, 803
charset 580, 804
CheckBox 418–421
and mask variables 420
compared to List 406
creating 418
versus RadioBoxes 418
CheckForInterrupt() 881
circular dependencies
Form 260
class names
application 21
classes
definition 12
client data 36
changing in callbacks 876
client messages 650–658
clip windows
and pixmaps 353
ClipboardBadFormat 689
ClipboardFail 679
ClipboardLocked 679, 680
ClipboardNoData 683
clipboards 673–696
and properties 694
and Text widgets 528
callback functions 687
copying data to 678, 692
by name 683, 684
incrementally 688
cut and paste 673
convenience routines 692
example 675–689
data formats 689, 691
locking 681, 688
messages, posting by name 675
querying for data size 682
retrieving data from
incrementally 680
with XmClipboardRetrieve() 680
with XmClipboardRetrieve-Copy() 678
with XmTextPaste() 693
routines
and Text widgets 551–556, 692
selection
and Text widgets 528
cut and paste 551
sending data to 678
terminating a copy 688
unlocking 681
versus primary and secondary selections 675, 695
ClipboardSuccess 679, 680, 681, 683
ClipboardTruncate 680, 681
close item
window manager 651
closing dialogs 140
color 83–85, 504
and DrawingArea 378–385
and gadgets 63
foreground 84
setting 84
Command 121, 180–181
command areas
in main windows 67, 118
Command.h 134
CommandBoxes 169

command-line arguments
specifying 22
commands
saving in a history list 180
Common User Access (CUA) 6
specifications 6, 283
composite widgets 55
compound objects 56
compound strings 777, 803–842
and Labels 46, 390
and menu accelerator text 609
and Text widgets 805
C strings, converting from 805, 807
concatenating 805
creating 805–819
dimensions of 841
freeing 806
in lists 436, 457
internationalization 806
language-independent 810
manipulating 819–823
rendering 839–841
retrieving 821
segments 814
specifying as normal strings 819
string direction 814
text, converting to 821–823
ConfigureNotify events 644
Constraint 62
resources 62, 217
control areas
and dialogs 78
dialog 131
control sashes 57
and PanedWindows 275
convenience functions 43
and dialogs 151
copying a file via drag and drop 728
copying and retrieving
clipboard 673–689
copying data via drag and drop 698
Core 46
CreateBitmapFromData() 81
creating
PushButtons 197, 357
ToggleButtons 357
ctrl key
and tab key 283
CUA (Common User Access) 6
specifications 6, 283
current items 64
cursor
callbacks 578
insertion 542–548
position 542
custom dialogs 78, 195
cut and paste
and Text widgets 551
clipboard 673
example 551, 675–689

D

data formats
clipboard 689
deactivating menu titles 606
default action
in list selection 454
default button
dialog 143
default font list tag 812
default language procedure 20
DefaultDepthOfScreen 355
delete key
remapping 34
deleting items from a list 444
deselecting items from a list 446
dialog actions 133
dialog boxes (see dialogs)
dialog callbacks
adding 148
routine for 198
dialog shells
BulletinBoard 237
versus dialog widgets 131
dialog widgets (see dialogs)
dialogs 74–80, 169–193
accessing internal widgets 155
action areas 131, 205–207, 219–227
buttons in 133
anatomy of 131–133
and BulletinBoards 130
and manager widgets 205
and shell widgets 73
and the window manager 141
buttons
default 143

fonts 147
keyboard focus 145
sizes 145
callback reasons 148
callback routines 147, 198, 220
example 149
closing 140–142
CommandDialog 77, 180–181
control areas 131, 205–207
creating 133–142, 203–221
custom 78, 195
definition 4
destroying 887
ErrorDialog 74
FileSelectionDialog 77, 181
fonts 813
full-application-modal 156
function of 128–131
header files 133
help 861–870
context-sensitive 869
example 862, 865
point-and-click 869
InformationDialog 75
interacting with other windows 156
labels
fonts 147
layout of 205
managing 136–140
MessageDialog 74, 75
modality 79, 156–167
example 162–164
implementing 159
setting 159
modifying 195–203
popping up 221, 231
positioning 229–231
pre-defined 127, 132
primary-application-modal 156
PromptDialog 77
QuestionDialog 75
relation to manager widgets 130
resizing 146
resources
setting 135, 215
reusing 140, 163
SelectionDialog 76, 169, 170–177
system-modal 156
TemplateDialog 75, 198
terminology 130
text
fonts 147
titlebar 146, 215
TopLevelShells, using with 227, 228
transient 128
unmanaging 140–142
unmapping 141
versus dialog shells 131
WarningDialog 75
WorkingDialog 75
DialogShell 152
and building dialogs 204
and convenience routines 152
and PanedWindow widgets 216
children of 216–219
creating 204
identifying 154
parents of 134, 205
using with RowColumns 269
versus ApplicationShells 152
versus TopLevelShells 152
direction
string 814
directional arrows 421
directory
dropping a file into a 728
directory searching 183, 191
disarm callbacks 400
and ToggleButtons 412
Display 701, 709, 713
do-not-enter symbol 699
double-clicking
on a list item 455
drag and drop 46, 50, 51, 529, 697–755
basic programming constructs 699–712
conceptual model 698
conversion procedures 700, 704, 729
customizing 700
dragging non-text data 719–736
dropping non-text data 736–755
encapsulating in widgets 755
handling the drop 746
implementation 700
Motif widgets 699
overview 697–699
programming model 709–712
protocols 708–709

starting the drag 727–729
transfer procedures 700
turning off 716–717
validity of drop 699, 706, 708, 712
drag icons 698, 706–707
coloring 703, 718, 728
components 706, 728
creating 727
customizing 700, 707
default operation icons for 707
default source icons for 707
destroying 729, 736
specifying default icons 717
drag protocols 708–709
dynamic 708, 711, 715
preregister 708, 711
resolution based on initiator and receiver 714
specifying 713–716
drag sources 698, 702–704
allowed data targets for 702
allowed operations on 702
and second mouse button 702
creating 725–727
file images as 726
incremental transfer of 703
modifying 730–733
multiple formats for 702
working with 719–736
DragContext 701, 702, 707
creating 703, 728
XmNdropStartCallback 735
DragIcon 701, 707, 718
drag-over visuals 699, 706
customizing 711, 734–735
modifying 717–719
DragStart() 727
drag-under visuals 699
customizing 700, 711, 715, 753–755
drawing
and backing store 368
and global variables 367
directly from the action function 373
freehand 372, 373
into windows 883
redrawing a DrawingArea 368
with color 378–385
Xlib 367
(see also DrawingArea)
DrawingArea 361–385
and color 378–385
and gadgets 367
and keyboard traversal 371
and scrollbars 377
as a manager widget 362
callbacks 363
children 367
clearing windows 364
creating 362
dragging 702
event-handling 363
example 364
geometry management 368
resizing 354
translation tables 363, 371
DrawingAreaInput() 372
DrawnButton 427–430
and PushButtons 427
creating 427
drop protocols 709
drop sites 698, 704–706
animated 700, 705, 755
creating 745
deactivating 716
drag-under visuals 705
garbage can 699
HELP on 699
modifying 745–746
operations supported by 704
overlapping 706, 711
providing help 749–753
registering widgets as 705
send message 704
shaping 706
working with 736–755
DropSite 701, 704, 716
XmNdropProc 735
XmNtransferProc 736
DropTransfer 701, 705, 709
creating 747

E

emacs
and Text widgets 527
encoding 580, 803
endpwent() 326

ErrorDialog 133
events
    field 578
    handling 32–39
    KeyPress 33
    KeyRelease 882
    processing 871, 877
    structures 670
    syntax 33
    timers 871
    translations 32, 288
    X event control loops 844
examples
    action areas, creating 222
    application managers, constructing 430
    application-defined scrolling 342–356
    attachments 257
    automatic scrolling 325
    bitmap display utility 904
    bitmaps, dynamically changing 110
    browse selection, specifying 455
    BulletinBoard geometry management 239
    button clicks, multiple 403
    calendars, creating 911
    changes in Motif 1.2 95
    CheckBox 418
    clipboards, copying data to 684
    Close item, mwm 651
    compound strings
        converting to text 823
        creating 810
    dialogs
        buttons, setting 136
        callback routines 136, 147, 149
        help 862, 865
        MessageDialog 887
        modal 162–164
        modifying 202
        positioning 229
        symbols 887
        WorkDialog 873
    DrawingArea 364
    dropping files into a text editor 736–744
        providing help 749–753
    file browser 536
    file manager with draggable files 719–725
FileSelectionDialog, creating 184
Form geometry management 248
Frame 273
freehand drawing 373
hello world, Motif-style 16
help dialogs 862, 865
icons, creating 646
keyboard traversal
    in ScrolledWindows 356–358
    processing manually 290
lists
    adding items to 440
    creating 435
    selecting items 447, 457
MainWindow
    displaying bitmaps in 110
    using a ScrolledList in 102
menu of common editing actions 551
MenuBars 110
menus
    cascading 619, 631
    help 610
    option 631
    popup 623, 631
    pulldown 616, 619, 623, 631
MessageDialog 164, 887
Motif 1.2 changes 95
multi-font strings, creating 817
PanedWindow 276, 279, 280
position attachments 255
PromptDialog
    creating 177
protocols 656, 667
RadioBoxes 414, 418
RowColumn 262, 264, 266, 270
Scales 498, 504
scrolling
    application-defined 342–356
    automatic 325
SelectionDialog 170
shell, resizing 641, 643
text
    converting to uppercase 568
    pattern, searching for 542
text editor 556
    dropping files into 736, 749–753
Text widgets, preventing text modification in 571

tic-tac-toe 255
timers, using 887
ToggleButtons 409
updating an existing drag source 731
window manager functions 651
WorkDialog 873, 877
XmCreateSimpleMenuBar() 106
XmNentryCallback 270
XmNsymbolPixmap 887
XtAppAddTimeOut() 887
exec() system call 885
execvp()
system call 853
Expose events 883
processing 882
expose_resize() 354
extended selection mode 458
external commands
executing 885–887

F

fallback resources 23
file browser 536
file descriptors
and running external processes 887
file manager with draggable files 719–725
file objects 736
file type masks 192
XmFILE_REGULAR 192
files
searching with FileSelectionDialog 186–191
selecting with FileSelectionDialog 181
FileSB.h 134
FileSelectionBox 169
FileSelectionDialog 110, 181, 536
callback routines 186
creating 183–185
searching directories 191
filesystem searches 186–191
FMT16BIT 570
FMT8BIT 570
focus~callbacks 579
font list tags 808–814
font lists 808–814
and BulletinBoard 238
internationalization 812
fonts
and compound strings 806
and dialogs 813
and Labels 396
used by widgets 813
ForceUpdate() 884
fork() system call 885
Form 243–261
and geometry management 248
attachments 217, 243
circular dependencies 260
common problems with 260
positions 253
resizing 244
Frame 234, 271–275
fread() 541
freehand drawing 372, 373
full-application-modal dialogs 156
function overloading 367
functions
callbacks
(see also callback routines)
popen 885
system 885

G

gadgets 63
and DrawingArea widgets 367
class hierarchy 53
coloring 63
creating 63
definition 25
header files 18
Label (see LabelGadget)
managing 63
pointers 389
ToggleButtonGadget 406
translation tables 290
GC (graphics context) 367
geometry management 61–63, 233–294
and BulletinBoard 239–243
and DrawingArea 368
and Form 243, 248
and RowColumn 262
getpwent() 326
global variables
and drawing 367

graphics context 367
grips 57, 275
GUI buttons
    and menus 591

H

hard-coded resources 27
header files 18
    dialog 133
    private 18
hello world program
    Motif-style 16
help
    and drop sites 699
    and Labels 286
help dialogs 861–870
    context-sensitive 869
    example 862, 865
    multi-level 865
    point-and-click 869
help keys 864
help menus 609–611
HOME 82
hooks (see attachments)
horizontal alignment of widgets 257
horizontal scrollbars 50
    and scrolled lists 438

I

ICCCM
    and clipboard functions 675
    and drag and drop 700
    and window managers 73
icons 645–647
    creating 646
    naming 647
    setting pixmap 645
    setting position 647
images
    and Label widgets and gadgets 391
    installing personalized 82
    names 81
    uninstalling 82
InformationDialog 133
    as help dialog 861
    for help about drag and drop 749, 753
initializing the toolkit 20–24
input context 584
input focus
    setting 293
input manager 584
input method 583
insertion cursor 527, 542–548
instantiating widgets 12
int_sqrt() 391
interactivePlacement 230
Inter-Client Communications Conventions Manual (ICCCM)
    drag and drop 700
    target types 700, 726
    window sizing and placement 638
interface design 6–9
    basic concepts 2
internationalization 19, 580, 803–804
    and compound strings 806, 841
    font lists 812

J

justification
    and Label widgets 395

K

keyboard traversal 283–293
    and DrawingAreas 371
    and ScrolledWindows 356–358
    processing manually 290
    translation table 288
KeyPress events 33
KeyRelease events 882
keysyms 33

L

Label 387–431
    aligning 395
    and color 85
    and PushButton 400
    and scrolled windows 342
    and Text widgets 388
    creating 389
    desensitizing 394

dragging and dropping 46, 699, 712
fonts 396, 813
help 286
images 391
justification 395
multi-colored 85
text 390
LabelGadget 389
LANG environment variable 20, 82
language procedure 20, 580, 803
libraries
Xlib 17
Xm 17
Xt 17
line wrapping
and Text widgets 541
linear search
and lists 442
linking data via drag and drop 698
List 433–461
and color 85
and tab groups 65
callback routines 454–460
installing 455
CheckBox, compared to 406
creating 434–437
dragging and dropping 50, 699, 713
RadioBox, compared to 406
selection policies 434, 436, 455
browse 434, 455
extended 434, 458
multiple 434, 457
single 434, 455
(see also lists)
lists
adding items to 439
example 440–442
deleting items from 444
deselecting items from 446
displaying 433
double-clicking on an item 455
finding items in 442
making items visible 437
positioning items in 452
replacing items in 443
scrolled (see scrolled lists)
searching
binary 442
linear 442
selecting items from 439, 445, 454
default action 454
example 447
loading pixmaps 117
locale 19, 580, 803
localization 19, 580, 803
locking
clipboards 678, 681, 688

M

main windows 66
command areas 118
configurability 123
message areas 118
suggested layout 97
using resources with 123
(see also MainWindow)
MainWindow 66, 551
creating 98
default size 101
layout for 67
using a ScrolledList in 102
when to use 97
manager widgets 25, 53–65, 233–294
and composite widgets 55
and dialogs 130, 205
and gadgets 63
and geometry management 61
and keyboard traversal 65, 356
and Shells 53
and XmNnavigationType 287
class hierarchy 55
creating 235–237
DrawingArea 362
translation table 288
ManagerEnter() 289
ManagerFocusIn() 289
ManagerFocusOut() 289
ManagerGadgetActivate() 290
ManagerGadgetArm() 290
ManagerGadgetButtonMotion() 290
ManagerGadgetDrag() 290
ManagerGadgetHelp() 290
ManagerGadgetKeyInput() 290
ManagerGadgetMultiActivate() 290
ManagerGadgetMultiArm() 290
ManagerGadgetNextTabGroup() 289

ManagerGadgetPrevTabGroup() 289
ManagerGadgetSelect() 290
ManagerGadgetTraverseDown() 289
ManagerGadgetTraverseHome() 289
ManagerGadgetTraverseLeft() 289
ManagerGadgetTraverseRight() 289
ManagerGadgetTraverseUp() 289
ManagerLeave() 289
ManagerParentActivate() 290
ManagerParentCancel() 290
mask variables
and CheckBoxes 420
menu bars
definition 4
MenuBar 103–118, 553
and CascadeButtons 103–106
and main windows 103
callback routines 108
children of 69
creating 103, 603, 618
example 110
item types 106
menus 589–636
accelerators 591, 608
Motif versus X Toolkit 608
and buttons 591
CascadeButtons 589
GUI buttons 591
ToggleButtons 623
and mnemonics 591, 607
and shell widgets 72
cascading 42, 598
creating 619
creating
example 616
designing 603–614
help 609–611
items 606
data structure 615
deactivating 611
setting and resetting sensitivity 611
Motif versus standard Xt 592
option 631–636
pop-up
building 623
pulldown 42, 106, 619
pulldown;
(see also pulldown menus)
pullright (see menus, cascading)
simple 592–603
submenus, building 623
titles 606, 611
message areas 67
in main windows 118
MessageB.h 133
MessageBox 130
MessageDialog 127
convenience routines for 151
creating 133, 142
definition 130
example 164
modifying 196–198
symbols 887–892
types of 133
messages
clipboard, posting by name 675
Microsoft Windows
Common User Access (CUA) specifications 6
mnemonics
and menus 591, 607
case sensitivity of 607
modal dialogs 79, 156–167
example 162–164
implementing 159
Motif
about 3–6
and Microsoft Windows 6
library 14–16
programming with 16–39
specifications 5
toolkit 5
versus X Toolkit 592, 608
Motif 1.2 35, 46, 50, 51, 57, 69, 75, 80, 84, 85–95, 271, 527, 529, 532, 533, 535, 591, 613, 806, 812, 813
drag and drop 697–755
Label 712
List 712
MessageDialog
additional children in 198
tear-off menus 69, 591, 613–614
TemplateDialog 75
Text and TextField 712
XmChangeColor() 84
XmFONTLIST_DEFAULT_TAG 812
XmGetFocusWidget() 293

XmGetPixmapByDepth() 80
XmGetTabGroup() 293
XmGetTearOffControl() 614
XmIsTraversable() 293
XmListAddItemsUnselected() 439
XmListDeletePositions() 445
XmListGetKbdItemPos() 453
XmListPosSelected() 446
XmListPosToBounds() 453
XmListReplaceItemsPosUnselected() 443
XmListReplaceItemsUnselected() 444
XmListReplacePositions() 444
XmListSetKbdItemPos() 453
XmListYToPos() 453
XmNchildHorizontalAlignment 273
XmNchildHorizontalSpacing 273
XmNchildVerticalAlignment 273
XmNdialogType value 198
XmNmodifyVerifyCallbackWcs 582
XmNpositionIndex 276
XmNtearOffMenuActivateCallback 614
XmNtearOffMenuDeactivateCallback 614
XmNtearOffModel 613
XmNtraverseObscuredCallback 356
XmNvalueWcs 581
XmRegisterSegmentEncoding() 812
XmStringCreateLocalized() 806
XmTextDisableRedisplay() 532
XmTextEnableRedisplay() 532
XmTextFindString() 545
XmTextGetSubstring() 533
XmTrackingEvent() 869
XmTranslateKey() 35
Motif Style Guide 41
key mapping conventions 698
mouse button conventions 698
moving data via drag and drop 698
multibyte strings 580, 804
multiClickTime 402, 556
multi-colored buttons 85
multi-colored labels 85
multi-font strings 809, 817–819
multi-line editing
and Text widgets 527, 534
multiple button clicks 403
multiple items
deleting from a list 444
selecting from a list 446, 457
MWM_DECOR_ALL 649
MWM_DECOR_BORDER 648
MWM_DECOR_MAXIMIZE 649
MWM_DECOR_MENU 649
MWM_DECOR_MINIMIZE 649
MWM_DECOR_RESIZEH 648
MWM_DECOR_TITLE 649
MWM_FUNC_ALL 650
MWM_FUNC_CLOSE 650
MWM_FUNC_MAXIMIZE 650
MWM_FUNC_MINIMIZE 650
MWM_FUNC_MOVE 650
MWM_FUNC_RESIZE 650

N

naming widgets 31
navigation groups 283, 284
navigation types
tab groups 287–288
newline character
interpreting as string separators 816

O

offsets
attachment 250–253
zero-length 252
off-the-spot interaction style 584
on-the-spot interaction style 584
Open Software Foundation (OSF) 5
operation icon (for a drag icon) 706
option menus 590
and CascadeButton 601
building 631–636
OptionMenu 69
output-only text 548
overriding~translation tables 372
over-the-spot interaction style 584

P

page length
scrollbars 332
PanedWindow 234, 275–283, 635

and DialogShells 216
bugs 216
example 276, 279, 280
resizing 216
sashes 283, 287
specifying resolution-independent dimensions 279
use in dialogs 205
password files
returning information about 326
pattern searches 451, 542
pix_hoffset
and scrolled windows 353
pix_voffset
and scrolled windows 353
pixels
converting scrollbar units to 354
pixmaps 80–83
and caching 80
and the clip window 353
icon 645
insensitive 394
loading 117
single-bit 80
single-plane 80
ToggleButton 409–412
pointers
xmLabelGadgetClass 389
popdown callbacks 141
popen() 396, 885
popup callbacks 141
popup dialogs 221, 231
popup menus 590, 593–595
building 623–631
compared to CheckBox and RadioBox 406
position attachments 253, 255
positioning dialogs 229–231
pre-edit area 584
PRIMARY selection property 695
and Text widgets 528
cut and paste 551
primary-application-modal dialogs 156
Primitive widgets 45
and tab groups 287
and XmNnavigationType 287
class hierarchy 46
private header files 18
programming
drag and drop 709–712
with Motif and Xt 16–39
PromptBoxes
(see also PromptDialog)
PromptDialog 134, 177–179
creating 177
TextField 179
properties
_MOTIF_WM_MESSAGES 667
and clipboards 694
definition 638
list of 639
protocol atoms 651
protocol callbacks 653
protocol widgets 654
protocols
adding 654
customizing 667–671
deactivating 669
for drag and drop 708–709
suites 669
window manager 71, 650–658
pulldown menus 42, 589
building 631
item types 106
pullright menus (see cascading menus)
PushButton 387, 400
and color 83
and DrawnButtons 427
and Labels 400
callbacks 367, 400
creating 197, 357, 400
fonts 813
in action areas 206
used in DrawingAreas 367
versus ArrowButton 49

Q

qualify search procedure 192
QuestionDialog 133
creating 196

R

radio buttons 48
RadioBox 115, 413–416
and XmVaRADIOBUTTON 107

compared to List 406
creating 414, 418
re_comp() 451
re_exec() 447, 451
realizing widgets 39
redraw()
and scrolling Text and List widgets 355
redrawing a DrawingArea 368
regcmp() 451
regex() 451
registering
clipboard data formats 689
regular expression matching 447
rendering compound strings 839–841
reparenting windows 145
replacing items in a list 443
resizing
BulletinBoards 241
dialogs 146
Form widgets 244
handles, window manager 648
RowColumns 263
scrolled window DrawingArea 354
Text widgets 542
resource database 13
resources
attachment 217
callbacks 429
color 83
configurable 13
constraint 217
for dialogs 143–147
for drag and drop 703
getting 26–32
hard-coding 13, 27
names 26
passing to more than one widget 30
setting
after widget creation 27
for dialogs 135
with convenience functions 28
shell 639
time 402
using with main windows 123
resources (see XmN for individual resources)
restarting applications 656
return key
and tab key 284
reusing
dialogs 140, 163
root-window interaction style 584
RowColumn 234, 261–271, 395
and CheckBoxes 418
and DialogShells 269
and Frame widgets 271
and geometry management 262
and RadioBoxes 414
and ToggleButtons 408
as a menubar 103
callbacks 269
creating 264–268
example 262, 264, 266
homogeneous children 268
resizing 263

S

Sash 283
modifying 287
Scale 497–508
bugs 502
callbacks 503
children 58
color 504
creating 498–500
labels 500
movement 501
values 500–501
Screen 701, 707, 717
setting resources for 718
ScrollBar 298, 319–359, 522
and color 83
retrieving values from 355
scrollbars
and DrawingAreas 377
and main windows 101–103
callback routines 338
design of 331
directional arrows 331
managing in scrolled lists 438
placing 541
scrolled lists 437–439
and Form widgets 260
scrolled text 260, 534
scrolled windows

automatic 338
creating 323
semi-automatic 338
scrolled() 353
scrolled~windows 319, 463, 475
ScrolledList 437
ScrolledText 218, 534
and displaying text 548
creating 535
dropping file data into 744
parent 535
ScrolledWindow 319–359, 463, 475
and keyboard traversal 356–358
and Lists 434, 437
and Scrollbar 50
default size 101
setting resources 218
scrolling 321
application-defined 321
example 342–356, 893
implementing 338
Lists 355
monitoring 338
Text widgets 355, 546
search and replace 546
search_item callback routine 451
SECONDARY selection property 528, 695
secondary windows 74
segments 814
select() system call 854
selecting items from a list 445
SelectioB.h 134
selection callbacks 455
selection methods (text)
modifying 555
selection policies 458
selection properties 695
SelectionBox 130, 169
SelectionDialog 128, 169, 170–177
callbacks 175
definition 134
example 170
types of 169
semi-automatic~scrolling 323
session managers 655–658
save-yourself protocol 656
setlocale() 19
Shell 70
and building dialogs 73, 204
class hierarchy 72
position 640
resizing 641–645
resources 639
shift key
and tab key 283
signal handling 843–859
in Xlib 846
in Xt 846–848
System V 854
timers 846, 854
work procedures 846
single-bit pixmaps 80
single-line editing
and Text widgets 526, 534
single-plane pixmaps 80
size
setting shell 641–645
source icon (for a drag icon) 706
spacebar key
and tab key 284
specifications
Motif 5
start_stop() 426
state icon (for a drag icon) 706
static~scrollbars 324
status area 584
strings
and menu accelerator text 609
arguments, passed to action functions 377
breaking into multiple lines 815
color, converting to 202
comparing in a list 442
compound 777, 803–842
and List widgets 436
and text Labels 390
(see also compound strings)
converting into pixels 203
direction 814
multibyte 580, 804
multi-font 809, 817–819
separators 815
wide-character 580, 804
style guide 41
submenus
building 623
suites
protocol 669

sw_hoffset 353
sw_voffset 353
symbols
MessageDialog 887–892
system() 885
system-modal dialogs 156

T

tab groups 283
and primitive widgets 287
excluding widgets from 285
navigation groups 284
navigation types 287–288
translation tables 288
traversing 64, 283
tab key 283
and shift key 283
target types
data 700
tear-off button 614
tear-off menus 69, 591, 613–614
TemplateDialog 198
Text 525–587
and clipboards 675
and color 85
and Label widgets 388, 390
and tab groups 65
and text deletion 572
and text editors 527
automatic resizing of 542
byte capacity of 533, 536
callbacks
single-line 565
text modification 567–578
clipboard functions 551–556, 692
clipboard selection 528
cut and paste 551
compound strings 805
creating 530
cut and paste 551
disabling drop site functionality 716
displaying file contents with, example 536
drag and drop 51, 712
dragging 699, 702
dropping data into 699, 713
editing modes 534
fonts 813
limitations of 525
line wrapping 541
modification 567–578
preventing 571
multiline editing 534
output-only 548
positions 542–548
PRIMARY selection 528
cut and paste 551
retrieving text 532
scrollable 534
scrolling 355, 546
search and replace 546
secondary selection 528
selecting 528, 555
setting resources on 218
single-line editing 526, 534
user interface for 527
varargs 548
XmNeditable 717
text editors
and Text widgets 525, 527
dropping files into 736–744
providing help 749–753
sample program 556
widget 51
TextField 51, 451
and color 85
creating 530
disabling drop site functionality 716
drag and drop 51, 712
dragging 699
dropping data into 699, 713
XmNeditable 717
(see also Text)
tick marks 506
TimeoutCursors() 881
timers
example 887
using to process tasks 876
Xt 846, 854
title bars
dialog 146, 215
strings, setting 146
window 649
ToggleButtonGadget 406
ToggleButtons 406–421
and color 83

and menus 623
callbacks 412
creating 357, 406–407, 409
determining state of 412
fonts 813
gadget 406
pixmaps 409–412
resources associated with 407
setting indicator width and height 409
setting state of 412
toggles (see ToggleButtons)
toolkit
initializing 20–24
OSF/Motif 5
TopLevelShell
and dialogs 204, 227, 228, 230
creating 204
versus DialogShells 152
transient dialogs
definition 128
TransientShell 152
translation tables 14, 32
and BulletinBoards 241
and tab groups 288
bugs 372
DrawingArea 363, 371
gadgets 290
manager widget 288
MenuShell 73
translations
and drag sources 702
and text widgets 527
overriding 726
traversal
processing manually 290
type conversion 202

U

unit length
scrollbars 331
UNIX signals 887
unlocking clipboards 681
unmanaging dialogs 140–142
uppercase
converting text to 568
user interface 6–9

V

values
scrollbar 331
varargs 23
and dialogs 135
VendorShell 152, 585, 639
resources 647–650
vertical scrollbars 50
and scrolled lists 438
vertically tiled format
and PanedWindows 275
vi
and Text widgets 527
view length
scrollbars 332
view_height
and scrolled windows 353
view_width
and scrolled windows 353
virtual bindings 33
virtual keysyms 33

W

WarningDialog 133
wide-character strings 580, 804
widgets
creating 12, 24–25
definition 12
drag and drop 699
event handling 32–39
instantiating 12
manager 25, 53–65
(see also manager widgets)
naming 31
overview 41–96
parent-child relationships 54
protocol 654
realizing 39
redrawing 427
window manager 637–672
and dialogs 141
and ICCCM 73
and shell widgets 70
borders 648
bugs 646
bypassing 72
close item 651

decorations 648
menu functions 650
protocols 71, 650–658
resize handles 648
resources
interactivePlacement 230
setting the title string 146
standard decorations 71
windows
border color 83
bypassing the window manager 72
clearing, in DrawingAreas 364
copying text between 673
drawing into 883
enlargening 649
iconifying 649
monitoring, example 884
moving text between 673
properties (see properties)
reparenting 145
scrolled
creating 323
secondary 74
sizing and placement conventions 638
titlebars 649
WM_DELETE_WINDOW atom 651
WM_PROTOCOLS protocol 651
WM_SAVE_YOURSELF protocol 655
WMShell 639
work areas
and dialogs 78, 131
in main windows 67
work procedures 871–876
and WorkDialog 873
in Xt 846
WorkDialog
and work procedures 873
example 877
WorkingDialog 133, 870–887
wprint() 548
WWM_SAVE_YOURSELF atom 651
WWM_TAKE_FOCUS atom 651

X

X event control loop 844
X Input Context 584
X Input Method 584
X toolkit
and Motif 608
basic terminology 11
X Toolkit Intrinsics
selection mechanisms 700
X11/cursorfont.h 881
X11R5 19
XAllocNamedColor() 118
XAPPLRESDIR 82
XBMLANGPATH 82
XChangeProperty() 670
XChangeWindowAttributes() 881
XCheckMaskEvent() 882
XClearWindow()
example 343
XClientMessageEvent() 670
xclipboard 695
XCopyArea() 355
example 343, 373
XCopyPlane() 355
XCreateFontCursor() 870, 881
XCreateGC()
example 373
XCreateIC() 584
XCreatePixmapFromBitmapData() 217
example 209
XCreateSimpleWindow() 646
XDrawLine()
example 343, 373
XDrawString() 85
example 343
XFillRectangle()
example 343, 373
XFlush() 881
XFontSet 582, 804
XFreePixmap()
versus XmDestroyPixmap 892
XGetGeometry() 646
XGetImage() 82
XGetSelectionOwner() 693
XGetWindowProperty() 670
XIC 584
XIM 584
Ximp input method 585
XInternAtom() 694
XKeysymDB 34
Xlib 17
drawing 367
XLoadQueryFont()

example 396
Xm library 14–16, 17
Xm/ArrowB.h 421
Xm/ArrowBG.h 421
Xm/BulletinB.h 238
Xm/Command.h 134
Xm/DialogS.h
and identifying DialogShells 154
Xm/DrawingA.h 362
Xm/DrawnB.h 427
Xm/FileSB.h 134, 183
Xm/Frame.h 272
Xm/Label.h 389
Xm/LabelG.h 389
Xm/List.h 434
Xm/MainW.h 98
Xm/MessageB.h 133
Xm/MwmUtil.h 649
Xm/PanedW.h 276
Xm/SashP.h 283
Xm/SelectioB.h 134
Xm/Text.h 530
Xm/TextF.h 530
Xm/ToggleB.h 406
Xm/ToggleBG.h 407
XmActivateWMProtocol() 669
XmADDITION 459
XmAddProtocolCallback()
example 667
XmAddProtocols()
example 667
XmAddTabGroup() 288
XmAddWMProtocolCallback() 653
example 651, 656
XmAddWMProtocols() 655
example 656
XmALIGNMENT_BASELINE_BOTTOM 268, 273, 395
XmALIGNMENT_BASELINE_TOP 268, 273, 395
XmALIGNMENT_BEGINNING 268, 273, 394, 840
XmALIGNMENT_CENTER 268, 273, 394, 840
XmALIGNMENT_CONTENTS_BOTTOM 268, 395
XmALIGNMENT_CONTENTS_TOP 268, 395
XmALIGNMENT_END 268, 273, 394, 840
XmALIGNMENT_WIDGET_BOTTOM 273
XmALIGNMENT_WIDGET_TOP 273
XmAnyCallbackStruct 38, 108, 149, 577
XmAPPLICATION_DEFINED 323, 325
XMapRaised() 868
XmARROW_DOWN 423
XmARROW_LEFT 423
XmARROW_RIGHT 423
XmARROW_UP 423
XmArrowButtonCallbackStruct 423
XmAS_NEEDED 101, 324, 438
XmATTACH_FORM 245
XmATTACH_NONE 245, 247
XmATTACH_OPPOSITE_FORM 245
XmATTACH_OPPOSITE_WIDGET 245, 246
XmATTACH_POSITION 245, 247
XmATTACH_SELF 245, 248
XmATTACH_WIDGET 245, 246
XmAUTOMATIC 101, 323
xmbind 35
XmbLookupString() 585
XmBOTTOM_LEFT 336
XmBOTTOM_RIGHT 336
XmBROWSE_SELECT 436, 455
XmCascadeButtonGadgetClass 606
XmCascadeButtonWidgetClass 606
XmChangeColor() 84
XmClipboardBeginCopy() 684
example 684
XmClipboardCancelCopy() 688
XmClipboardCopy() 677, 679
example 675, 684, 690
XmClipboardCopyByName() 688
and incremental copying 688
example 684
XmClipboardEndCopy() 678, 679
example 675, 684, 690
XmClipboardEndRetrieve() 680
example 681, 684
XmClipboardInquireCount() 691
XmClipboardInquireFormat() 691
XmClipboardInquireLength() 682
example 683
XmClipboardLock() 688
XmClipboardRegisterFormat() 689
example 690
XmClipboardRetrieve()

example 675, 681, 683, 684
XmClipboardStartCopy() 677, 678
example 675, 690
XmClipboardStartRetrieve() 680
example 681, 684
XmClipboardUndoCopy() 679
XmClipboardUnlock() 688
XmClipboardWithdrawFormat() 680, 688, 691
XmCommandAppendValue() 181
XmCommandCallbackStruct 181
XmCommandError() 181
XmCommandGetChild() 181
XmCommandSetValue() 181
XmCommandWidgetClass 180
XmCONSTANT 324, 438
XmCR_ACTIVATE 109, 269, 367
and DrawnButton 429
and PushButton 402
XmCR_APPLY 176, 186
XmCR_ARM 426
and DrawnButton 429
and PushButton 402
XmCR_BROWSE_SELECT 457
XmCR_CANCEL 149, 176, 186
XmCR_CLIPBOARD_DATA_DELETE 687
XmCR_CLIPBOARD_DATA_REQUEST 687
XmCR_COMMAND_CHANGED 181
XmCR_COMMAND_ENTERED 181
XmCR_DECREMENT 339
XmCR_DEFAULT_ACTION 457
XmCR_DISARM 426
and DrawnButton 429
and PushButton 402
XmCR_DRAG 339, 503
XmCR_EXPOSE 367, 429
XmCR_EXTENDED_SELECT 457
XmCR_HELP 149, 176, 186
XmCR_INPUT 366, 367
XmCR_MODIFYING_TEXT_VALUE 569, 582
XmCR_MOVING_INSERT_CURSOR 578
XmCR_MULTIPLE_SELECT 457
XmCR_NO_MATCH 176, 186
XmCR_OK 149, 176, 186
XmCR_PAGE_DECREMENT 339
XmCR_PROTOCOLS 653
XmCR_RESIZE 367, 429
XmCR_SINGLE_SELECT 457
XmCR_TO_BOTTOM 339
XmCR_TO_TOP 339
XmCR_VALUE_CHANGED 109, 339, 412, 503, 577
XmCreateCommand() 134, 170, 180
XmCreateDialogShell() 204, 215
XmCreateDragIcon() 707, 718, 727
XmCreateErrorDialog() 134
XmCreateFileSelectionBox() 134, 170, 183
XmCreateFileSelectionDialog() 117, 134, 170, 183
example 184, 536, 557, 612
XmCreateInformationDialog() 134
example 862, 865
XmCreateMenuBar() 603
example 604
XmCreateMessageBox() 134
XmCreateMessageDialog() 134, 891
example 133, 170–171, 862, 887
XmCreateOptionMenu() 631
example 631
XmCreatePopupMenu() 625
example 623
XmCreatePromptDialog() 134, 169
example 177
XmCreatePulldownMenu() 604
example 610, 623
XmCreatePushButton() 24
XmCreatePushButtonGadget() 24
XmCreateQuestionDialog() 134
example 136, 162
XmCreateRadioBox() 268, 406, 414
example 414
XmCreateScrolledList() 437
and Form widgets 260
example 102, 455, 911
XmCreateScrolledList(^^)
example 396
XmCreateScrolledText() 122, 218, 527, 535
and Form widgets 260
example 209, 333
XmCreateSelectionBox() 134, 169
XmCreateSelectionDialog() 134, 169
XmCreateSimpleCheckBox() 406, 418
XmCreateSimplePulldownMenu()
example 557
XmCreateTemplateDialog() 134
XmCreateWarningDialog() 134

XmCreateWarningDialog(^)
example 144
XmCreateWorkingDialog() 134, 875
example 873
XmDeactivateWMProtocol() 669
XmDESTROY 141, 891
XmDestroyPixmap() 80, 118
versus XFreePixmap 892
XmDIALOG_CANCEL_BUTTON 143
XmDIALOG_COMMAND 170
XmDIALOG_ERROR 152
XmDIALOG_FILE_SELECTION 170
XmDIALOG_HELP_BUTTON 143
XmDIALOG_INFORMATION 152
XmDIALOG_MESSAGE 152
XmDIALOG_NONE 143
XmDIALOG_OK_BUTTON 143
XmDIALOG_PROMPT 170
XmDIALOG_SELECTION 170
XmDIALOG_TEMPLATE 152
XmDIALOG_WARNING 152
XmDIALOG_WORK_AREA 170
XmDIALOG_WORKING 152
XmDialogShellWidgetClass 151
XmDO_NOTHING 141
XmDRAG_DROP_ONLY style 715
XmDragCancel() 735
XmDragDropFinishCallbackStruct 736
XmDragProcCallbackStruct 754
XmDragStart() 702, 711, 727, 729, 730, 731
XmDrawingAreaCallbackStruct 366
XmDrawnButtonCallbackStruct 429
XmDropProcCallbackStruct 746, 753
XmDropSiteConfigureStackingOrder() 706
XmDropSiteEndUpdate() 746
XmDropSiteEnterCallbackStruct 734
XmDropSiteQueryStackingOrder() 706
XmDropSiteRegister() 704
for registering Label 745
XmDropSiteRetrieve() 704, 735, 745
XmDropSiteStartUpdate() 746
XmDropSiteUnregister() 716, 717
XmDropSiteUpdate() 704, 716, 717, 746
XmDropStartCallbackStruct 735
XmDropTransferAdd() 705, 748, 749
XmDropTransferEntryRec 747, 748
XmDropTransferStart() 705, 709, 712, 747, 753
xmemo.c program 911
XmEXCLUSIVE_TAB_GROUP 288
XmEXTENDED_SELECT 436
XmEXTENDED_SELECT selection policy 458
XmFILE_ANY_TYPE file type mask 192
XmFILE_DIRECTORY file type mask 192
XmFILE_REGULAR file type mask 192
XmFileSelectionBoxCallbackStruct 186
XmFileSelectionBoxWidgetClass 183
XmFileSelectionDoSearch() 186
XmFONTLIST_DEFAULT_TAG 583, 806, 812
XmFontListAppendEntry()
example 396
XmFontListEntryCreate()
example 396
XmFontListFree()
example 396
XmFRAME_TITLE_CHILD 272
XmFRAME_WORKAREA_CHILD 272
XmGetDragContext() 733
XmGetFocusWidget() 293
XmGetPixmap() 118, 393, 411, 726
and XCreatePixmapFromBitmapData() 217
example 409
XmGetPixmap(^^)
example 427
XmGetPixmapByDepth() 727
XmGetTabGroup() 293
XmGetTearOffControl() 614
XmGetXmDisplay() 713, 715
XmGetXmScreen() 717, 718
XmHORIZONTAL 263, 266, 336, 501
XmINITIAL 459
XmInstallImage() 80, 81, 82
XmInternAtom() 653, 694, 726, 745
example 651, 667
versus XInternAtom() 653
XmIsDialogShell macro 154
XmIsMessageBox macro 155
XmIsTraversable() 293
xmLabelGadgetClass pointer 389
XmListAddItem() 439
XmListAddItems() 439
XmListAddItemsUnselected() 439
XmListAddItemUnselected() 439
example 440, 447
XmListDeleteAllItems() 444

XmListDeleteItem() 444
XmListDeleteItems() 445
XmListDeleteItemsPos() 444, 445
XmListDeletePos() 444
XmListDeletePositions() 445
XmListDeselectAllItems() 446
XmListDeselectItem() 446
XmListDeselectPos() 446
XmListGetKbdItemPos() 453
XmListGetMatchPos() 443
XmListGetSelectedPos() 446
XmListItemExists() 442
XmListItemPos() 443
XmListPosSelected() 446
XmListPosToBounds() 453
XmListReplaceItems() 444
XmListReplaceItemsPos() 443
XmListReplaceItemsPosUnselected() 443
XmListReplaceItemsUnselected() 444
XmListReplacePositions() 444
XmListSelectPos()
    example 911
XmListSetBottomItem() 452
XmListSetBottomPos() 452
    example 452
XmListSetItem() 452
XmListSetKbdItemPos() 453
XmListSetPos() 452
    example 452
XmListWidgetClass 434
XmListYToPos() 453
XmMainWindowSetAreas() 122
    example 536
XmMAX_ON_BOTTOM 336, 501
XmMAX_ON_LEFT 336, 501
XmMAX_ON_RIGHT 336, 501
XmMAX_ON_TOP 336, 501
XmMENU_BAR 122, 262
XmMENU_POPUP 262
XmMenuPosition()
    example 593, 626
XmMessageBoxGetChild() 198
    example 162, 862, 865, 877, 887
XmMODIFICATION 459
XmMULTI_LINE_EDIT 534
XmMULTICLICK_DISCARD 405
XmMULTICLICK_KEEP 406
XmMULTIPLE_SELECT 436, 445, 457
XmNaccelerator 608
XmNacceleratorText 608, 609
XmNactivateCallback
    and DrawnButton 429
    and nonstandard buttons 197
    and PushButton 367, 400
    and Text widgets 565
    and TextField widgets 565
    automation of 567
    example 565
    reasons 367
XmNalignment 268, 394
XmNallowShellResize 644
XmNanimationMask
    for DropSite 754
XmNanimationPixmap
    for DropSite 754
XmNanimationStyle 705, 754
    for DropSite 753
XmNapplyCallback 135, 177
XmNapplyLabelString 135
XmNarmCallback 400
XmNarmColor 83
XmNarrowDirection 423
XmNattachment 707
XmNaudibleWarning 571
XmNautoShowCursorPosition 546
XmNautoUnmanage 140, 179
XmNbackground 83
XmNbackgroundPixmap 431
XmNbaseHeight 641
XmNbaseWidth 641
XmNblendModel 707, 728
XmNborderColor 83
XmNbottomAttachment 245
XmNbottomOffset 250
XmNbottomPosition 247
XmNbottomShadowColor 83
XmNbottomWidget 246
XmNbrowseSelectionCallback 454, 455
XmNbuttonFontList 147, 238, 813
XmNcancelCallback 135
XmNcancelLabelString 135
XmNchildHorizontalAlignment 273
XmNchildHorizontalSpacing 273
XmNchildPlacement 198
XmNchildType 272
XmNchildVerticalAlignment 273
XmNclientData 729
XmNclipWindow 325

XmNcolormap 384
XmNcolumns 540
XmNcommandWindow 123
XmNconvertProc 703, 704, 709, 711, 712, 729, 730, 733, 748
XmNcursorPositionVisible 540
XmNdecimalPoints 500
XmNdecrementCallback 339
XmNdefaultActionCallback 454
XmNdefaultButton 144, 155, 227
XmNdefaultButtonShadowThickness 219
XmNdefaultButtonType 143
XmNdefaultCopyCursorIcon 717
XmNdefaultFontList 813
XmNdefaultInvalidCursorIcon 717
XmNdefaultLinkCursorIcon 717
XmNdefaultMoveCursorIcon 717
XmNdefaultNoneCursorIcon 717
XmNdefaultPosition 153
XmNdefaultSourceCursorIcon 717, 718
XmNdefaultValidCursorIcon 717
XmNdefaultVirtualBindings 35
XmNdeleteResponse 141, 891
XmNdepth 707
XmNdestroyCallback 613
  and destroying dialogs 887
  and Text widgets 536
  callback routine 152
XmNdialogStyle 159
XmNdialogTitle 146
XmNdialogType 151, 170
  in Motif 1.2 198
XmNdirectoryValid 191
XmNdirListItemCount 191
XmNdirListItems 191
XmNdirListLabelString 135
XmNdirSearchProc 191, 192
XmNdirSpec 187
XmNdisarmCallback 400
XmNdoubleClickInterval 455
XmNdragCallback 503
  and ScrollBars 339
XmNdragDropFinishCallback 712, 729, 736
XmNdragInitiatorProtocolStyle 709, 713, 714, 715, 716
XmNdragMotionCallback 734
XmNdragOperations 702, 728
  for DragContext 754
XmNdragProc 705, 708, 709, 711, 715, 754
XmNdragReceiverProtocolStyle 709, 713, 715, 716
XmNdropFinishCallback 712, 736
XmNdropProc 711, 712, 746, 748
  for DropSite 735, 745, 746, 749
  for Text drop site 745
XmNdropRectangles 706
XmNdropSiteActivity 716, 717
XmNdropSiteEnterCallback 734
XmNdropSiteLeaveCallback 734
XmNdropSiteOperations 704, 745, 754
XmNdropSiteType 706
XmNdropStartCallback 712
  of DragContext 735
XmNdropTransfers 705, 747
XmNeditable 218, 540, 548, 717
XmNeditMode 534
XmNentryAlignment 267, 395
XmNentryCallback 269
  and menus 607
  example 270
XmNentryClass 268
XmNentryVerticalAlignment 268, 395
XmNexportTargets 703, 728, 733
  for DragContext 748
  for DragSource 754
XmNexposeCallback
  and DrawingArea 351
  and DrawnButton 429
  and Expose events 363
  and expose_resize() 353
  and Scrollbars 377
  Eventtypes 367
XmNextendedSelectionCallback 454, 458
XmNfileListItemCount 187
XmNfileListItems 187
XmNfileListLabelString 135
XmNfileSearchProc 187
XmNfilterLabelString 135
XmNfocusCallback 237, 579
XmNfontList 396, 583, 809, 819
XmNforeground 83
XmNfractionBase 226, 247, 253
XmNheight 230, 707
XmNheightInc 641
XmNhelpCallback 135, 389, 864
XmNhelpLabelString 135, 167
XmNhightlightColor 83

XmNhorizontalScrollBar 123, 325
XmNhorizontalSpacing 257
XmNhotX 707
XmNhotY 707
XmNiconic 646
XmNiconName 647
XmNiconPixmap 645
XmNiconWindow 646
XmNiconX 647
XmNiconY 647
XmNimportTargets 704
for DropSite 745, 754
for Text drop site 745, 746
XmNincrement 332, 335
XmNincremental 703, 709
XmNincrementCallback 339
XmNindicatorOn 411
XmNindicatorSize 409
XmNindicatorType 407, 408, 414
XmNinitialFocus 145
XmNinputCallback 363, 366
Eventtypes 367
XmCR_INPUT as reason 366
XmNinputMethod 585
XmNinvalidCursorForeground 718
of DragContext 735
XmNisAligned 267
XmNisHomogeneous 268, 414
XmNitemCount 436
XmNitems 436
XmNlabelFontList 147, 239, 813
XmNlabelInsensitivePixmap 394, 430
XmNlabelPixmap 217, 391, 430, 726
XmNlabelString 390, 748
XmNlabelType 217, 390, 391, 409
XmPIXMAP as value 217
XmNleftAttachment 245
XmNleftOffset 250
XmNleftPosition 247
XmNleftWidget 246
XmNlistItemCount 174
XmNlistItems 174
XmNlistLabelString 135, 174
XmNlistSizePolicy 438
XmNlistUpdated 187, 191
XmNlistVisibleItemCount 174
XmNlosingFocusCallback 579, 580
XmNmainWindowMarginHeight 124
XmNmainWindowMarginWidth 124
XmNmapCallback 229, 237
and positioning dialogs 229
XmNmask 707, 727
XmNmaxHeight 641
XmNmaximum 333, 335, 500
XmNmaxLength 533
XmNmaxWidth 641
XmNmenuBar 103, 123
XmNmenuHelpWidget 115, 610, 618
XmNmenuPost 631
XmNmessageString 135, 198
XmNmessageWindow 123
XmNminHeight 641
XmNminimizeButtons 145
XmNminimum 333, 335, 500
XmNminWidth 641
XmNmnemonic 607
XmNmodifyVerifyCallback 568, 582
XmNmodifyVerifyCallbackWcs 582
XmNmotionVerifyCallback 578
XmNmultiClick 405
XmNmultipleSelectionCallback 454, 457
XmNmustMatch 175
XmNmwmDecorations 648
XmNmwmFunctions 650
XmNmwmMenu 669
XmNnavigationType 287
XmNnoMatchCallback 175
XmNnoneCursorForeground
of DragContext 735
XmNnoResize 146
XmNnumColumns 265, 393
XmNnumDropRectangles 706
XmNnumDropTransfers 705, 747
XmNnumExportTargets 703, 728, 733
XmNnumImportTargets 704
for Text drop site 745
XmNoffsetX 707
XmNoffsetY 707
XmNokCallback 135, 175
XmNokLabelString 135
XmNONE 287
XmNoperationChangedCallback 734
XmNoperationCursorIcon 703, 717, 728
of DragContext 735
XmNorientation 336, 501
and RowColumns 263, 265, 266
XmNpacking 265
XmNpageDecrementCallback 339

XmNpageIncrement 333
XmNpageIncrementCallback 339
XmNpaneMaximum 227, 278
XmNpaneMinimum 227, 278
XmNpixmap 707, 727
XmNpopdownCallback 141, 231, 613
XmNpopupCallback 141, 231
XmNpositionIndex 276
XmNpreeditType 585
XmNprocessingDirection 336, 501
XmNpromptString 135
XmNpushButtonEnabled 430
XmNqualifySearchProc 192
XmNradioAlwaysOne 414
    used in menus 115
XmNradioBehavior
    and CheckBox 418
    and RadioBox 414
    and RowColumn 408
    used in menus 115
XmNresizable 257
XmNresizeCallback 363
    and DrawingAreas 351
    and DrawnButton 429
    and expose_resize() 353
    Eventtypes 367
XmNresizeHeight 542
XmNresizePolicy 439
XmNresizeWidth 542
XmNrightAttachment 245
XmNrightOffset 250
XmNrightPosition 247
XmNrightWidget 246
XmNrowColumnType 122, 262
XmNrows 540
XmNrubberPositioning 257
XmNsashHeight 216
XmNsashWidth 216
XmNscaleMultiple 502
XmNscrollBarDisplayPolicy 101, 323, 325, 438
XmNscrollBarPlacement 336
XmNscrollHorizontal 541
XmNscrollingPolicy 101, 323, 325, 377
XmNscrollLeftSide 541
XmNscrollTopSide 541
XmNscrollVertical 541
XmNselectColor 83
XmNselectedItemCount 445
XmNselectedItems 445
XmNselectInsensitivePixmap 411
XmNselectionArray 555
XmNselectionArrayCount 555
XmNselectionLabelString 135
XmNselectionPolicy 436, 445, 455, 457
XmNselectPixmap 409
XmNselectThreshold 556
XmNsensitive 205
    and deactivating menus 606
    and pixmaps 411
XmNset 409
XmNshadowThickness 238
XmNshadowType 238, 272, 430
XmNshowAsDefault 219
XmNshowSeparator 124
XmNshowValue 500, 501
XmNsingleSelectionCallback 454, 455
XmNskipAdjust 216
XmNsliderSize 333
XmNsource 587
XmNsourceCursorIcon 703, 717, 728, 735
    of DragContext 735
XmNsourcePixmapIcon 735
    of DragContext 735
XmNstateCursorIcon 703, 717
    of DragContext 735
XmNstringDirection 394, 541, 808, 814
XmNsubMenuId 605
XmNsymbolPixmap 135, 198, 887
    example 887
XmNtearOffMenuActivateCallback 614
XmNtearOffMenuDeactivateCallback 614
XmNtearOffModel 613
XmNtextFontList 147, 239, 813
XmNtextString 179
XmNtitle 146, 215, 647
XmNtitleString 500
XmNtoBottomCallback 339
XmNtopAttachment 245
XmNtopItemPosition 452
XmNtopLevelEnterCallback 734
XmNtopLevelLeaveCallback 734
XmNtopOffset 250
XmNtopPosition 247
XmNtopShadowColor 83
XmNtopWidget 246
XmNtoTopCallback 339
XmNtransferProc 705, 709, 712

for DropTransfer 748
of DropSite 736
XmNtransferStatus 705, 753
XmNtranslations 373, 702, 716, 726, 731
XmNtraversalOn 285, 439
XmNtraverseObscuredCallback 356
example 356
XmNtroughColor 83
XmNuserData 202, 367, 726, 731
for Label 729
XmNvalidCursorForeground 718
of DragContext 735
XmNvalue
and Scale 500
and ScrollBars 333
and Text widget 218, 531, 581
and TextField widget 531, 581
XmNvalueChangedCallback 339, 412, 414, 503, 568, 577, 623
XmNvalueWcs 581
XmNverifyBell 571
XmNverticalScrollBar 123
XmNverticalSpacing 257
XmNvisibleItemCount 261, 437, 438
XmNvisibleWhenOff 414
XmNvisualPolicy 323
XmNwidth 230, 707
XmNwidthInc 641
XmNwordWrap 541
XmNworkWindow 123, 325
XmNx 230
XmNy 230
XmONE_OF_MANY 414
XmPACK_COLUMN 265
XmPIXMAP 217, 390, 391, 409
XmPLACE_ABOVE_SELECTION 199
XmPLACE_BELOW_SELECTION 199
XmPLACE_TOP 199
XmProcessTraversal() 292, 567, 579
XmPULLDOWN 262
XmPushButtonCallbackStruct 38, 108, 401
XmRegisterSegmentEncoding() 812
XmRESIZE_IF_POSSIBLE 438
XmRESIZE_NONE 439
XmRowColumn 261–271
XmRowColumnCallbackStruct 269
XmScaleCallbackStruct 503
XmScaleGetValue() 501
XmScaleSetValue() 501
XmScrollBarCallbackStruct 338
XmScrollBarGetValues() 355
XmScrollBarSetValues() 355
XmScrolledWindowSetAreas() 352
example 343
XmScrollVisible() 358
XmSELECT_ALL 556
XmSELECT_LINE 556
XmSELECT_PARAGRAPH 556
XmSELECT_POSITION 556
XmSELECT_WHITESPACE 556
XmSELECT_WORD 556
XmSelectionBoxCallbackStruct 176
XmSelectionBoxGetChild() 176, 179
example 170, 177, 202
XmSelectItem() 445
XmSelectPos() 445
XmSHADOW_ETCHED_IN 238, 272, 430
XmSHADOW_ETCHED_OUT 238, 272, 430
XmSHADOW_IN 238, 272, 430
XmSHADOW_OUT 238, 272, 430
XmSINGLE_LINE_EDIT 534
XmSINGLE_SELECT 436, 455
XmSTATIC 438
value 324
XmSTATIC value 325
XmSTICKY_TAB_GROUP 288
XmSTRING 390
XmString type
example 16
XmSTRING_DIRECTION_L_TO_R 815, 840
XmSTRING_DIRECTION_R_TO_L 815, 840
XmStringBaseline() 841
XmStringByteCompare() 442
XmStringConcat() 817
XmStringCreate() 808, 817
example 396
XmStringCreateLocalized() 805, 864
example 16, 133, 136, 160, 416, 447, 536, 551, 557, 593, 911
XmStringCreateLocalized(^^)
example 403
XmStringCreateLtoR() 815, 864
example 144, 170–171, 396
XmStringCreateSimple()
example 209

XmStringDirection type 815
XmStringDirectionCreate() 808, 815
XmStringDraw() 839
XmStringDrawImage() 839, 840
XmStringDrawUnderline() 839, 840
XmStringExtent() 841
XmStringFree() 806
    example 16, 210, 396, 440, 447, 551, 557
XmStringFree(^^)
    example 403
XmStringFreeContext() 822
    example 823
XmStringGetLtoR() 176, 186, 457
    example 170–171, 455, 457, 536
XmStringGetNextSegment() 822
    example 823
XmStringHeight() 841
XmStringInitContext() 821
    example 823
XmStringLineCount() 817
XmStringSegmentCreate() 814
XmStringTable
    example 396
XmStringWidth() 841
XmTAB_GROUP 287
XmTEAR_OFF_DISABLED 614
XmTEAR_OFF_ENABLED 614
XmTEXT_BACKWARD 546
XmTEXT_FORWARD 546
XmTextBlock 570
XmTextBlockWcs 582
XmTextClearSelection() 555, 693
XmTextCopy() 554, 692
XmTextCut() 554, 692
XmTextDisableRedisplay() 532
XmTextEnableRedisplay() 532
XmTextField
    used in selection box 169
XmTextFieldGetString() 533
XmTextFieldSetCursorPosition()
    example 536
XmTextFieldSetString() 532
    example 440, 447, 536
XmTextFindString() 545
XmTextFindStringWcs() 582
XmTextGetCursorPosition() 542
    example 542, 557
XmTextGetInsertionPosition() 542
XmTextGetLastPosition() 546, 572
    example 557
XmTextGetSelection() 555, 693
XmTextGetSelectionPosition() 555
XmTextGetSelectionWcs() 582
XmTextGetString() 532
    example 547, 557
XmTextGetStringWcs() 582
XmTextGetSubstring() 533
XmTextGetSubstringWcs() 582
XmTextHighlight() 557
XmTextInsert() 550
    example 549
XmTextInsertWcs() 582
XmTextPaste() 692
XmTextPosition type 542
XmTextReplace() 546
    example 547, 557
XmTextReplaceWcs() 582
XmTextScanType type 556
XmTextScroll() 546
XmTextSetCursorPosition() 542
XmTextSetHighlight()
    example 557
XmTextSetInsertionPosition() 542, 577
    example 542
XmTextSetSelection() 693
XmTextSetString() 181, 532, 748
    example 536, 542, 547, 551
XmTextSetStringWcs() 582
XmTextShowPosition() 546, 550
    example 549
XmTextVerifyCallbackStruct 569
XmTextVerifyCallbackStructWcs 582
XmToggleButtonCallbackStruct 412
XmToggleButtonGadgetGetState() 412
XmToggleButtonGadgetSetState() 413
XmToggleButtonGetState() 412
XmToggleButtonSetState() 413
XmTOP_LEFT 336
XmTOP_RIGHT 336
XmTrackingEvent() 869
XmTranslateKey() 35
XmTRAVERSE_CURRENT 292
XmTRAVERSE_DOWN 292
XmTRAVERSE_HOME 292
XmTRAVERSE_LEFT 292
XmTRAVERSE_NEXT 292
XmTRAVERSE_NEXT_TAB_GROUP 292,

567
XmTRAVERSE_PREV 292
XmTRAVERSE_PREV_TAB_GROUP 292
XmTRAVERSE_RIGHT 292
XmTRAVERSE_UP 292
XmTraverseObscuredCallbackStruct 357
XmUninstallImage() 82
XmUNMAP 141
XmUNSPECIFIED_PIXMAP 393
XmUpdateDisplay() 883
    example 877, 884
XmVaCASCADEBUTTON 107
XmVaCHECKBUTTON 107
XmVaCreateSimpleCheckBox() 418
XmVaCreateSimpleMenuBar() 104, 592
    example 106, 210, 536, 551
XmVaCreateSimpleOptionMenu() 592
    example 601
XmVaCreateSimplePopupMenu() 592
    example 593, 598
XmVaCreateSimplePulldownMenu() 105, 116, 592
    example 536, 551, 598
XmVaCreateSimpleRadioBox() 416
    example 416
XmVaDOUBLE_SEPARATOR 107
XmVaPUSHBUTTON 106
XmVaRADIOBUTTON 107, 416
XmVARIABLE 438
    value 325
XmVaSEPARATOR 107
XmVaSINGLE_SEPARATOR 107
XmVaTITLE 107
XmVaTOGGLEBUTTON 106
XmVERTICAL 263, 266, 336, 501
XmWORK_AREA 262
XOpenIM() 584
XQueryBestCursor() 715
XQueryTree() 670
XSendEvent() 670
XSetErrorHandler() 550
XSetFont()
    example 343
XSetForeground()
    example 343, 373
XSetWindowBackgroundPixmap() 431
    example 646
xshowbitmap.c program 905
Xsi input method 585
Xt
    programming with 16–39
Xt library 14–16, 17
Xt selection mechanisms 709
XtActionsRec 241
XtAddActions() 241
XtAddCallback() 198, 227
    adding dialog callbacks 148
    example 147, 170–171, 368, 498, 504
XtAddEventHandler() 630
    example 643
XtAppAddActions() 702, 726, 731
XtAppAddInput() 886
XtAppAddTimeOut() 871, 876
    and signal handling 853
    example 887
XtAppAddWorkProc() 847, 875
XtAppMainLoop()
    and signal handling 846
XtAppSetErrorHandler() 550
XtAppSetWarningHandler() 550
XtCallCallbacks() 870
XtConvertSelectionIncrProc 703
XtConvertSelectionProc 703, 729
XtCreateApplicationShell() 204
XtCreatePopupShell() 151, 204
XtDestroyWidget() 220, 736
XtDisplayOfObject() 678
XtGetMultiClickTime() 556
XtGetValues() 733
XtGrabExclusive 137
XtGrabKind 137
XtGrabNone 137
XtGrabNonexclusive 137
XtInitialize() 21
XtIsVendorShell() 639
XtMalloc()
    example 170–171
XtManageChild() 136
    and popping up dialogs 136, 231
XtNew() 891
XtNtitle 215
XtOverrideTranslations() 241, 372
    example 239
XtParent() 137, 535
XtParseTranslationTable() 241, 702, 726, 731
    example 239, 373
XtPopdown() 231

XtPopup() 231
XtQueryGeometry() 841
    and PanedWindows 280
XtRemoveEventHandler()
    example 643
XtRemoveWorkProc() 875
XtResolvePathname() 803
XtSelectionCallbackProc 705, 748
XtSetArg()
    example 211
XtSetLanguageProc() 20, 580, 803
XtSetMultiClickTime() 402, 556
XtSetSensitive() 394, 611
    example 170–171, 612
XtSetValues()
    example 102
XtVaAppCreateShell() 204
XtVaAppInitialize() 21
XtVaCreateApplicationShell() 204
XtVaCreateArgsList() 506
    example 504
XtVaCreateManagedWidget() 149, 406
    and creating Frame widgets 272
    and creating List widgets 434
    versus XtVaCreateWidget() 235
XtVaCreatePopupShell() 204
    example 215, 222
XtVaCreateWidget() 149
    versus XtVaCreateManagedWidget() 235
XtVaGetValues() 27, 532, 704, 715, 746
XtVaSetValues() 27, 704, 715, 717
XtVaTypedArg 84, 102, 202
    and compound strings 806
XtWidgetGeometry type 282
XtWindowOfObject() 678
XwcLookupString() 585

Z

zero-length attachment offsets 252