MOTIF 2.1—Programmer's Guide

Desktop Product Documentation

The Open Group

Desktop Product Documentation:

Motif 2.1—Programmer's Guide

ISBN 1-85912-134-9

Document Number M213

Published in the U.K. by The Open Group, 1997

Any comments relating to the material contained in this document may be submitted to:

The Open Group
Apex Plaza
Forbury Road
Reading
Berkshire, RG1 1AX
United Kingdom

or by Electronic Mail to:
OGPubs@opengroup.org

OTHER NOTICES

Contents

List of Figures

List of Tables

Preface

The Open Group

The Open Group is the leading vendor-neutral, international consortium for buyers and suppliers of technology. Its mission is to cause the development of a viable global information infrastructure that is ubiquitous, trusted, reliable, and as easy-to-use as the telephone. The essential functionality embedded in this infrastructure is what we term the IT DialTone. The Open Group creates an environment where all elements involved in technology development can cooperate to deliver less costly and more flexible IT solutions.

Formed in 1996 by the merger of the X/Open Company Ltd. (founded in 1984) and the Open Software Foundation (founded in 1988), The Open Group is supported by most of the world's largest user organizations, information systems vendors, and software suppliers. By combining the strengths of open systems specifications and a proven branding scheme with collaborative technology development and advanced research, The Open Group is well positioned to meet its new mission, as well as to assist user organizations, vendors, and suppliers in the development and implementation of products supporting the adoption and proliferation of systems which conform to standard specifications.

With more than 200 member companies, The Open Group helps the IT industry to advance technologically while managing the change caused by innovation. It does this by:

- consolidating, prioritizing, and communicating customer requirements to vendors
- conducting research and development with industry, academia, and government agencies to deliver innovation and economy through projects associated with its Research Institute
- managing cost-effective development efforts that accelerate consistent multi-vendor deployment of technology in response to customer requirements
- adopting, integrating, and publishing industry standard specifications that provide an essential set of blueprints for building open information systems and integrating new technology as it becomes available
- licensing and promoting the Open Brand, represented by the ‘‘X’’ mark, that designates vendor products which conform to Open Group Product Standards
- promoting the benefits of open systems to customers, vendors, and the public.

The Open Group operates in all phases of the open systems technology lifecycle including innovation, market adoption, product development, and proliferation. Presently, it focuses on seven strategic areas: open systems application platform development, architecture, distributed systems management, interoperability, distributed computing environment, security, and the information superhighway. The Open Group is also responsible for the management of the UNIX trademark on behalf of the industry.

The Development of Product Standards

This process includes the identification of requirements for open systems and, now, the IT DialTone, development of CAE and Preliminary Specifications through an industry consensus review and adoption procedure (in parallel with formal standards work), and the development of tests and conformance criteria.

This leads to the preparation of a Product Standard which is the name used for the documentation that records the conformance requirements (and other information) to which a vendor may register a product. There are currently two forms of Product

Standard, namely the Profile Definition and the Component Definition, although these will eventually be merged into one.

The ‘‘X’’ mark is used by vendors to demonstrate that their products conform to the relevant Product Standard. By use of the Open Brand they guarantee, through the X/Open Trade Mark Licence Agreement (TMLA), to maintain their products in conformance with the Product Standard so that the product works, will continue to work, and that any problems will be fixed by the vendor.

Open Group Publications

The Open Group publishes a wide range of technical documentation, the main part of which is focused on specification development and product documentation, but which also includes Guides, Snapshots, Technical Studies, Branding and Testing documentation, industry surveys, and business titles.

There are several types of specification:

CAE Specifications

CAE (Common Applications Environment) Specifications are the stable specifications that form the basis for our Product Standards, which are used to develop X/Open branded systems. These specifications are intended to be used widely within the industry for product development and procurement purposes.

Anyone developing products that implement a CAE Specification can enjoy the benefits of a single, widely supported industry standard. Where appropriate, they can demonstrate product compliance through the Open Brand. CAE Specifications are published as soon as they are developed, so enabling vendors to proceed with development of conformant products without delay.

Preliminary Specifications

Preliminary Specifications usually address an emerging area of technology and consequently are not yet supported by multiple sources of stable conformant implementations. They are published for the purpose of validation through implementation of products. A Preliminary Specification is not a draft specification; rather, it is as stable as can be achieved, through applying The Open Group's rigorous development and review procedures.

Preliminary Specifications are analogous to the trial-use standards issued by formal standards organizations, and developers are encouraged to develop products on the basis of them. However, experience through implementation work may result in significant (possibly upwardly incompatible) changes before its progression to becoming a CAE Specification. While the intent is to progress Preliminary Specifications to corresponding CAE Specifications, the ability to do so depends on consensus among Open Group members.

Consortium and Technology Specifications

The Open Group publishes specifications on behalf of industry consortia. For example, it publishes the NMF SPIRIT procurement specifications on behalf of the Network Management Forum. It also publishes Technology Specifications relating to OSF/1, DCE, OSF/Motif, and CDE.

Technology Specifications (formerly AES Specifications) are often candidates for consensus review, and may be adopted as CAE Specifications, in which case the relevant Technology Specification is superseded by a CAE Specification.

In addition, The Open Group publishes:

Product Documentation

This includes product documentation—programmer's guides, user manuals, and so on—relating to the Prestructured Technology Projects (PSTs), such as DCE and CDE. It also includes the Single UNIX Documentation, designed for use as common product documentation for the whole industry.

Guides

These provide information that is useful in the evaluation, procurement, development, or management of open systems, particularly those that relate to the CAE Specifications. The Open Group Guides are advisory, not normative, and should not be referenced for purposes of specifying or claiming conformance to a Product Standard.

Technical Studies

Technical Studies present results of analyses performed on subjects of interest in areas relevant to The Open Group's Technical Program. They are intended to communicate the findings to the outside world so as to stimulate discussion and activity in other bodies and the industry in general.

Versions and Issues of Specifications

As with all live documents, CAE Specifications require revision to align with new developments and associated international standards. To distinguish between revised specifications which are fully backwards compatible and those which are not:

- A new Version indicates there is no change to the definitive information contained in the previous publication of that title, but additions/extensions are included. As such, it replaces the previous publication.
- A new Issue indicates there is substantive change to the definitive information contained in the previous publication of that title, and there may also be additions/extensions. As such, both previous and new documents are maintained as current publications.

Corrigenda

Readers should note that Corrigenda may apply to any publication. Corrigenda information is published on the World-Wide Web at *http://www.opengroup.org/public/pubs*.

Ordering Information

Full catalogue and ordering information on all Open Group publications is available on the World-Wide Web at *http://www.opengroup.org/public/pubs*.

This Book

The *Motif 2.1—Programmer's Guide* describes how to use the Motif application programming interface to create Motif applications. The book gives an overview of the architecture of the Motif widget set, explains features of the Motif toolkit, and presents a model and examples for constructing Motif applications.

Audience

This document is written for designers and developers of Motif programs. It does not provide sufficient information to develop new Motif widgets, although widget developers need to be familiar with the facilities and the issues discussed in this book.

This document assumes that the reader is familiar with the American National Standards Institute (ANSI) C programming language. It also assumes that the reader has a general understanding of the X Window System, the Xlib library, and the X Toolkit Intrinsics (Xt).

Applicability

This is revision 2.1 of this document. It applies to Version 2.1 of the Motif software system.

Purpose

The purpose of this guide is to explain how to write application programs using the Motif toolkit. After reading this book, you should have a general understanding of the Motif toolkit and the Motif widget set and should be able to write applications that use them. This guide is not intended to explain how to develop new classes of widgets.

Organization

This document is organized as follows:

- Chapter 1 introduces the book and gives an overview of the X Window System, Xlib, Xt, and Motif.
- Chapter 2 summarizes the structure of the Motif widget hierarchy and discusses general principles for writing Motif programs.
- Chapter 3 explains the structure and general elements of a Motif application. This chapter also discusses the use of multiple threads in application code.
- Chapter 4 describes the structure of a program that uses the User Interface Language (UIL) and Motif Resource Manager (MRM).
- Chapter 5 discusses most of the primitive widgets that form the building blocks of a Motif application.
- Chapter 6 describes how to use the RowColumn widget to build menus, radio boxes, and check boxes.
- Chapter 7 describes the widgets most appopriate for conducting dialogs with the user.
- Chapter 8 discusses composite widgets commonly used to contain other widgets in the application.
- Chapter 9 explains compound strings (XmString), render tables, tab lists, fonts, and other aspects of rich text.
- Chapter 10 explains the Motif Text and TextField widgets, which provide general display and editing of text.
- Chapter 11 is a guide to internationalizing applications and providing text, font, and other information that is specific to particular language environments.
- Chapter 12 explains how Motif uses colors and pixmaps and how an application can provide its own.
- Chapter 13 discusses issues in handling input, including keyboard focus and traversal, translations, and actions.
- Chapter 14 describes the layout-management policies of particular Motif widgets.
- Chapter 15 describes DrawingArea, a general-purpose widget for displaying graphics and handling user input at a low level.

• Chapter 16 describes how an application can transfer data using the Uniform Transfer Method (UTM).

• Chapter 17 is an extensive discussion of the Motif drag and drop interface, with which the user transfers data by manipulating iconic representations with the pointer.

• Chapter 18 discusses communication between an application and the Motif Window Manager (MWM), by means of resources, selections, protocols, and properties.

• Chapter 19 describes printing from a Motif application.

• Appendix A describes the *XmClipboard* calls.

• Appendix B summarizes data transfer support built into the standard Motif widget set.

Related Documents

For information on Motif and CDE style, refer to the following documents:

CDE 2.1/Motif 2.1—Style Guide and Glossary
Document Number M027 ISBN 1-85912-104-7

CDE 2.1/Motif 2.1—Style Guide Certification Checklist
Document Number M028 ISBN 1-85912-109-8

CDE 2.1/Motif 2.1—Style Guide Reference
Document Number M029 ISBN 1-85912-114-4

For additional information about Motif and CDE, refer to the following Desktop Documentation:

CDE 2.1/Motif 2.1—User's Guide
Document Number M021 ISBN 1-85912-173-X

CDE 2.1—System Manager's Guide
Document Number M022 ISBN 1-85912-178-0

CDE 2.1—Programmer's Overview and Guide
Document Number M023 ISBN 1-85912-183-7

CDE 2.1—Programmer's Reference, Volume 1
Document Number M024A ISBN 1-85912-188-8

CDE 2.1—Programmer's Reference, Volume 2
Document Number M024B ISBN 1-85912-193-4

CDE 2.1—Programmer's Reference, Volume 3
Document Number M024C ISBN 1-85912-174-8

CDE 2.1—Application Developer's Guide
Document Number M026 ISBN 1-85912-198-5

Motif 2.1—Programmer's Reference, Volume 1
Document Number M214A ISBN 1-85912-119-5

Motif 2.1—Programmer's Reference, Volume 2
Document Number M214B ISBN 1-85912-124-1

Motif 2.1—Programmer's Reference, Volume 3
Document Number M214C ISBN 1-85912-164-0

Motif 2.1—Widget Writer's Guide
Document Number M216 ISBN 1-85912-129-2

For additional information about Xlib and Xt, refer to the following X Window System documents:

Xlib—C Language X Interface

X Toolkit Intrinsics—C Language Interface

Typographic and Keying Conventions

This book uses the following conventions.

DocBook SGML Conventions

This book is written in the Structured Generalized Markup Language (SGML) using the DocBook Document Type Definition (DTD). The following table describes the DocBook markup used for various semantic elements.

Markup Appearance	**Semantic Element(s)**	**Example**
AaBbCc123	The names of commands.	Use the **ls** command to list files.
AaBbCc123	The names of command options.	Use **ls –a** to list all files.
AaBbCc123	Command-line placeholder: replace with a real name or value.	To delete a file, type **rm** *filename*.
AaBbCc123	The names of files and directories.	Edit your **.login** file.
AaBbCc123	Book titles, new words or terms, or words to be emphasized.	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.

Terminology Conventions

This book uses the term *primitive* to mean any subclass of **XmPrimitive** and the term *manager* to mean any subclass of **XmManager**. Note that both of these terms are in lowercase.

Keyboard Conventions

Because not all keyboards are the same, it is difficult to specify keys that are correct for every manufacturer's keyboard. To solve this problem, this guide describes keys that use a *virtual key* mechanism. The term *virtual* implies that the keys as described do not necessarily correspond to a fixed set of actual keys. Instead, virtual keys are linked to actual keys by means of *virtual bindings*. A given virtual key may be bound to different physical keys for different keyboards.

See Chapter 13 of this guide for information on the mechanism for binding virtual keys to actual keys. For details, see the **VirtualBindings**(3) reference page in the *Motif 2.1—Programmer's Reference*.

Mouse Conventions

This guide assumes a 3-button mouse. On a 3-button mouse, the leftmost mouse button is usually defined as Btn1, the middle mouse button is usually defined as Btn2, and the rightmost mouse button is usually defined as Btn3. For details about how mouse buttons are usually defined, see the **VirtualBindings**(3) reference page in the *Motif 2.1—Programmer's Reference*.

Problem Reporting

If you have any problems with the software or vendor-supplied documentation, contact your software vendor's customer service department. Comments relating to this Open Group document, however, should be sent to the addresses provided on the copyright page.

Trademarks

Motif® OSF/1®, and UNIX® are registered trademarks and the IT DialTone™, The Open Group™, and the ''X Device''™ are trademarks of The Open Group.

AIX is a trademark of International Business Machines Corp.

HP/UX is a trademark of Hewlett Packard Company.

Solaris is a trademark of Sun Microsystems, Inc.

UnixWare is a trademark of Novell, Inc.

Microsoft Windows is a trademark of Microsoft.

Chapter 1
Introduction

Motif is a **graphical user interface**, a means by which an application program can obtain input from and display output to a user of the application. Motif provides the intermediary mechanisms for communication between the application and the user. To both sides, these mechanisms appear as a set of objects with graphical representations on the screen. The program creates and displays objects of a variety of types provided by Motif for showing the user particular kinds of output and requesting particular kinds of input. The user supplies input by manipulating the screen representations of these objects with the pointer, the keyboard, or both.

This book explains the Motif **application programming interface**. This is the set of facilities that Motif gives an application developer to create and interact with a Motif interface for the application. This book is not a reference work; that is, it does not attempt to describe the API in exhaustive detail. Its focus is on giving an overview of the Motif architecture, explaining the concepts and conventions required to use the API, and providing examples. This book complements other volumes in the Motif documentation set:

- The *Motif 2.1—Programmer’s Reference* describes each element of the Motif programming interface in detail. It is organized into reference pages, one for each element of the interface.
- The *Application Environment Specification — User Environment Volume* describes which elements of the interface an application should use for maximum portability. All implementations of Motif must support the interfaces described in the *AES*.
- The *CDE 2.1/Motif 2.1—Style Guide and Glossary* describes *how* an application should use the interface for maximum consistency with other Motif applications.
- The *CDE 2.1/Motif 2.1—User’s Guide* describes the appearance and interaction style of Motif from the user’s point of view.

The Motif API as supplied by OSF® is implemented in the C programming language. Motif requires that an application written in C conform to American National Standards Institute (ANSI) C. This book assumes knowledge of ANSI C, which is explained by other published reference and tutorial books. It is also possible to write applications in other languages, including C++, but this book gives explanations and examples only for applications written in C.

1.1 The X Window System

Motif is based on the X Window System, often abbreviated as X. The X Window System is fundamentally a protocol by which an application can generate output on a computer that has a bitmapped display and can receive input from devices associated with the display.

X is based on a **client-server** computing model. The application program is the client, communicating through the X protocol with a server that handles the direct output to and input from the display. This model has several important features:

- The client and server may be running on the same machine or on different machines, communicating over a network.
- Only the server need concern itself with the display hardware. The X protocol is hardware independent, so a client can run without alteration using any kind of display that supports the protocol.
- A server may handle multiple clients on the same display at the same time. These clients may communicate with each other, using the server to transfer information.

- A client may communicate with multiple servers.

A *display* is an abstraction that represents the input and output devices controlled by a single server. Usually a display consists of a keyboard, a pointing device, and one or more *screens*. A screen is an abstraction that represents a single bitmapped output device.

Each client creates one or more *windows* on one or more screens of a given display. A window is a rectangular area of the screen on which the client displays output. Windows are arranged in hierarchies of children and parents. The server maintains a tree of windows for each screen. The top-level window is the **root window** of the screen. Each client typically creates at least one window as a child of the root window, and any other client windows are descendants of these top-level client windows. Windows may overlap, and the server maintains a stacking order for all windows on a screen. A child window may extend beyond the boundaries of its parent, but output is *clipped* or suppressed outside the parent's borders.

A client asks the server to create and destroy windows, but the windows themselves are resources controlled by the server. The server maintains other resources, including the following:

- A *pixmap* is a rectangular off-screen area into which an application can draw output. Both windows and pixmaps are *drawables* or entities on which a client can display output. The units of height and width in windows and pixmaps are *pixels*. Each pixel has a given depth, represented as a number of bits or *planes*. Thus, each pixel has an integral value whose range depends on the depth of the drawable. A one-bit-deep pixmap is called a *bitmap*. Each pixel in a bitmap has two possible values: 0 and 1.
- A *colormap* is an association between pixel values and colors. Each color is represented by a triple of red, green, and blue values that result in a particular color on a particular screen. Each window has an associated colormap that determines what color is used to display each pixel.
- A *font* is a collection of glyphs usually used to display text.
- A *cursor* is an object containing information needed for a graphical representation of the position of the pointer. It consists of a source bitmap, a shape bitmap, a *hotspot* or location representing the actual pointer position, and two colors.
- A **graphics context** or GC is a collection of attributes that determine how any given graphics operation affects a drawable. Each graphics operation on a drawable

is executed using a given GC specified by the client. Some attributes of a GC are the foreground pixel, background pixel, line width, and clipping region.

- A *property* is a named data structure associated with a window. Clients often use properties to communicate with each other.

Each client opens a connection to one or more servers. Clients and servers interact by means of *requests*, *replies*, *errors*, and *events*. A client sends a request to the server asking it to take some action, such as creating a window or drawing a line into a pixmap. Some requests, such as requests for information, cause the server to generate replies to the client. A request that results in an error condition may cause the server to generate an error report to the client. The server executes requests from each client in the order in which it receives the requests from that client, although the server may execute requests from other clients at any time.

The server notifies clients of changes of state by means of *events*. An event may be a side effect of a client request, or it may have a completely asynchronous cause, such as the user's pressing a key or moving the pointer. In addition, a client may send an event, through the server, to another client.

Each client asks the server to send that client events of particular types that occur with respect to particular windows. The server generally reports an event with respect to some window. For example, the keyboard is conceptually attached to a window, known as the *focus* window. When the user presses a key, the server usually reports an event with respect to the focus window. If a client has asked the server to send it events of type **KeyPress** occurring with respect to some window, the server sends that client an event whenever the user presses a key while that window has the focus.

From the point of view of a client reading events from the server, events that result from that client's own requests arrive in the order in which it makes the requests. However, those events may be interspersed with events that result from other causes, such as user input or another client's actions. Furthermore, the client may buffer requests and the server may buffer events before actually transmitting them, so an event may arrive long after the client makes the request that generates the event.

The point is that for the most part event processing in X is inherently asynchronous. Most client applications continually loop, reading an event, processing the event (possibly making requests during the processing), and then reading another event. The client cannot assume, for example, that a given input event was generated after a given client request just because the client read the event after it made the request.

Many events have *timestamps* that indicate when the server actually generated the events. A client that depends on the temporal ordering of events must often examine these timestamps.

1.2 Xlib, Xt, and Motif

Applications do not communicate with the X server directly. Instead, they use one or more libraries that provide high-level interfaces to the X protocol. The three principal libraries available to a Motif application are Xlib, the X Toolkit Intrinsics (Xt), and the Motif toolkit.

1.2.1 Xlib

X clients do not have to deal with the server at the level of the X protocol. X includes a C language client interface to the protocol, called Xlib. Among the Xlib facilities are the following:

- Routines for creating and managing the basic server resources, including windows, pixmaps, fonts, cursors, GCs, and properties
- Routines for rendering text and graphics into drawables
- Buffering of requests to the server and queuing of events from the server
- Data structures representing events of all types, and routines for selecting and reading events
- Routines for handling colormaps and for using device-independent color spaces
- Routines for generating text input and output in different locales
- The X resource manager (Xrm), a database of options specified by the user or the application

The resource manager is the keystone of a fundamental tenet of X: that the user and application should control the appearance, interaction style, and other optional characteristics of a client. For example, the background and foreground colors and the fonts used by an application might be represented as resources. Typically, an application provides default values for such resources but allows the user to override the defaults.

A resource is a triple consisting of a name, a class, and a value. A class may include a set of resources with different names. Resources may be arranged hierarchically; a name and class may consist of components, each identifying the name or class of a particular level of the hierarchy. The **fully qualified** name or class of a resource is the list of names or classes at all levels, starting with the name or class of the application and ending with the name or class of the resource itself.

The resource manager permits a user or application to specify resource values in a file, on the command line while starting the application, or by calling an Xrm routine in the program. A resource specification must include either the name or the class of the resource, but it may be either partially or fully qualified according to name, class, or a mixture of name and class components. The resulting resource database may include a variety of general and specific resource specifications. When an application queries the database for a resource value, it supplies a fully qualified name and class. The resource manager uses a search algorithm that returns the value from the most specific specification that matches the requested name and class.

1.2.2 Xt

Although Xlib provides the fundamental means of interacting with the X server, developing a complex application using only Xlib would be a formidable task. Xlib essentially supplies the primitives for an X client. A complex application needs to combine these primitives into constructs that handle aspects of interaction with the server in a more general way.

X includes a library, the X Toolkit Intrinsics (abbreviated Xt), that supplies some of these higher-level interfaces. Three of the most important Xt contributions are the following:

- Objects, known as widgets, used to hold data and present an interface to the user
- Management of widget geometry
- Dispatching and handling of events

1.2.2.1 Widgets

At the heart of Xt is a set of data abstractions built on an object metaphor. Each of these objects, called a *widget*, is a combination of state and procedure. Each instance of a widget is a member of a class. A widget *class* holds a set of procedures and data structures that are common to all widgets of that class. A widget *instance* contains the procedures and data structures that are particular to that single widget. A widget instance also has a pointer to its class.

Each widget class typically provides the general behavior associated with a particular kind of interaction with the user. For example, Motif has a widget class designed to let the user enter and edit text. This class provides the general behavior to support text input and display, including editing, selection, cutting, and pasting of text. The class has data structures related not only to the content of the text but also to the appearance of the widget's on-screen representation. To use this class, an application creates an instance of this class of widget and provides some of its own data and procedures for the widget instance.

Xt supports single inheritance of widget classes. That is, a widget class may be a subclass of another class, its superclass. A subclass is often a specialized variant of a more general superclass. The subclass may inherit, override, or supplement the procedures and data structures of its superclass. Xt generally supplies widget classes designed to be superclasses for other classes. Motif supplies the subclasses of which the application constructs widget instances. Section 1.3 summarizes the Motif and Xt widget class hierarchy.

Widget instances form another, separate hierarchy. Every widget except the top-level widget (or widgets) in an application has a parent widget. Widgets of some classes, called *composites*, may have children. Other kinds of widgets, often called either *primitives* or *gadgets*, generally do not have children. An application constructs one or more trees of widget instances made up of composites, primitives, and gadgets. For example, a menu may consist of a composite parent representing the menu and a number of primitive children representing buttons. The menu and its children are one branch of the overall widget tree of the application.

Xt and Motif provide all the widget classes that most applications need. It is possible for an application to define new widget classes, but this requires knowledge of Xt and of Motif internals that is beyond the scope of this book. A typical application creates widget instances of the built-in classes, providing its own procedures and data for its widgets.

Xt uses an extension of the resource mechanism to represent the widget instance data that is available to an application. Each widget class defines a set of resources that apply to widgets of that class. A class may inherit or override the resources of its superclasses as well.

A widget class declares a name and a class for each of its resources. Xt and Motif give each widget class a name, and the application gives each widget instance a name. Finally, the application developer provides a name and a class for the application itself. For a given resource of a given widget, the fully qualified name is the list of names beginning with the application name, continuing with the name of the top-level widget and then with the names of descendant widgets down to the name of the given widget, and ending with the name of the resource. The fully qualified class is the list of classes beginning with the application class, continuing with the class of the top-level widget and then with the classes of descendant widgets down to the class of the given widget, and ending with the class of the resource.

The user, the application, and the widget class combine to provide values for resources and thus to control the appearance and other attributes of components of the application. Both the user and the application developer can provide either specific or general specifications for widget resources in several resource files and on the command line. They can also supply different resource specifications depending on the locale, the characteristics of the screen, or arbitrary customization criteria.

When the application starts up, Xt combines these specifications into an initial resource database. When the application creates a widget, Xt assigns initial values to the widget's resources using a specification from the database, from values supplied by the application at creation time, or from defaults supplied by the widget class. After creating a widget, the application can use the **XtGetValues** routine to retrieve the value of a widget resource and the **XtSetValues** routine to supply a new value for a resource.

1.2.2.2 Widget Geometry

Most widgets either have an associated window or occupy a defined rectangular area of their parent's window. Each widget has a height, width, and a position with respect to its parent, expressed as the x and y coordinates of the upper left corner of the widget. Specification of the dimensions of widgets and their positions with respect to each other constitutes the layout or *geometry* of the application.

Application geometry results from the interaction of several factors:

- The user or application may supply values for resources that influence geometry, such as the height and width of a widget.
- A **window manager**, a special client that controls the positions and sizes of top-level windows, runs on most displays. Motif provides a window manager called the Motif Window Manager (MWM). The user can change the size of most top-level windows by means of window manager facilities.
- A child widget may have preferences about its size. For example, a widget that displays a label may wish to be wide enough to display all the text of the label.
- A parent widget may have preferences about the sizes and locations of its children. For example, a menu widget may wish to lay out its button children aligned in a given number of rows or columns.

The process of accounting for all these factors and determining widget layout is known as **geometry management**. Xt provides the essential means of handling geometry management:

- All widgets have resources that specify, either directly or indirectly, the geometry intended by the user or the application.
- Xt has special widgets known as *shells* whose function is largely to handle interaction between the application and outside agents such as window managers.
- Widget class procedures may ask the widget's parent to change the widget's geometry, may calculate a preferred size, and may recalculate the widget's layout when the widget is given a new size.
- Parent widgets have ultimate control over the geometry of their children. A widget class procedure of a parent may accept or reject a child's request to change its geometry. In general a parent may change a child's geometry at any time.

A child is *managed* when it and its parent are prepared to negotiate geometry. In general, widgets are eligible to appear on the screen only after they are managed.

See Chapter 14 for more information on geometry management and the specific management policies of Motif widgets.

1.2.2.3 Event Handling

Xt has an event-handling procedure that reads events from the server and dispatches them to appropriate widgets. Each widget that has an associated window may also have a **translation table**. This table maps descriptions of events to names of procedures, known as *actions*. When Xt reads an event associated with a widget, it looks up the event description in the translation table and dispatches the event to the associated action routine.

An application can provide its own action routine, but most such routines are supplied by the widget class. An action routine often takes some action on its own and then notifies the application by invoking an application procedure known as a *callback*. Many widgets have resources whose value is a list of callback procedures. The widget invokes the procedures on these lists at specified times, often when the widget receives certain kinds of events. Xt supplies other means for an application to receive and respond to events, but many applications need only add appropriate callback procedures. These callbacks do most of the "work" of the application in the course of interacting with the user.

The Xt event-handling mechanism leads naturally to an event-driven structure for an application program. Most applications have the same general form:

- Initialize the application
- Create and manage the application widgets
- Provide callback procedures to be invoked by widgets
- Enter the Xt event-dispatching loop, which usually does not return

See Chapter 3 for more information about the structure of a Motif application.

1.2.3 Motif

Xt provides the substrate for creating a set of widgets responsible for specific aspects of a user interface. Motif uses the Xt substrate to build both base classes and specialized subclasses of widgets for a variety of purposes. Section 1.3 outlines the Motif widget set.

In addition to supplying widgets, Motif adds a number of features that are of general use to applications and users. The following sections summarize some of these features.

1.2.3.1 Visual Style

Motif widgets have a distinctive visual style. Many widgets have *shadows* with a three-dimensional look that makes the widget appear to be raised above or depressed below the background. A widget that has keyboard focus may have a rectangular *highlight* border. When the user presses the Btn1 mouse button and focus is in a button, the color of the button face changes to indicate that the user has selected or "pressed" the button.

Motif automatically generates default colors for widget foregrounds, shadows, highlights, and selections states. The user or application can supply its own colors or pixmaps as values for widget resources.

See Chapter 12 for more information on colors and pixmaps in Motif.

1.2.3.2 Selections and Drag and Drop

The X Window System establishes conventions for clients to follow in allowing the user to transfer data from one application to another. These transfers operate through *selections* of several kinds, including primary, secondary, and clipboard selections. A selection is a shared resource that can be owned by only one client at a time for a given display. When the user wants to transfer data from one application to another, the receiving client asks the selection owner to convert the data into a form the receiving client understands, and then the receiver inserts the data. This mechanism can also transfer data between one widget and another in the same application.

The Motif Text and TextField widgets support primary, secondary, and clipboard selections. Motif also has routines that handle the clipboard selection, allowing an application to copy data easily to and from the clipboard. Xt provides more general routines for transferring data by means of selections.

Motif has an extensive **drag and drop** mechanism for transferring data. The user begins a transfer by pressing the mouse button attached to the transfer function (usually

Btn2) with the pointer over a data source. The user then drags an iconic representation of the data to a spot that can receive the data, called a **drop site**. When the user releases the mouse button the data is moved, copied, or linked to the drop site.

The Motif Text, TextField, List, and Label subclasses automatically support drag and drop transfer of textual and some pixmap data. Motif includes an extensive programming interface of objects and routines that allow an application to establish its own drag sources and drop sites, control negotiation between sender and receiver, customize the visual elements, and convert arbitrary kinds of data.

See Chapters 16 and 17 for more information about data transfer.

1.2.3.3 Keyboard Traversal

Motif provides two styles of transferring keyboard focus from widget to widget. In one style, the widget that contains the pointer has focus. In the other style, the user presses a key or Btn1 to move focus to another widget, and the pointer location does not otherwise affect the focus.

In the second style, Motif distinguishes between traversal to a composite or a widget with internal navigation, called a **tab group**, and navigation to a widget or element within a tab group. Motif has a number of resources and routines to control traversal using this style.

See Chapter 13 for more information on keyboard traversal and other input issues.

1.2.3.4 Compound Strings and Renditions

Motif represents much textual data using a data type called a **compound string**. (Also called **XmString**.) This is a stream of components representing text, a display direction, and rendition tags. These rendition tags specify how the text is to appear when rendered, including parameters such as the following:

- Font
- Color
- Tab stops (if any)

- Underlining
- Other text features

A compound string can have multiple text segments, possibly with different directions and rendition tags. Motif uses compound strings to represent all text except that in the Text and TextField widgets.

For each widget that can contain text, Motif maintains information about fonts, color, tab stops, and other qualities of written text using a data type called a **render table**. A render table is a list of entries, each consisting of either a font or a font set and the rendition information. A font set is a construct representing a group of fonts needed to display text in the locale of the application.

When Motif displays the text of a compound string segment, it matches the segment's rendition tags with tags from the widget's render table. It then uses the associated font or font set to display the text of the segment. A special rendition tag (called **_MOTIF_DEFAULT_LOCALE**) indicates text to be parsed in the encoding of the locale and displayed using the fonts needed for that locale.

See Chapter 9 for more information on compound strings, fonts, renditions, and render tables. Chapter 11 contains information on using these tools to prepare an application for different language environments.

1.2.3.5 Motif Window Manager

The Motif Window Manager (MWM) is a Motif client that is capable of managing windows of either Motif or non-Motif applications. MWM provides window decorations and functions for moving, resizing, raising, lowering, maximizing, and minimizing windows. The user can display icons either on the root window or in an icon box. MWM has many resources that permit the user to customize its appearance and behavior.

See Chapter 18 for more information on the application interfaces to MWM. See the **mwm**(1) reference page in the *Motif 2.1—Programmer's Reference* for information on MWM resources and functions.

1.2.3.6 UIL and MRM

Motif has a specification language called the User Interface Language (UIL). The developer uses UIL to define widgets and data in a text file, and then compiles this file into a binary format. At run time the application, using Motif Resource Manager (MRM) routines, retrieves the widget descriptions and data definitions from the binary file, and MRM creates the widgets and data structures from these descriptions.

UIL and MRM work in conjunction with the Motif toolkit. The application defines callback procedures and interacts with the widgets as if it were using the Motif toolkit alone. By using UIL to define the program's widget hierarchies, the developer can separate the user interface specification from the application code. A developer can change the interface by editing and recompiling a text file without recompiling and relinking the application program. As with resource files, a developer can use separate UIL files to contain text, render tables, and other data specific to particular locales.

See Chapter 4 for information on using UIL and MRM in an application. See the **UIL**(5) reference page in the *Motif 2.1—Programmer's Reference* for information on UIL syntax.

1.2.4 Using Xlib, Xt, and Motif

Xt is built atop Xlib, and Motif is built atop Xt. One goal of Xt is to give applications a set of high-level interfaces and objects that relieve the program of the need to deal with many primitive Xlib routines. A goal of Motif is to give applications still higher-level interfaces and, particularly, a versatile set of widgets to relieve the program of the need to define its own widgets for most tasks.

However, Xt does not strive to replace all Xlib interfaces, and Motif does not strive to replace all Xt interfaces. Even a simple Motif application must use basic Xt routines to initialize the toolkit, manage widgets, create windows for widgets, get and set resources, add callback routines, and enter the event-dispatching loop.

Many Motif applications do not need to call Xlib routines. However, Motif does not have its own graphics routines, color-space facilities, or support for application management of input methods. Programs that need these features must either use vendor-supplied tools or call Xlib routines directly.

As a general rule, an application should use the highest-level interfaces sufficient for the tasks at hand. Not only does this usually result in a concise program, but it also ensures that the program functions as intended when a higher-level procedure supersedes a lower-level procedure.

For example, Xlib, Xt, and Motif all have routines to set keyboard focus to a window or widget. Xt and Motif both maintain an internal state that keeps track of focus changes. If a Motif application uses the Xt or Xlib routine, it may cause Motif or Xt to become internally inconsistent.

By convention, the names of Xlib routines and data structures begin with "X"; the names of Xt routines and data structures begin with "Xt"; and the names of Motif routines and data structures begin with "Xm".

This book does not document Xlib or Xt interfaces. A Motif application developer must have a working knowledge of basic Xt application interfaces and should have at least general familiarity with Xlib. For more information on Xlib, see the X Consortium Standard *Xlib—C Language X Interface*. For more information on Xt, see the X Window System document *X Toolkit Intrinsics—C Language Interface*.

1.3 Widget Classes and Hierarchy

This section gives a brief overview of the hierarchy of widget classes in Xt and Motif. Chapter 2 discusses this hierarchy in more detail.

1.3.1 Xt Classes

Xt defines the base classes for all widgets. **Core** is the fundamental class for all widgets that can have windows. This class has basic resources for a widget’s geometry, background color, translations, and sensitivity to input. Widgetlike objects that do not have windows—called *gadgets* in Motif—are subclasses of **RectObj**. This class has geometry resources but no colors or translations.

Composite is the base class for all widgets that can have children. This class maintains a list of its children and is responsible for managing their geometry. **Constraint** is a subclass of **Composite** that maintains additional data for each child, represented by **constraint resources** for the child.

Shell is the base class for shell widgets. Shells envelop other widgets whose windows are children of the root window. Shells are responsible for interaction with the window manager. **Shell** is a subclass of **Composite**. Xt has the following subclasses of **Shell**:

OverrideShell	Envelops widgets that the window manager should ignore, such as menus
WMShell	Superclass for shells that need to interact specifically with the window manager
VendorShell	Subclass of **WMShell** that implements toolkit-specific behavior
TransientShell	Subclass of **VendorShell** for widgets such as dialogs that appear briefly on behalf of other widgets
TopLevelShell	Subclass of **VendorShell** for top-level widgets for components of the application
ApplicationShell	Subclass of **TopLevelShell** for the top-level widget that represents the application as a whole

1.3.2 Motif Classes

Motif has three broad groups of widgets: primitives and gadgets, managers, and shells.

1.3.2.1 Primitives and Gadgets

Primitives are widgets that have no children. They are commonly the fundamental units of input and output, and they are usually building blocks for composite widgets. **XmPrimitive**, a subclass of **Core**, is the base class for all primitives. **XmPrimitive** has basic color resources and provides keyboard traversal behavior.

XmPrimitive is used only as a superclass for classes with more specific behavior. Following are the subclasses of Motif primitives:

XmSeparator
: Used to separate other widgets; usually appears as a line.

XmLabel
: Displays text or a pixmap. As a superclass for buttons, provides specialized behavior, such as keyboard traversal, inside menus.

  XmLabel subclasses **XmCascadeButton**, **XmDrawnButton**, **XmPushButton**, and **XmToggleButton** perform some action when activated or "pressed" by the user. Subclasses have roles as menu activators, toggles, pushbuttons, and small graphics areas.

XmScrollBar
: Control that moves a scroll widget horizontally or vertically with respect to a fixed viewport on the scroll.

XmList
: Array of textual items from which the user can select one or more entries.

XmText
: Widgets for display and possible editing of text. **XmText** may be multiline or constrained to a single line. **XmTextField** is a variant optimized for single-line text.

Gadgets are variants of primitives that have no windows. Gadgets have geometry, but they inherit colors from their parents and depend on their parents to dispatch input events to them. **XmGadget**, a subclass of **RectObj**, is the base class for gadgets. Gadget variants exist for separators, labels, and most button classes. **XmIconGadget**, which displays both text and a pixmap in various combinations, can be manipulated by an **XmContainer** widget.

See Chapter 10 for more information on the Text and TextField widgets. See Chapter 5 for more information on other primitives.

1.3.2.2 Managers

A manager is a widget that generally has children and manages their geometry. **XmManager**, a subclass of **Constraint**, is the base class for managers. This class has special responsibilities when it has gadget children. It provides color resources that its gadget children inherit, and it dispatches input events to appropriate gadgets. Following are the subclasses of Motif managers:

XmFrame Surrounds a child with a shadow and a margin.

XmScale Displays a value within a range and optionally allows the user to supply a new value.

XmPanedWindow
Arranges children, called panes, vertically or horizontally (from left to right or right to left, depending on **XmNorientation**). May also insert a control called a sash that lets the user adjust the size of a pane.

XmScrolledWindow
Provides a viewport onto a child widget that behaves as a virtual scroll. Manages ScrollBars to move the scroll with respect to the viewport.

XmMainWindow
Subclass of ScrolledWindow that provides support for a MenuBar and other specialized areas.

XmRowColumn
Implements menus, RadioBoxes, and CheckBoxes, usually consisting of button children. Can be used to lay out arbitrary widgets in rows, columns, or two-dimensional formations.

XmBulletinBoard
Superclass for dialogs, widgets that present information to the user or seek information from the user. The dialog widget may be a BulletinBoard, which provides general behavior, or a specialized subclass. Common subclasses present a list from which the user makes a selection, display filenames and allow the user to choose one, ask the user to enter a command, and display a message. One subclass, **XmForm**, performs general constraint-based geometry management for its children.

XmDrawingArea
General-purpose manager suitable for use as a canvas for graphics operations.

XmContainer
Manages a collection of certain kinds of child widgets (typically, IconGadgets). These child widgets can be viewed in several different layout formats, selected using different selection types and techniques,

and directly manipulated by the user. One possible use for a Container widget would be to build a graphical user interface to a file system.

XmComboBox
: Manages a TextField widget and a List widget. A user can select an item from a ComboBox by clicking on it in the List widget or by typing it directly into the TextField widget.

XmSpinBox
: Allows the user to select a value from a ring of related but mutually exclusive choices which are displayed in sequence. For example, a user might use a SpinBox to select a month by clicking through the months on one ring and to select the day of the month by clicking through the days on another ring.

XmNotebook
: Organizes its children into pages, tabs, status areas, and page scrollers to simulate a real notebook. Each page of the Notebook can hold different kinds of items. For example, the first page could contain text in a Text widget, the second page might be a graphic displayed in a Label widget, and the third page might be a questionnaire managed by a Form widget.

1.3.2.3 Shells

Motif has three shell classes:

VendorShell
: Motif-specific implementation of the Xt class. Among other responsibilities, manages communication with MWM.

XmDialogShell
: Subclass of **TransientShell** that envelops dialogs. Cooperates with BulletinBoard in popping up and positioning transient dialogs.

XmMenuShell
: Subclass of **OverrideShell** that envelops menus.

1.3.2.4 Other Motif Classes

Motif uses a number of specialized objects that are not intended to be used in creating widgets. These objects exist primarily to hold resources and other information that would be difficult to make available in another way. **XmDisplay** holds resources specific to a given display, and **XmScreen** holds resources specific to each screen on which the application has created a widget. The drag and drop interface includes objects representing several aspects of a drag and drop transaction, including the general context, drop sites, drag icons, and data transfers.

In addition to the Motif objects provided by the standard Motif toolkit, your application may use widgets provided by other vendors. In fact, you can write your own Motif widgets. See the *Motif 2.1—Widget Writer’s Guide* for details.

1.4 Header Files and Libraries

Xlib, Xt, and Motif all have header files (also called **#include** files) that an application must include in its code. However, the Motif header files themselves include the required Xt files, which in turn include the required Xlib files. An application usually needs to include only the proper Motif files.

All Motif applications must include the file **<Xm/Xm.h>**. This file contains definitions that all applications need. Each Motif widget also has a **#include** file. An application must include this header files for all widgets it creates. In addition, some groups of Motif routines have their own header files. Required **#include** files for each Motif widget and routine are documented in the *Motif 2.1—Programmer’s Reference*.

Instead of using a large number of header files for particular widgets and routines, an application can include **<Xm/XmAll.h>**. This file incorporates all documented Motif header files. Including **<Xm/XmAll.h>** may slow compilation a little; however, including **<Xm/XmAll.h>** will not increase the size of the resulting application. In other words, the link profile will not change as a result of including **<Xm/XmAll.h>** instead of including individual header files.

When building a Motif application, a developer must link the program with the appropriate libraries. Xlib, Xt, the Motif toolkit, and MRM have separate libraries. An application that does not use MRM must be linked with the Motif toolkit, Xp, Xext, Xpm, Xt, and Xlib libraries. An application that uses MRM must be linked with these libraries and also with the MRM library. A developer might also need to link the application with additional libraries, depending on the platform and operating system. Consult your system administrator and vendor documentation for more information on the libraries required for Motif applications.

Chapter 2

The Motif Programming Model

Motif accommodates a variety of application programming styles. An application can accomplish most tasks, such as handling a particular kind of user input or displaying a particular kind of output, in more than one way. While this flexibility is one of the strengths of Motif, the toolkit has been designed with a set of programming principles in mind. This chapter explains at a general level the intended uses of Motif widgets and other features of the toolkit. The next chapter outlines the structure of common Motif programs, and succeeding chapters explain toolkit features in more detail.

The following general principles make sense in writing any Motif program:

- Adopt a user-centered perspective. In most Motif programs, the application does its work in response to commands or other input from the user. An important part of interface design is deciding precisely which commands, options, and other information the user can give the application. The interface then consists largely of procedures that execute the user's commands or otherwise respond to the user's input.

- Separate the design of the core application and the user interface. The core application should not depend on a particular user interface. Often it's a good idea to specify a set of generic routines and data structures for obtaining input and displaying output. The developer can then implement these routines in different ways to provide different user interfaces for the application.

- Follow the *CDE 2.1/Motif 2.1—Style Guide and Glossary* in designing the user interface. Although an application can use Motif widgets in many configurations, users find some more common, intuitive, and comfortable than others. The *CDE 2.1/Motif 2.1—Style Guide and Glossary* contains requirements and recommendations for compliant applications, and it offers more advice on application design.

- Outline the widget hierarchy. Once you have settled on one or more combinations of widgets, you may find the implementation more tractable if you sketch a genealogy of all the widgets the program uses. Constructing a widget tree can reveal gaps and awkwardness in the design. Attaching dialogs and menus to the hierarchy may help ensure consistency and completeness in the presentation and solicitation of information.

- Use high-level interfaces when possible. A Motif application must use some X Toolkit Intrinsics (Xt) interfaces, and it may call other public Xt and Xlib routines. For some tasks, such as drawing graphics, an application must call lower-level routines. However, Motif provides interfaces such as resources, callback lists, and convenience routines to handle many common tasks. Motif also includes both simple and composite widgets that do most of the work related to their specific functions, such as text editing or constraint-based geometry management. Using a high-level Motif interface instead of a comparable series of lower-level calls can make code simpler and more maintainable.

- Use resource files and the User Interface Language (UIL) to specify characteristics of the interface. Avoid locking the user-interface specification and data into the application code. Using resources gives the user the power to override application-supplied default behavior. UIL provides the opportunity to separate the widget hierarchy from the application. With both resources and UIL, the developer can change the interface without recompiling the application code. These mechanisms also provide the means to tailor the interface and data for particular language environments.

2.1 A User-Centered Model

A basic principle of Motif and Xt programming is that the user is in charge of the application. Except in unusual circumstances, the program takes action in response to commands or other input from the user. In fact, a typical Motif program spends most of its real time waiting for the user to provide input.

The fundamental object type in a Motif interface is the *widget*. Some widgets can display output or process input or both; some widgets serve to contain other widgets. A widget is usually associated with a *window* or a rectangular area of the screen. A widget also has attributes, called *resources*, which can often be set by the user or the application. An application organizes widgets into one or more hierarchies or trees of parent widgets and their children.

Motif and Xt define a set of widget types or *classes*. A widget class may be a subclass of another class; in that case it inherits some of the attributes and behavior of the superclass. Motif has three basic classes of widgets:

- **Primitives** are the basic units of input and output. Primitives usually do not have children. Specialized Motif primitives include labels, separators, buttons, scroll bars, lists, and text widgets. Some primitive classes have equivalent objects called *gadgets*. These are just like primitives except that, to enhance performance, they have no associated windows.
- **Managers** are composite widgets that contain primitives, gadgets, or other managers. Managers are responsible for the geometrical arrangement of their children. They also process and dispatch input to their gadget children. Specialized Motif managers include frames, scrolled and paned windows, menus, constraint-based geometry managers, and several kinds of dialogs.
- **Shells** are widgets whose main purpose is to communicate with the window manager. Most shells have only one child, and they maintain the same size and position as the child. Specialized Motif shells exist to envelop applications, dialogs, and menus.

Defining a widget hierarchy is one of the two main tasks of a Motif application. The other is to define a set of **callback procedures**. Callbacks are the primary means by which the application responds to user input. When the user takes an action like pressing a key or a mouse button, the X server sends the application an *event*. Xt dispatches these events to the appropriate widget, usually the one to which the user directed the input. Xt maps the event to one or more widget **action routines**. The action may change the state of the widget and, if the application has asked to be notified of that action, may "call back" to the program by invoking an application callback procedure.

Many Motif widgets have resources that are lists of callback procedures. Motif invokes a list of callbacks when the user takes an action that has a particular meaning. For example, most buttons have callbacks that Motif invokes when the user *activates* the button. The user may activate the button in a number of ways, such as by pressing the osfSelect key or the Btn1 mouse button. The events that constitute activation and other meaningful user actions are defined in a general way in the *CDE 2.1/Motif 2.1—Style Guide and Glossary* and are documented for specific widgets in the *Motif 2.1—Programmer's Reference*.

The user action may cause Motif to change to the state and appearance of a widget. For example, when the user presses osfSelect in a PushButton, Motif may make the button appear to be depressed and then released, like a mechanical push button. The action may have other effects depending on the context. For example, Motif has a dialog widget called a FileSelectionBox, used for finding and selecting files. When the user activates the "filter" PushButton in a FileSelectionBox, Motif searches for and displays the names of files that match a pattern displayed elsewhere in the FileSelectionBox.

In general Motif takes care of changing the state and appearance of a widget to correspond to the user's action. By default, though, this action has no effect on the application. The application programmer must interpret the meaning of the action for the application by providing a callback routine, which Motif invokes when the user takes that action. The callback routine may change the state of the application, for example, by changing the value of a variable when the user selects a new value from a Scale widget. The callback may cause the application to take an action. It may also change the state of one or more widgets itself, or it may create an entirely new widget hierarchy.

When both Motif and the application have finished responding to a user action, the application waits for the user to provide more input. Xt provides a routine in which applications spend most of their time. This routine waits for an event, dispatches it to the appropriate widget, and then waits for another event. After initializing the toolkit and creating the initial widget hierarchy, most applications enter this loop and remain there until the user terminates the program.

Motif and Xt provide other ways for applications to direct and respond to events, but for simple programs, virtually the entire interface between the user and the application consists of callback routines.

2.2 Separating Interface from Application

A widely accepted principle of application design is that a core application should not rely on a specific user interface. Separating the application from the interface allows developers to work on the two components independently. It also allows the program to run with different interfaces without changes in the core application. This makes it easier to port the application to more than one interface and to experiment with different configurations of a single interface.

Many applications need to collect input from the user and to display output in some form. It may be easier to separate the core application from the user interface if the developer specifies a set of generic input and output routines along with any necessary data structures. If these generic interfaces have no dependence on specific user interfaces, they can be implemented in different ways for different interfaces without changing the core application. They form a module for communication between the core application and the interface.

The Motif implementation of the interface module consists of code to perform the following tasks:

- Initialize the Intrinsics
- Create the widget hierarchy
- Define callback procedures
- Make widgets visible
- Enter a loop that waits for and responds to user input

These steps are explained in detail in Chapter 3.

The User Interface Language (UIL) helps enforce the separation of the interface from the core application. With UIL, the developer defines widgets and their characteristics in a text file and then compiles the text file into a binary format. At run time, the application uses Motif Resource Manager (MRM) routines to retrieve the widget descriptions from the binary file, and MRM creates the widgets from these descriptions. The UIL file can also define data such as text strings and colors, and MRM can retrieve the data at run time.

In this way, an application can remove the description of the widget hierarchy from the program code. In its source code, the application defines callback procedures and interacts with the widgets as if it were using the Motif toolkit alone. If the application has defined all the callback procedures it needs, a developer can change the widget hierarchy by editing and recompiling the UIL file without recompiling and relinking the source program.

2.3 Building Blocks: Primitive Widgets and Gadgets

Primitive widgets are the fundamental units of input and output in Motif. Primitives are commonly the widgets at the leaves of an application’s widget hierarchy. These widgets do not have children of their own. The name *primitive* does not imply simplicity; some primitives, such as the Text widget, have quite complicated behavior. *Primitive* is meant to contrast with *manager*, a widget that usually has children. It also suggests a basic component from which composite widgets are built. Primitives are often referred to as *controls*.

The **XmPrimitive** Motif widget class is the superclass for all primitives. **XmPrimitive** is itself a subclass of the fundamental Xt widget class, **Core**. **Core** has resources that describe the widget’s width, height, and x and y coordinates with respect to its parent. Other **Core** resources control characteristics of the window, such as its background color; whether or not the widget can receive input events; and the mapping that Xt uses to translate events into calls to the widget’s action routines.

XmPrimitive adds two groups of features to the **Core** class. One group consists of resources to control additional visual characteristics, including the characteristic three-dimensional shadow and a highlighting rectangle that can appear when the widget is the focus for keyboard input. The second group controls **keyboard traversal**, the use of the keyboard to move focus from one widget to another. This group includes several resources and a set of translations and actions that allow the user to move the keyboard focus to another widget by pressing an arrow key. **XmPrimitive** also provides callbacks to let the application provide help information when the user presses osfHelp.

The **XmGadget** widget class is the superclass for all gadgets. **XmGadget** is a subclass of the Xt widget class **RectObj**. This class provides resources to determine the dimensions and position of the gadget’s rectangular area inside its parent. **XmGadget** is equivalent to **XmPrimitive**, with two exceptions:

- Gadgets have color and pixmap resources, but if these resources are not set, gadgets inherit their values from their parents.
- Gadgets do not have translations or actions. A gadget's parent controls keyboard traversal from the gadget to another widget, and it dispatches events to the gadget when appropriate.

XmPrimitive and **XmGadget** are used only as superclasses for other classes of widgets. **XmPrimitive** and **XmGadget** are not *instantiable*; that is, an application cannot create an actual widget that is an instance of either of these classes. Motif has several specialized subclasses of primitives and gadgets, summarized in the following sections.

2.3.1 Label and Separator

Labels provide the ability to display static (uneditable) text or a pixmap. A Label or LabelGadget itself is useful for displaying a message, title, or description. Label and LabelGadgets are also superclasses for buttons used as menu items, toggles, or controls.

A Label can display either text or a pixmap. When a Label displays text, it uses a construct called a **compound string**. This is a stream of components that represents zero or more pieces of text, each with an associated rendition tag and display direction. When Motif displays the compound string, it matches each tag with a tag in the widget's render table and uses the corresponding font or fonts, colors, and other features so described.

A Separator or SeparatorGadget separates controls or groups of controls. It usually appears as a horizontal or vertical line and supports several styles of line drawing.

Labels and Separators are described in more detail in Chapter 5.

2.3.2 Buttons

A button is a basic control that performs some action when the user activates it. Buttons commonly appear in menus, RadioBoxes and CheckBoxes, SelectionBoxes and MessageBoxes. Motif has the following classes of buttons:

- A CascadeButton or CascadeButtonGadget is used inside a menu and, when activated, usually causes a PulldownMenu to appear.
- A PushButton or PushButtonGadget can appear either inside or outside a menu. It performs some action determined by the application. When a PushButton is armed, or ready to be activated, it changes its appearance so that it looks as if the user has pressed it in. When it is disarmed, it reverts to the appearance of extending out.
- ToggleButtons and ToggleButtonGadgets have different states: like toggle switches, they are either on or off. They can appear in menus or in nonmenu RowColumn WorkAreas, including RadioBoxes and CheckBoxes. They can also have an indeterminate state.
- A DrawnButton is an empty button surrounded by a shadow border. It is intended to be used as a PushButton but with graphics drawn by the application.

Buttons are described in more detail in Chapter 5.

2.3.3 ScrollBar

A widget can act as a viewport onto a virtual scroll. The scroll is a plane with text, graphics, a list of items, or other contents. The viewport is a fixed-size window onto a portion of the scroll.

A ScrollBar is the control that moves the viewport horizontally or vertically relative to the underlying scroll. A ScrollBar consists of a rectangle, called the scroll region, representing the full size of the scroll. It has a smaller rectangle, called the slider, within the scroll region, representing the position and size of the viewport relative to the full scroll. The ScrollBar usually has arrow graphics at both ends of the larger rectangle.

ScrollBars are described in more detail in Chapter 5.

2.3.4 List

A List is an array of textual items from which the user selects one or more entries. Each item is a compound string. A List has four modes for selecting items: two that allow the user to select one item at a time, and two that allow the user to select more than one item in either contiguous or discontiguous ranges.

Lists are described in more detail in Chapter 5.

2.3.5 Text

The Text widget is available for displaying and possibly editing text, and represents text as a multibyte or wide-character string. When the widget is editable and the user presses a key that represents a text character, that character is inserted into the text. Other translations and actions allow the user to navigate or to select, cut, copy, paste, or scroll the text.

The text can be multiline or constrained to be a single line. In a single-line widget, actions that move up and down one line in a multiline widget instead traverse to another widget, and pressing osfTab moves the keyboard focus to another group of widgets instead of inserting a **Tab** character.

A TextField is essentially the same as a Text widget in single-line mode, except that its performance is optimized for single-line text operations.

Text is described in more detail in Chapters 5 and 10.

2.4 Managers

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

The **XmManager** Motif widget class is the superclass for all managers. **XmManager** is a subclass of **Core**. Like **XmPrimitive**, **XmManager** has resources to control colors or pixmaps used for the foreground, shadows, and highlighting rectangle. Most managers do not have shadows or highlighting rectangles, but gadget children inherit the related resources. Managers also have resources that control keyboard traversal, and they provide callbacks for processing user requests for help. In addition, they have translations and actions for dispatching input events to gadget children, usually to the child that is the current focus of keyboard events.

XmManager is not an instantiable widget class; it is used only as a subclass for other widgets. Motif has several specialized subclasses of managers, summarized in the following sections.

2.4.1 Frame

A Frame is a simple manager that surrounds a single child with a shadow and a margin. A Frame can also have another child that appears as a title for the Frame.

Frames are discussed in more detail in Chapter 8.

2.4.2 Scale

A Scale is a manager that functions as a control. It displays a value within a range and optionally allows the user to supply a new value. Its appearance and behavior are much like those of a ScrollBar without arrows. It also has a title and can display the current value next to the slider. If the application adds other children to a Scale, the Scale positions them evenly along the rectangular area that represents the range of values, and these children then act as tic marks or value labels.

Scales are discussed in more detail in Chapter 5.

2.4.3 PanedWindow

A PanedWindow arranges its children vertically from top to bottom or horizontally, and forces them all to have the same width. Each child is a *pane* of the window. Between each pair of panes, PanedWindow inserts an optional Separator and a control called a *sash*. By manipulating a sash with the mouse or keyboard, the user can increase or decrease the height of the pane above. PanedWindow has resources to control the margins, the spacing between panes, and the appearance of the sashes. Each pane of a

PanedWindow has resources specifying a maximum and minimum height and whether or not either the pane itself or the PanedWindow should be allowed to resize the pane without user intervention.

PanedWindow is discussed in more detail in Chapter 8.

2.4.4 ScrolledWindow and MainWindow

A ScrolledWindow manages a viewport and ScrollBars to implement a window onto a virtual scroll. The user can move the viewport to display different portions of the underlying scroll by using the ScrollBars or keyboard scrolling commands.

ScrolledWindow is capable of performing scrolling operations automatically. In this mode, the application creates the widget that represents the scroll as a child of the ScrolledWindow. The ScrolledWindow then creates a clipping window to act as the viewport, creates and manages the ScrollBars, and moves the viewport with respect to the scroll when the user issues a scrolling command.

ScrolledWindow can also allow the application to perform scrolling operations. In this mode, the application must create and manage the ScrollBars and must change the contents of the viewport in response to the user’s scrolling commands.

List and Text widgets are often used as virtual scrolls. Motif has convenience routines to create List and Text widgets inside ScrolledWindows, and the resulting ScrolledList and ScrolledText widgets perform scrolling operations without intervention by the application.

MainWindow is a subclass of ScrolledWindow that is intended as the primary window in an application. In addition to a viewport and ScrollBars, MainWindow includes an optional MenuBar and an optional command window and message window.

The ScrolledWindow and MainWindow widgets are described in more detail in Chapter 8.

2.4.5 RowColumn

RowColumn implements both menus and nonmenu WorkAreas. Menus are widgets that allow the user to make choices among actions or states. Motif offers four basic kinds of menu:

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

One use for a nonmenu RowColumn WorkArea is to contain a set of ToggleButtons constituting a RadioBox or a CheckBox. When the user selects a ToggleButton, its state changes from on to off or from off to on. Another use is to lay out an arbitrary set of widgets in a row, column, or two-dimensional formation.

RowColumn is discussed in more detail in Chapter 6.

2.4.6 BulletinBoard, Form, MessageBox, SelectionBox

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

The usual superclass for a dialog widget is **XmBulletinBoard**. The dialog widget can be either a BulletinBoard itself or one of its more specialized subclasses. BulletinBoard is a container with no automatically created children; it supplies general behavior needed by most dialogs. Its subclasses provide child widgets and specific behavior tailored to particular types of dialogs:

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

Dialogs are discussed in more detail in Chapter 7.

2.4.7 DrawingArea

A DrawingArea is a manager suited for use as a canvas containing graphical objects. An application must interact with a DrawingArea at a somewhat lower level than with other Motif widgets, but a DrawingArea provides the application with more finegrained information about events. DrawingArea has callbacks to notify the application when the widget is exposed or resized and when it receives keyboard or mouse input. An application generally must use Xlib routines to draw into the DrawingArea, and the application is responsible for updating the contents when necessary. The flexibility of a DrawingArea makes it a useful widget for implementing both graphical and text features not provided by other Motif widgets.

DrawingArea is discussed in more detail in Chapter 15.

2.4.8 ComboBox

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

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

ComboBox is discussed in more detail in Chapter 6.

2.4.9 Spin Boxes

A SpinBox is a manager that functions as a control. It creates a pair of arrows that can be used to spin through a set of choices. The choices, which are usually related but mutually exclusive, are displayed consecutively one at a time in a single text field.

Choices can be a range of numeric values or an ordered list of compound strings. The arrow buttons allow the user to advance or back up through the choices until the desired choice is displayed.

SimpleSpinBox is a ready-to-use subclass of SpinBox. The text field for a SimpleSpinBox is created automatically.

SpinBox and SimpleSpinBox are discussed in more detail in Chapter 6.

2.4.10 Container

A Container is a manager that accepts only widgets that are classes or subclasses of **XmIconGadget** as children. It arranges these children in different formats or views depending upon its resource settings, and allows selection and manipulation of the children.

Container is discussed in more detail in Chapter 8.

2.4.11 Notebook

A Notebook is a manager that displays only one child at a time. Each child is assigned a page number in the Notebook and is displayed by user manipulation of tab buttons and page scrollers in the Notebook.

Notebook is discussed in more detail in Chapter 8.

2.5 Shells

Users of X Window System applications normally employ a window manager, a special application that may control the positions, sizes, and border decorations of top-level windows on the display. Motif supplies its own window manager, the Motif Window Manager (MWM), but Motif applications can cooperate with other window managers as well.

A window manager communicates with other applications through a protocol defined in an X Window System document, the *Inter-Client Communication Conventions Manual* (ICCCM). Xt and Motif define a group of widgets whose main responsibility is to envelop other widgets and communicate with the window manager. These widgets are called shells.

A shell is nearly invisible to the application. Each shell has a single managed child, and the shell's window usually remains coincident with the child's window. The application must create shells when needed, but many Motif convenience routines that create widgets also create shells automatically. Once it has created a shell, the application may not need to handle the shell again. For example, an application can position or resize a Motif shell by positioning or resizing the child widget.

Each widget with a top-level window—that is, a window whose parent is the root window of the screen—needs to be enclosed in a shell. This is true of the main application widget, but it is also true of dialogs, menus, and any top-level widgets other than the main application widget. Motif provides three classes of shell: VendorShell, DialogShell, and MenuShell.

2.5.1 VendorShell

VendorShell is the shell class that provides Motif-specific behavior for shells other than those surrounding menus. It is responsible for communication between the application and MWM. VendorShell is a superclass for other classes. TopLevelShell is an Xt subclass of VendorShell that surrounds a top-level widget in an application. ApplicationShell is another Xt subclass of VendorShell that surrounds the main widget in the application.

Many applications create only one ApplicationShell. A program can create this shell explicitly, or it can use the Xt convenience routine **XtAppInitialize** to initialize the application and automatically create the ApplicationShell.

2.5.2 DialogShell

A DialogShell is a VendorShell subclass that envelops dialogs. Although the window manager takes account of dialogs, they are usually transient; they appear to provide information to or solicit information from the user, and then they disappear. DialogShell is a subclass of the Xt TransientShell class, which keeps track of the application to which the dialog belongs. Users cannot iconify a dialog separately from the main application window.

DialogShell is designed to have a child that is a subclass of BulletinBoard. Most Motif convenience routines that create dialogs create DialogShell parents automatically.

2.5.3 MenuShell

MenuShell is the class of shell that surrounds PopupMenus and PulldownMenus. MenuShell is a subclass of the Xt OverrideShell class. This class enables the shell to bypass the window manager. Most Motif convenience routines that create PopupMenus and PulldownMenus create MenuShell parents automatically.

2.6 Applications, Top-Level Widgets, and Dialogs

Primitives, managers, and shells are the components Motif provides for building an interface. A developer assembles these components into the broadest units of the program: dialogs, top-level widgets, and the application itself.

One approach to this construction is to specify the connection between the core application and the user interface. The developer determines what information the application needs to obtain from and present to the user. From this assessment, the developer specifies a generic interface to the application and then implements a Motif version using particular combinations of widgets.

Another approach is to design the user interface from the application level down to specific widgets. The developer decides what the top-level components of the application should be and how they relate to each other. From this assessment, the developer designs a combination of widgets that presents the application clearly to the user and permits a graceful transition from one task to another. The developer can then finely adjust the visual appearance of the interface.

In practice, a developer is likely to use both the bottom-up and top-down approaches at different stages of the program design. The approaches converge at the level of the application.

2.6.1 Applications

The application is the highest level of abstraction of a Motif program. In one sense the application embodies the entire program. In another sense, the application is the primary widget in the program. The user may cause other widgets to appear, but the application is the focus of activity and is usually the first widget to appear when the user starts the program.

The widget that represents the application is commonly a MainWindow. For many applications, the essential operations should be available from the MenuBar at the top of the MainWindow. By browsing through the MenuBar, the user can quickly determine what general functions the application provides. The activation callbacks for the buttons in menus that are pulled down from the MenuBar initiate the general operations of the application. The *CDE 2.1/Motif 2.1—Style Guide and Glossary* contains requirements and recommendations for the contents of the application MenuBar and its PulldownMenus.

The MainWindow usually contains a large scrollable work area. Single-component applications usually perform most of their work using this region. Other applications may require more than one work area.

An ApplicationShell encloses the main widget of an application. The developer can use the Xt function **XtAppCreateShell** to create an ApplicationShell directly or can let Xt create the shell during a call to **XtAppInitialize**.

Usually a program has only one application, but sometimes a program is made up of multiple logical applications. In this case, the program may have more than one main window, each enveloped in a separate ApplicationShell. Multiple ApplicationShells are also required for applications which use multiple displays.

2.6.2 Top-Level Widgets

Although it is unusual for a program to have more than one logical application, it is more common for an application to require multiple top-level widgets. For example, a mail-processing program may consist of a component for reading mail and another for composing and sending it.

Each major component of an application may reside in a top-level widget. Each top-level widget must be enclosed in a TopLevelShell or an ApplicationShell. One approach is to have a single ApplicationShell for the application, with each TopLevelShell a *popup* child of the ApplicationShell. The program does not create a window for the ApplicationShell. Another approach is to designate one top-level widget the application, enclosed in an ApplicationShell, and make the other TopLevelShells popup children of the ApplicationShell. A popup child is one whose window is a child of the root window and whose geometry is not managed by its parent widget.

Multiple top-level widgets are discussed in more detail in Chapter 3.

2.6.3 Dialogs

Dialogs are transient components used to display information about the current state of the application or to obtain specific information from the user. A dialog widget is usually a BulletinBoard or one of its subclasses, enclosed in a DialogShell. The DialogShell is a popup child of another widget in the hierarchy. Its window is a child of the root window, but the user cannot iconify a dialog separately from the main application.

A dialog can be *modal*—that is, it can prevent other parts of the application from processing input while the dialog is active. It can also be *modeless* so that the user can interact with the rest of the application while the dialog is visible. Motif has convenience routines that create both the dialog widget and the DialogShell for several kinds of information.

Dialogs are discussed in more detail in Chapter 7.

2.7 Resources: User and Program Customization

A widget, a class of widgets, and an application as a whole have a set of attributes that the program can examine and that the user and program may be able to specify. These attributes are implemented as X *resources*. Xlib has a facility called the X resource manager (Xrm) whose purpose is to establish and query databases of resources. Xt and Motif build on Xrm to make resources the repository of publicly available attributes of widgets as well as applications.

Xt maintains databases of resources that apply to several levels:

- To the application as a whole
- To the display on which an application is running
- To the screen on which a widget hierarchy is created
- To a class of widgets
- To an individual widget

The user can specify resources at any of these levels through resource files or the command line used to start the program. The application can also specify resources through resource files.

Each application has a name and a class; each widget within an application has a name and a class; and each resource has a name and a class. When supplying resource values in a file or on the command line, the user or the application specifies the scope of the resource value by qualifying the resource according to its name or class. For example, a user might specify that all resources of the class Background should have a particular value for all widgets; or the user might specify that only the resource named background within a particular hierarchy of named widgets should have a particular value. The qualification mechanism allows resource values to be specified at any level.

Most widget classes define a set of resources, by name and class, that apply to those classes. Subclasses inherit superclass resources, unless a subclass overrides the superclass resource specification. A widget class also defines a default value for each of its resources, used in case the user and the application do not provide another value.

When an application starts up, Xt constructs an initial database of resource values. This database is derived from a combination of user and application resource files and the command line. Some resources in the database may have different values depending on the display or the screen on which the application is running. When an application creates a widget, Xt uses this initial database in combination with the widget class resource defaults to supply values for the widget's resources. The application can override these values by supplying arguments to the routine that creates the widget. It can set a resource value after creating the widget by using the Xt function **XtSetValues**.

Setting resources is the primary means by which an application changes the attributes of a widget. However, an application should be careful not to override the user's specification of many resources governing characteristics such as visual appearance and the policy for determining which widget has keyboard focus. In general, the application should set only those resources necessary for the proper functioning of the program. An application can specify preferences for other resource values in an application defaults file. Xt reads this file when an application starts up, but a user can override the values supplied there.

The process by which Xt creates the initial resource database is discussed in more detail in Chapter 3.

2.8 Handling Input and Output

The X server communicates input to a client through input *events* associated with a window. In the simplest case, when a keyboard or pointer event occurs, the X server sends the event to the client that has expressed interest in events of that type on the window that contains the pointer. However, processing can be more complex. A client can *grab* a pointer button or key, the pointer or keyboard, or the entire server; the client then receives the relevant events. A client can set the **input focus** to some window, and the X server then reports events with respect to this window even if the pointer is outside this window.

To insulate applications from such complexities, Xt and Motif supply facilities for low-level processing of user input to an application:

- A VendorShell resource, **XmNkeyboardFocusPolicy**, allows the user or application to determine whether keyboard events go to the widget that contains the pointer or the widget in which the user presses Btn1 (a "click-to-type" policy).
- In the click-to-type model, the user can also use keys to navigate from widget to widget or from one group of widgets to another.

- Xt provides the basic event-dispatching loop used by most applications. Xt takes events out of the application’s queue and dispatches them to the appropriate widget, usually the widget that has input focus. Xt usually invokes an *action* associated with the particular event through a table of *translations* from event specifications to action routines. The action, in turn, often invokes a callback list.
- Motif and Xt provide *mnemonics* and *accelerators*, which are shortcuts for taking actions associated with a widget when the widget does not have input focus. A *mnemonic* is a keysym for a key that activates a visible button in a menu. An *accelerator* is a description for an event that invokes an action routine through a translation.

Most applications can use these high-level interfaces, allowing Xt and Motif to process user input at lower levels. If an application needs more control, it can also provide its own **event handler**, a routine invoked by the Xt dispatching loop when the widget receives events of the specified type. An application can also provide its own event-dispatching loop.

Issues of input, focus, and keyboard navigation are discussed in more detail in Chapter 13.

For most widgets, Xt and Motif handle low-level output processing as well. For example, in a Label or Text widget, when an application changes the text to be displayed, Motif automatically redisplays the contents of the widget. Most widgets have resources that control the appearance of the output, such as the fonts used to display text.

Motif provides the DrawingArea widget for applications that need to produce graphic output or that need more control or flexibility in displaying text. DrawingArea is discussed in more detail in Chapter 15.

Chapter 3

Structure of a Motif Program

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

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

- Include the required header files
- Initialize the Intrinsics
- Create one or more widgets

- Define callback procedures and attach them to widgets
- Make the widgets visible
- Enter the event loop

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

Table 3–1. Steps in Writing Widget Programs

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

3.1 Including Header Files

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

Each Motif widget also has an **#include** file. An application must include the header files for all widgets it creates. In addition, some groups of Motif routines have their own header files. For example, an application that uses any of the Motif clipboard routines must include the file **<Xm/CutPaste.h>**. Required **#include** files for each Motif widget and routine are documented in the *Motif 2.1—Programmer's Reference*.

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

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

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

3.2 Initializing the Intrinsics

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

1. Initialize the state of the Intrinsics. An application can also do this by calling **XtToolkitInitialize**.

2. Create an application context. Xt uses this construct to contain the information it associates with each instance of an application. Its purpose is to allow multiple instances of an application to run in a single address space. Most applications need only create an application context and pass it to Intrinsics routines that take an application context as an argument. The data type is *XtAppContext*. An application can create an application context explicitly by calling **XtCreateApplicationContext**.

3. Open a connection to a display and attach it to an application context. When an application uses **XtAppInitialize**, the display specification comes from the command line invoking the application or from the user's environment. After opening the display, Xt builds a resource database by processing resource defaults and command-line options. The construction of this database is described in the next section. An application can perform these steps explicitly by calling **XtOpenDisplay**. If an application already has an open display as a result of

calling **XOpenDisplay**, it can attach the display to an application context and build the initial resource database by calling **XtDisplayInitialize**.

4. Create a top-level shell widget for the application. **XtAppInitialize** creates an ApplicationShell and returns it as the function’s return value. An application can create a top-level shell by calling **XtAppCreateShell**.

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

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

3.2.1 The Initial Resource Database

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

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

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

3.2.1.1 File Search Paths

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

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

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

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

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

3.2.1.2 Initial Database Components

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

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

3.2.1.3 Command-Line Specifications

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

3.2.1.4 Per-Host User Resources

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

3.2.1.5 Screen-Specific Resources

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

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

3.2.1.6 Server or User-Preference Resources

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

3.2.1.7 User Application File

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

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

   $XAPPLRESDIR with **%C**, **%N**, **%L** or with **%C**, **%N**, **%l**, **%t**, **%c**
   $XAPPLRESDIR with **%C**, **%N**, **%l**
   $XAPPLRESDIR with **%C**, **%N**
   $XAPPLRESDIR with **%N**, **%L** or with **%N**, **%l**, **%t**, **%c**

```
$XAPPLRESDIR with %N, %l
$XAPPLRESDIR with %N
$HOME with %N
```

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

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

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

3.2.1.8 Application Class Resource File

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

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

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

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

3.3 Creating Widgets

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

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

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

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

Some convenience routines, such as **XmCreateMenuBar**, create specialized widgets. These routines usually set some initial resource values to configure the widget for a particular use—for example, to configure a RowColumn widget for use as a MenuBar. In some cases, such as **XmCreatePulldownMenu** and **XmCreateScrolledList**, these routines create a widget hierarchy rather than a single widget. The documentation for each convenience routine in the *Motif 2.1—Programmer's Reference* explains what the routine does.

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

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

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

3.3.1 Setting Resources during Widget Initialization

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

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

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

The documentation for each widget class in the *Motif 2.1—Programmer’s Reference* lists the data type and default value for each resource. For resources whose default values are computed dynamically, the documentation describes how the default values are determined.

3.3.2 Arguments that Specify Resource Values

To specify initial resource values in a call to a widget creation routine, an application supplies two arguments: a list of elements representing resource settings and an integer specifying the number of elements in the list. Each element in the list is a structure of type **Arg**. This structure has two members: a string representing the name of the resource, and a value specifier representing the resource value. The value specifier is of type *XtArgVal*. This is a data type large enough to hold a *long* or one of several types of pointers to other data. If the resource value is of a type small enough to fit into an *XtArgVal*, the value specifier contains the resource value itself; otherwise, it contains a pointer to the actual value. For most resources, an application supplies integer values (including such types as **Position** and **Dimension**) directly in the value specifier; otherwise, the application supplies a pointer to the value.

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

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

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

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

3.3.3 Setting Resource Values

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

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

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

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

3.3.4 Retrieving Resource Values

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

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

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

```
XtGetValues(text, args, n);
```

3.3.5 Resource Value Data Types

The documentation for each widget class in the *Motif 2.1—Programmer's Reference* lists the data types to use when setting and retrieving values for resources. The user and application do not always have to supply data of the type documented. Motif has routines, called converters, that convert resource values from one data type to another. For example, when a value for the resource database comes from a file or the command line, Motif processes the value as a string. Motif and Xt have routines to convert strings to most common resource types, including **Boolean**, **Dimension**, **Position**, **Pixel**, and **XmFontList**.

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

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

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

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

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

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

3.3.6 Resource Values and Memory Management

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

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

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

For resources whose values are not immediate data, **XtGetValues** sometimes makes a copy of values and sometimes does not. For example, Motif always makes copies of compound strings retrieved by **XtGetValues**, but it does not make copies of lists of compound strings (data of type **XmStringTable**). An application should free compound strings retrieved by **XtGetValues**, but in general it should not free values of other types unless the documentation for the particular resource in the *Motif 2.1—Programmer's Reference* says the application must free that value.

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

Table 3–2. Motif Memory Deallocation Routines

Routine	**Recovers Memory Used By:**
XmFontListEntryFree	A font list entry
XmFontListFree	A font list
XmFontListFreeFontContext	A font list context

XmImFreeXIC	The widgets associated with a specified X Input Context (XIC)
XmParseMappingFree	A parse mapping
XmParseTableFree	A parse table
XmRenderTableFree	A render table
XmRenditionFree	A rendition
XmStringConcatAndFree	Two input compound strings
XmStringFree	A compound string
XmStringFreeContext	The string scanning context
XmTabFree	A tab
XmTabListFree	A tab list

3.4 Adding Callback Procedures

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

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

- The Xt event dispatcher calls an event handler whenever an event of a particular type occurs in a specified widget.
- The Xt translation manager calls an action routine when an event sequence matches an event specification in a widget translation table. In a translation table, actions are associated with event specifications. More than one event sequence can invoke the same action routine.

- A Motif widget invokes callback procedures when user input signifies an action that is meaningful to the widget, such as activating a PushButton. Widgets often invoke callbacks from action routines. More than one action can invoke the same callback list.

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

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

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

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

The documentation for each widget class in the *Motif 2.1—Programmer’s Reference* describes any callback structures that the widget passes to callback procedures as widget data. Note that a callback procedure can change the values of some members of these structures. Because the order of procedures in a callback list is unspecified, an application that uses multiple callback procedures in the same list must use caution in changing these values.

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

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

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

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

    for(i = 0, valve_p = valves; i < N_VALVES;
                                    i++, valve_p++) {
        sprintf(name, "valve_state_%d", i);
        toggles[i] = XmCreateToggleButton(parent, name,
            (ArgList) NULL, 0);
        XtAddCallback(toggles[i], XmNvalueChangedCallback,
            (XtCallbackProc) ToggleValueChangedCB,
            (XtPointer) valve_p);
    }
```

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

3.5 Making Widgets Visible

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

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

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

3.5.1 Managing Widgets

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

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

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

When managing more than one child of a realized parent, it is more efficient for an application to call **XtManageChildren** than to call **XtManageChild** separately for each child being managed. Widget layout can be computationally expensive, and

XtManageChild invokes the parent's **change_managed** procedure each time it is called. **XtManageChildren** calls the parent's **change_managed** procedure only once for all children being managed.

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

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

3.5.2 Realizing Widgets

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

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

Note these implications:

- Geometry negotiation proceeds from the bottom up; then window creation proceeds from the top down.
- After a widget is realized, all its managed descendants are realized and, by default, mapped.

- If no widget in the tree is realized, all geometry negotiation between parents and their managed children takes place before any widget is realized.

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

3.5.3 Mapping Widgets

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

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

3.5.4 Multiple Screens, Displays, and Applications

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

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

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

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

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

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

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

    app_shell = XtAppInitialize(&app, "Example",
        (XrmOptionDescList) NULL, 0, &argc, argv,
        (String *) NULL, (ArgList) NULL, 0);
    display = XtDisplay(app_shell);
    for (i = 0; i < ScreenCount(display); i++) {
        sprintf(name, "top_shell_%d", i);
        n = 0;
        XtSetArg(args[n], XmNscreen,
            ScreenOfDisplay(display, i));    n++;
        top_shell = XtCreatePopupShell(name,
                        topLevelShellWidgetClass, app_shell,
                        args, n);
        /* Create and manage descendants of top shell */
        ...
        /* Realize and map the top shell */
        XtPopup(top_shell, XtGrabNone);
    }
    ...
}
```

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

3.6 Entering the Event Loop

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

3.7 Writing Threaded Applications

In general, writing multithreaded Motif applications requires familiarity with:

- General concurrent programming principles.
- Specific threads interfaces.

- New interfaces in X11R6 Xt for multithreaded applications.

3.7.1 Why Multithreading?

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

3.7.2 Input Processing Loop

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

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

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

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

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

3.7.3 Xt Interfaces For Multithreading

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

3.7.4 Initializing Xt For Use In Multiple Threads

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

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

3.7.5 Using XtAppLock and XtAppUnlock

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

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

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

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

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

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

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

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

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

3.7.6 New XtAppMainLoop

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

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

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

```
XtAppContext app;
do {
        XEvent event;
        XtAppNextEvent(app, &event);
        XtDispatchEvent(&event);
} while(XtAppGetExitFlag(app) == FALSE);
```

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

3.7.7 Destroying An Application Context

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

3.7.8 Event Management In Multiple Threads

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

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

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

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

Chapter 4

Structure of a Program Using UIL and MRM

The User Interface Language (UIL) allows an application developer to separate the specification of particular widget hierarchies from the application source code. The application defines widgets and their characteristics in a text file, which the developer compiles into a User Interface Definition (UID) file in binary format. At run time the application, using Motif Resource Manager (MRM) routines, retrieves the widget descriptions from the binary file, and MRM creates the widgets from these descriptions. The application defines callback procedures and interacts with the widgets as if it were using the Motif toolkit alone.

UIL offers several advantages over toolkit-only applications:

- UIL enforces the separation of the user interface specification from the application.
- A developer can change the interface by editing and recompiling a text file without recompiling and relinking the application program.
- The UIL compiler generates warnings for errors that the developer otherwise would not discover until running the program, if then. For example, the UIL compiler checks the spelling of resource names.

- The toolkit may handle large databases more efficiently when they are represented as UID files rather than resource files.

An application that uses UIL has two separate components: the UIL file and the application program.

The UIL file consists mainly of definitions of the application’s widget hierarchy. The declaration for each widget typically includes the following components:

- Widget type
- Widget children
- Initial resource values
- Declarations for callback procedures

The UIL file can also define values for data such as compound strings, colors, and icons.

The structure of the application program is similar to that of a toolkit-only program. The chief difference is that, instead of explicitly creating each widget, the program uses MRM routines to retrieve widget definitions from the UID file and to create the widgets themselves. The program might also use MRM routines to retrieve data values defined in the UIL file. An application program using UIL must take the following actions:

- Include the required header files
- Initialize the Intrinsics
- Initialize MRM
- Open the UID file
- Register the names of callback procedures and values of identifiers specified in the UID file
- Retrieve and create widgets and data defined in the UID file
- Close the UID file
- Define callback procedures
- Make the widgets visible
- Enter the event loop

4.1 Structure of a UIL Module

A UIL module is a block of declarations and definitions for the values, procedures, literals, and objects that make up a user interface specification. Each UIL file contains either one complete module or, if the file is to be included in another UIL file, at least one complete top-level construct within a module.

Each module has the following structure:

- A *module* clause
- Zero or more declarations for the module as a whole
- Zero or more *include* directives
- Zero or more *value* declarations
- Zero or more *identifier* declarations
- Zero or more *procedure* declarations
- Zero or more *object* declarations
- Zero or more *list* declarations
- An **end module** clause

This section discusses the components of a UIL module, but it does not describe the UIL syntax in detail. For more information, see the **UIL(5X)** reference page in the *Motif 2.1—Programmer's Reference*.

4.1.1 The module Clause

Each module begins with the declaration *module name*. The keyword *module* must be in lowercase.

4.1.2 Module-Level Declarations

Several optional declarations at the beginning of the module modify characteristics of the module as a whole:

names The *names* declaration specifies whether names in the UIL file are stored in a case-sensitive or case-insensitive way. The following declaration, the default, means that names are stored as they appear in the UIL file, and all UIL keywords must be in lowercase:

```
names = case_sensitive
```

The following declaration means that all names are stored in uppercase, and UIL keywords can be in uppercase, lowercase, or mixed case:

```
names = case_insensitive
```

The entire *names* declaration itself must be in lowercase, and it affects only the part of the module that follows it.

character_set
The **character_set** clause declares the default character set for strings and compound strings specified in the module by double quotes ("*string*"). If this clause is not present, UIL derives the default character set from the language environment in which the UIL file is compiled. This does not affect the character set of strings specified in the module by single quotes ('*string*'). UIL derives the character set of these strings from the language environment in which the UIL file is compiled. The character set in this clause must be either a keyword representing one of the character sets UIL knows about or a character set returned by the **character_set** function.

objects The *objects* clause specifies whether UIL should define objects of the specified types as widgets or gadgets. For example, this declaration specifies that UIL should define objects of type **XmPushButton** to be gadgets:

```
objects = { XmPushButton = gadget; }
```

A declaration for an individual object can override this specification.

4.1.3 The include Directive

The *include* directive includes the contents of a file in the current module. The directive consists of the keywords **include file** followed by a string representing the filename. If the filename has a full directory specification, UIL searches that directory for the file. Otherwise, UIL searches the directory of the main UIL source file and then the directory of the current UIL source file. The **–I** option to the **uil** command adds a directory to the search list.

Included files are useful for definitions common to more than one UIL module. In conjunction with the **–I** option to **uil**, they are also useful in internationalizing applications. Localized definitions for strings, font lists, and the like can reside in files included from different directories depending on language environment. In this case, the *include* directives should not specify the directories; instead, you can use the **–I** option to **uil** to compile files for different language environments without editing or duplicating UIL files.

4.1.4 The value Declaration

The *value* clause defines one or more names and associates them with values. The names can stand for values elsewhere in the module.

The specification for each value is either a literal expression or a call to a UIL function that generates a value. Each value has a UIL type that depends on the representation of the literal or the type of value returned by the UIL function. For more information on UIL types, literals, and functions, see the **UIL(5X)** reference page in the *Motif 2.1—Programmer's Reference*.

By default, the names and their associated values are private to the module. The *value* declaration can also export a value to other modules or import a value from another module. For each name declared to be imported, MRM assigns the value from the corresponding exported declaration at run time.

In this example, the value **id_1** is exported:

```
value
    id_1: exported 1;
    label_1: compound_string('Off');
```

Another module can use the value **id_1** as follows:

```
value
    id_1: imported integer;
```

4.1.5 The identifier Declaration

An *identifier* clause declares one or more names that can appear elsewhere in the module. At run time, MRM assigns values to these names from data defined in the application program. The application uses the **MrmRegisterNames** or **MrmRegisterNamesInHierarchy** routine to establish the correspondence between UIL identifier names and application-defined data. The UIL compiler performs no type checking on identifiers.

The following example identifies names for x and y values that the application defines at run time:

```
identifier
    app_x_value;
    app_y_value;
```

4.1.6 The procedure Declaration

A *procedure* clause declares names of callback procedures or of creation routines for user-defined widgets. The application program itself defines the actual procedures. As with identifiers, the application must use **MrmRegisterNames** or **MrmRegisterNamesInHierarchy** to associate the procedure names with the actual procedures at run time.

For a callback procedure, the *procedure* declaration can also specify the type of data represented by the second argument (the application data pointer) to the callback routine:

```
procedure
    toggle_cb (integer);
    push_button_cb (integer);
```

4.1.7 The object Declaration

An *object* clause defines a widget or gadget and assigns a name that can stand for the object elsewhere in the UIL module. As with values, an object definition by default is private to the UIL module, but the *object* clause can declare it to be exported or imported. In addition to the UIL name, the *object* clause specifies the object's type and a list (enclosed in braces) that can define children, initial resource values, and callback procedures.

4.1.7.1 Object Type

The object type specification is a keyword that is usually the same as the name of the corresponding toolkit widget class. For example, the type keyword for a MainWindow is **XmMainWindow** and for a PushButton is **XmPushButton**. UIL also allows type specifications that correspond to toolkit convenience routines for creating some kinds of specialized widgets, including menus, dialogs, ScrolledList, and ScrolledText. For example, the keyword *XmPulldownMenu* specifies a PulldownMenu, and the keyword *XmPromptDialog* specifies a PromptDialog.

The *object* clause can also specify that the object is to be either a widget or a gadget, overriding the default specified by the *objects* clause. For example, the following defines a PushButtonGadget:

```
object
    pb: XmPushButton gadget {};
```

Alternately, an *object* clause can specify a gadget by using the gadget class name (for example, **XmPushButtonGadget**) as the type specification.

4.1.7.2 Children

An *object* clause can specify the children of a composite widget. This specification appears inside the object list section and consists of the keyword *controls* followed by a list of child declarations. The declaration for each child consists of an object type and, usually, a name that refers to the definition for the child widget in its own *object* clause. Instead of a name for the child, the declaration can contain an entire local definition for the child widget in the form of an object list section. The child declaration can optionally begin with the keyword *managed* or *unmanaged*, which specifies whether or not MRM should manage the child after creating it. The default is to manage the child.

Some manager widgets automatically create children. For example, MainWindow creates three separators to separate its main components. The *controls* list can contain declarations for these children so that the UIL file can specify resource values for them. The declaration for an automatically created child begins with a specification of the name of the child, formed by prepending **Xm_** to the actual name of the child widget. The names of automatically created children are documented in the reference pages for the manager widgets in the *Motif 2.1—Programmer’s Reference*.

Following is an example of specifications for child widgets:

```
object
    main_win: XmMainWindow {
        controls {
            XmMenuBar main_menu;
            Xm_Separator1 { arguments
                               {
                                XmNseparatorType = XmSHADOW_ETCHED_OUT;
                                XmNmargin = 5;
                               };
                          };
            XmScrolledText text_win;
        };
    };
```

In general, a child widget can be of any type the Motif toolkit allows for a child of the parent widget. In some cases, the type of the child differs from the Motif toolkit class. For example, dialogs and menus require shells as their parents, but in UIL a dialog or menu is declared to be a direct child of its parent, with no intervening shell. MRM creates the shell at run time. In this way, UIL and MRM act like the Motif convenience routines for creating dialogs and menus.

Some widget hierarchies in UIL are slightly different from the corresponding hierarchies in the toolkit. For example, in UIL a PulldownMenu in an OptionMenu is described as a child of the OptionMenu, not of the OptionMenu’s parent as it is in the toolkit. In a PulldownMenu system from a MenuBar or a PopupMenu, each PulldownMenu is a child of the associated CascadeButton, not of the CascadeButton’s parent as it is in the toolkit. For more information, see Chapter 6.

4.1.7.3 Resource Values

An *object* clause can specify resource values for MRM to pass to the widget's creation function. This specification appears inside the object list section and consists of the keyword *arguments* followed by a list of resource declarations. The declaration for each resource consists of the name of the resource as in the toolkit (for example, **XmNheight**) followed by = (equals sign) and a value for the resource. The type of the value must be of the proper UIL type for that resource. For information on the required UIL type for each resource, please refer to the *Motif 2.1—Programmer's Reference*.

Following is an example of specifications for initial resource values:

```
object
    main_win: XmScrolledText {
        arguments {
            XmNrows = 10;
            XmNwordWrap = true;
            XmNbackground = color('red');
        };
    };
```

In some cases, UIL provides a value for a resource related to a resource that appears in a specification. For example, if a specification contains a value for **XmNitems** in a List, UIL provides the appropriate value for **XmNitemCount**. For resources of type, dimension, or position, you can specify units by using the syntax described in the **XmNunitType** resource of the **XmPrimitive** reference page.

4.1.7.4 Callback Procedures

An *object* clause can specify procedures to appear in callback lists for the object. This specification appears inside the object list section and consists of the keyword *callbacks* followed by a list of callback list declarations. The declaration for each callback list consists of the name of the callback resource as in the toolkit (for example, **XmNactivateCallback**) followed by = (equals sign) and a value specification for the resource.

In addition to appropriate toolkit resources, the specification can include the special callback list name **MrmNcreateCallback**. MRM invokes callback procedures on this list when it creates the widget. These procedures provide a means for the application to identify the widget ID of a widget created by MRM.

The value specification can be one of two forms:

- If the callback list contains only one procedure, the specification consists of the keyword *procedure* followed by the procedure name and, optionally, a value in parentheses for the application data argument to the procedure.
- If the callback list contains more than one procedure, the specification consists of the keyword *procedures* followed by a list of procedure specifications. Each specification consists of the procedure name and, optionally, a value in parentheses for the application data argument to the procedure.

The UIL compiler issues a warning if a procedure specification contains an application data argument whose type does not match the argument type in the corresponding *procedure* declaration.

The application uses the **MrmRegisterNames** routine or the **MrmRegisterNamesInHierarchy** routine to establish the correspondence between UIL procedure names and the application-defined procedures.

Following is an example of specifications for a callback list:

```
object
    pb: XmPushButton {
        callbacks {
            XmNactivateCallback =
                procedure pb_activate_cb (pb_ident);
        };
    };
```

4.1.8 The list Declaration

A *list* clause defines one or more lists of specifications for resources, callbacks, procedures, or widget children. Each list has a symbolic name that the application can use to refer to the list elsewhere in the UIL file, usually in an *object* declaration.

The main use for this clause is to define lists of specifications that are common to more than one object definition.

A *list* clause consists of the keyword *list* followed by one or more list specifications. Each list specification contains the name, type, and contents of the list. Following are the four kinds of lists:

- A list of resources consists of the keyword *arguments* followed by a list of resource specifications.
- A list of callbacks consists of the keyword *callbacks* followed by a list of callback specifications.
- A list of procedures consists of the keyword *procedures* followed by a list of procedure specifications.
- A list of widget children consists of the keyword *controls* followed by a list of specifications for the children.

In each case, the form of the list is the same as that of the corresponding clause of an *object* declaration.

Following is an example of a *list* declaration:

```
list
    pb_activate_procs: procedures {
        pb_ac_proc_1 ();
        pb_ac_proc_2 ();
    };

list
    pb_callbacks: callbacks {
        XmNactivateCallback = pb_activate_procs;
        XmNarmCallback = procedure pb_arm_proc ();
    };

list
    pb_args: arguments {
        XmNheight = 10;
        XmNbackground = color('red');
    };
```

```
object
    pb_1: XmPushButton {
        arguments {
            arguments pb_args;
            XmNlabelString = pb_label_1;
        };
        callbacks pb_callbacks;
     };

object
    pb_2: XmPushButton {
        arguments {
            arguments pb_args;
            XmNlabelString = pb_label_2;
        };
        callbacks pb_callbacks;
     };

list
    menu_items: controls {
        XmPushButton pb_1;
        XmPushButton pb_2;
    };

object
    menu_1: XmPulldownMenu {
        controls menu_items;
    };
```

4.1.9 The end module Clause

Each UIL module must end with an **end module** clause.

4.2 Structure of a Program Using MRM

The following sections describe the structural elements of an application program that uses the Motif Resource Manager.

4.2.1 Including Header Files

An application that uses MRM must include all the header files it would need if it did not use MRM. These include **<Xm/Xm.h>**, header files specific to each widget the program uses, and any header files needed by Motif routines. In addition, the application must include the file **<Mrm/MrmPublic.h>**. This file contains definitions that the MRM routines need.

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

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

4.2.2 Initializing the Intrinsics

An application initializes the Intrinsics as in any other program, usually by calling **XtAppInitialize**. The application must call **XtDisplayInitialize** either directly or indirectly before opening any UID files.

4.2.3 Initializing MRM

An application that uses MRM must initialize MRM by calling **MrmInitialize** before fetching any widgets from UID files. It is a good idea to call **MrmInitialize** before using any other MRM routines.

4.2.4 Opening UID Files

After initializing MRM and the Intrinsics, an application uses **MrmOpenHierarchyPerDisplay** to find and open one or more UID files that contain the widget definitions and other information to be loaded.

Note: The UID format is portable only across same-size architecture machines.

MrmOpenHierarchyPerDisplay uses search paths in much the same way **XtDisplayInitialize** uses them to build the initial resource database. One argument to **MrmOpenHierarchyPerDisplay** is a list of UID filenames, each of which represents either a full pathname or a name to be substituted in a file search path. The search path comes from the **UIDPATH** environment variable or, if **UIDPATH** is not set, from a series of default paths. **MrmOpenHierarchyPerDisplay** calls **XtResolvePathname** to search these paths. When it uses a search path, **MrmOpenHierarchyPerDisplay** looks for files first by using a suffix of **.uid** and then by using a NULL suffix.

As with the initial resource database, UID files can reside in different directories depending on the language environment. The search paths can include these substitutions, as well as others recognized by **XtResolvePathname**:

- **%N** is replaced by the class name of the application.
- **%L** is replaced by the display's language specification.
- **%l** is replaced by the language part of the language specification.
- **%U** is replaced by the current filename from the list of filenames passed as an argument to **MrmOpenHierarchyPerDisplay**.

MrmOpenHierarchyPerDisplay returns an ID that identifies the list of open UID files for subsequent calls to routines that load data from the files. On each request to load data, MRM searches the list of files in order. This ordered list of open files is the UID hierarchy. The program can retrieve data from the hierarchy until it calls **MrmCloseHierarchy**.

Following is an example of a call to **MrmOpenHierarchyPerDisplay**. The example initializes MRM and the Intrinsics, opens a UID hierarchy, and closes the hierarchy.

```
int main(int argc, char **argv)
{
    Widget          app_shell;
    XtAppContext   app;
    static String  file_names[] = { "app_1", "app_2" };
    MrmHierarchy   hierarchy_id;

    app_shell = XtAppInitialize(&app, "Example",
        (XrmOptionDescList) NULL, 0, (Cardinal *) &argc, argv,
        (String *) NULL, (ArgList) NULL, 0);
    MrmInitialize();
    switch (MrmOpenHierarchyPerDisplay(XtDisplay(app_shell),
             (MrmCount) XtNumber(file_names), file_names,
             (MrmOsOpenParamPtr *) NULL, &hierarchy_id)) {
    case MrmSUCCESS:
        if (MrmCloseHierarchy(hierarchy_id) == MrmSUCCESS) {
            exit 0;
        } else {
            fprintf(stderr,
                     "Unable to close UID hierarchy.\n");
            exit 1;
        }
    case MrmNOT_FOUND:
        fprintf(stderr, "Unable to open UID files.\n");
        exit 1;
    default:
        fprintf(stderr, "Unable to open UID hierarchy.\n");
        exit 1;
    }
}
```

4.2.5 Registering Callbacks and Identifiers

The application must register the names of all callback procedures and identifiers defined in the UIL files. Registering the names associates the symbolic names in the UIL files with procedures and data defined in the program. **MrmRegisterNames** and **MrmRegisterNamesInHierarchy** accomplish this task. Names registered by **MrmRegisterNames** are global to all UID hierarchies, whereas names registered by **MrmRegisterNamesInHierarchy** are local to a particular hierarchy. When MRM looks up the program-defined value associated with a name in a given hierarchy, it searches first for an association local to the hierarchy and then for a global association.

Following is an example using **MrmRegisterNames**:

```
void PBActivateCB_1(Widget pb, XtPointer app_data,
    XtPointer widget_data);
void PBActivateCB_2(Widget pb, XtPointer app_data,
    XtPointer widget_data);
void PBArmCB(Widget pb, XtPointer app_data,
    XtPointer widget_data);

static MrmRegisterArg cb_list[] = {
    { "pb_ac_proc_1",   (XtPointer) PBActivateCB_1 },
    { "pb_ac_proc_2",   (XtPointer) PBActivateCB_2 },
    { "pb_arm_proc",    (XtPointer) PBArmCB }
};

...
    if (MrmRegisterNames(cb_list,
                         (MrmCount) XtNumber(cb_list))
         == MrmSUCCESS) {
        ...
    } else {
        ...
    }
```

4.2.6 Fetching Information from UID Files

MRM can fetch the following information from UID files:

- Named widgets, defined by *object* clauses, and their descendants. Use **MrmFetchWidget** or **MrmFetchWidgetOverride**.
- Named color literals, defined by *color* or *rbg* functions and appearing in *value* clauses. Use **MrmFetchColorLiteral**.
- Named icon literals, defined by *icon* functions and appearing in *value* clauses. Use **MrmFetchIconLiteral**.
- Other named literals appearing in *value* clauses. Use **MrmFetchLiteral** or **MrmFetchSetValues**.

MRM can fetch literals appearing in *value* clauses only if they are defined as *exported*.

After creating a top-level shell, using **XtAppInitialize** or **XtAppCreateShell**, the application can use **MrmFetchWidget** to fetch the child of the top-level shell and its descendants. For each widget in the tree, **MrmFetchWidget** does the following:

- Calls the appropriate widget creation routine, passing it the initial resource values defined in the *arguments* specification in the *object* clause
- Adds the callback routines defined in the *callbacks* specification of the *object* clause
- Calls any **MrmNcreateCallback** callbacks
- Manages all child widgets unless they are defined to be *unmanaged*

The application does not have to fetch all widgets at the beginning of the program. To create widgets such as menus and dialogs as needed, the application can call **MrmFetchWidget** at any time.

The application can fetch the same widget definition more than once. MRM creates a new widget each time, essentially using the UIL definition as a template. **MrmFetchWidgetOverride** is useful here, as it allows the application to override the initial resource values specified in the UIL file.

Following is a simple example using **MrmFetchWidget** to create the main widget hierarchy for an application:

```
int main(int argc, char **argv)
{
    Widget          app_shell, top_level;
    XtAppContext    app;
    static String   file_names[] = { "app_1", "app_2" };
    MrmHierarchy    hierarchy_id;
    MrmType         top_level_class;

    MrmInitialize();
    app_shell = XtAppInitialize(&app, "Example",
        (XrmOptionDescList) NULL, 0, (Cardinal *) &argc, argv,
        (String *) NULL, (ArgList) NULL, 0);
    switch (MrmOpenHierarchyPerDisplay(XtDisplay(app_shell),
             (MrmCount) XtNumber(file_names), file_names,
             (MrmOsOpenParamPtr *) NULL, &hierarchy_id)) {
    case MrmSUCCESS:
        if (MrmFetchWidget(hierarchy_id, "top_level",
              app_shell, &top_level, &top_level_class)
                                    != MrmSUCCESS) {
            fprintf(stderr,
                      "Unable to fetch top-level widget.\n");
        }
        if (MrmCloseHierarchy(hierarchy_id) == MrmSUCCESS) {
            exit 0;
        } else {
            fprintf(stderr,
                      "Unable to close UID hierarchy.\n");
            exit 1;
        }
    case MrmNOT_FOUND:
        fprintf(stderr, "Unable to open UID files.\n");
        exit 1;
    default:
        fprintf(stderr, "Unable to open UID hierarchy.\n");
        exit 1;
    }
}
```

4.2.7 Closing the UID File

MrmCloseHierarchy closes all files in the specified UID hierarchy. The application can close and reopen a hierarchy, but usually it does not close a hierarchy until it is finished reading data from the UID files. When the application uses multiple hierarchies, operating system limits on the number of open files may make it necessary to close one hierarchy before opening another.

4.2.8 Defining Callback Procedures

An application that uses MRM defines callback procedures in the same way as an application that uses only the toolkit. For callbacks delared in UIL files, the application must use **MrmRegisterNames** or **MrmRegisterNamesInHierarchy** to associate the UIL callback procedure names with the actual procedures defined in the program.

An application can create widgets, such as dialogs and PopupMenus, as the program needs them. If these widgets are defined in UIL files, a callback procedure can call **MrmFetchWidget** to fetch them from UID files.

4.2.9 Making Widgets Visible

MrmFetchWidget never manages the widget the application is fetching. It does manage all other widgets in the tree whose root is the widget being fetched, except for widgets declared *unmanaged* in the UIL file. **MrmFetchWidget** does not realize any widgets in the tree.

The application must manage any unmanaged widgets created by **MrmFetchWidget**, and it must realize all widgets it wants to make visible. In the simple case where the application fetches the entire widget hierarchy at the beginning of the program, it typically manages the widget it fetches and then realizes the top-level shell:

```
int main(int argc, char **argv)
{
 Widget         app_shell, top_level;
 XtAppContext   app;
 static String  file_names[] = { "app_1", "app_2" };
 MrmHierarchy   hierarchy_id;
 MrmType        top_level_class;

    MrmInitialize();
    app_shell = XtAppInitialize(&app, "Example",
        (XrmOptionDescList) NULL, 0, (Cardinal *) &argc, argv,
        (String *) NULL, (ArgList) NULL, 0);
    switch (MrmOpenHierarchyPerDisplay(XtDisplay(app_shell),
             (MrmCount) XtNumber(file_names), file_names,
             (MrmOsOpenParamPtr *) NULL, &hierarchy_id)) {
    case MrmSUCCESS:
        if (MrmFetchWidget(hierarchy_id, "top_level", app_shell,
              &top_level, &top_level_class) == MrmSUCCESS) {
            XtManageChild(top_level);
            XtRealizeWidget(app_shell);
        } else {
            fprintf(stderr,
                     "Unable to fetch top-level widget.\n");
        }
        if (MrmCloseHierarchy(hierarchy_id) == MrmSUCCESS) {
            exit 0;
        } else {
            fprintf(stderr,
                      "Unable to close UID hierarchy.\n");
            exit 1;
        }
    case MrmNOT_FOUND:
        fprintf(stderr, "Unable to open UID files.\n");
        exit 1;
    default:
        fprintf(stderr, "Unable to open UID hierarchy.\n");
        exit 1;
    }
}
```

4.2.10 Entering the Event Loop

As with toolkit applications that do not use MRM, a program using MRM typically calls **XtAppMainLoop** to enter the event loop after realizing the top-level shell.

Chapter 5
Basic Controls

Controls are widgets and gadgets with which the user interacts directly. They form the leaves of the widget tree whose root is the application's top-level shell. In most cases, controls are subclasses of **XmPrimitive** or **XmGadget**, and their parents are subclasses of **XmManager**. (**XmScale** is a manager, but in many ways the application treats it as a primitive.) Motif provides the following basic controls:

- Labels, buttons, and separators
- ScrollBar
- Scale
- List
- Text and TextField

5.1 Core, RectObj, XmPrimitive, and XmGadget Classes

Nearly all the basic controls are subclasses of **XmPrimitive** or **XmGadget**. **XmPrimitive**, in turn, is a subclass of the Intrinsics **Core** class, and **XmGadget** is a subclass of the Intrinsics **RectObj** class.

5.1.1 Core

The **Core** class provides basic attributes of all widgets that have associated windows. It has the following groups of resources:

- Specifications of the widget’s x and y coordinates, width and height, and border width
- A resource specifying whether or not the widget is sensitive or able to receive input events from the Intrinsics event manager
- Characteristics of the window, including background and border color or pixmap, colormap, depth, and screen
- A resource controlling whether or not the Intrinsics map the window when the widget is managed
- A table associating translations with actions
- A set of accelerators, which is a translation table bound in the context of a particular widget

5.1.2 RectObj

RectObj is the foundation for gadget classes; it is essentially **Core** without the attributes related to having a window. **RectObj** resources control the position and dimensions of the gadget’s rectangular area within its parent widget. A **RectObj** resource also determines whether or not the gadget is sensitive.

5.1.3 XmPrimitive

XmPrimitive is the fundamental Motif class for all basic control widgets—widgets that do not have children. It includes the following resources and behavior:

- Foreground color, top and bottom shadow colors or pixmaps, and shadow thickness
- Thickness and color or pixmap for the highlighting rectangle, which is displayed when the widget has keyboard focus
- Resources to determine whether the user can traverse to the widget and whether or not it is a tab group
- A resource to determine what unit of measurement the widget uses for size and position resources
- Callbacks for the widget to invoke when the user presses osfHelp
- A resource for the application to use in associating arbitrary data with the widget
- Translations and actions for keyboard traversal to another widget
- A resource that specifies the direction in which components of the primitive (including strings) are laid out.
- Callbacks that allow the application to control which popup menu will be automatically posted.

5.1.4 XmGadget

XmGadget is the fundamental Motif class for all basic control gadgets. **XmGadget** is equivalent to **XmPrimitive**, with two major exceptions:

- It has no associated window.
- It has no translations or actions. The Manager parent controls traversal between its gadget children, keeps track of gadgets that have input focus, and dispatches events to them.

5.2 Labels, Buttons, and Separators

Labels, buttons, and separators are simple widgets built on **XmPrimitive**.

5.2.1 Labels

Labels provide the ability to display static (uneditable) text or a pixmap. Your application can use a Label or LabelGadget to display a message, title, or description. Label and LabelGadget also serve as superclasses for button widgets and gadgets. However, unlike button widgets, Label provides no additional callbacks beyond those that it inherits from Primitive.

The application can specify the following characteristics of Labels, LabelGadgets, and their subclasses:

- The compound string or pixmap to be displayed. When using a pixmap, the application can supply a separate pixmap to be displayed when the widget is insensitive.
- The render table that describes the attributes of the compound string.
- The positioning of the text or pixmap within the widget. One set of resources determines the space allocated for the margins; another determines the distance between the margins and the text or pixmap inside. The **XmNalignment** and **XmNlayoutDirection** resources together determine whether the text or pixmap is centered or is left or right justified within the widget.
- A resource, **XmNrecomputeSize**, that determines whether the widget attempts to remain large enough to contain the text or pixmap. When this resource is True and a resource that affects the size of the text or pixmap, the margins, or the widget itself is changed, the widget tries to resize itself to be just large enough to contain the text or pixmap.

In addition, Label and LabelGadget provide the following facilities for button subclasses in menus:

- A keysym used as a mnemonic to select the button. The user can activate the button by pressing the mnemonic key when the button is visible.

- An accelerator, a KeyPress event by which the user can activate the button whether or not it is visible. Accelerators are supported only for PushButtons and ToggleButtons in PulldownMenus and PopupMenus.
- Translations and actions for keyboard traversal within the menu or menu system.

A Label or LabelGadget can be the source of a drag and drop operation, but cannot be the destination. In other words, you can drag a value from a Label or LabelGadget, but you cannot drop a value into a Label or LabelGadget. The **XmNenableUnselectableDrag** resource of **XmDisplay** holds a Boolean value. If this value is True, then users can drag from a Label or LabelGadget. If this value is False, then users cannot drag from a Label or LabelGadget.

5.2.2 Buttons

A button is a basic control that performs some action when the user activates it. Buttons commonly appear in menus, RadioBoxes, CheckBoxes, SelectionBoxes, and MessageBoxes. This section describes some of the functions of each subclass.

5.2.2.1 CascadeButtons

A CascadeButton or CascadeButtonGadget is used inside a menu and, when activated, usually causes a PulldownMenu to appear. CascadeButtons have the following resources and behavior:

- A pixmap displayed at one end of the widget in a PopupMenu or PulldownMenu to indicate that activating the CascadeButton posts another menu.
- A resource, **XmNsubMenuId**, that holds the widget ID of the PulldownMenu posted when the user activates the button.
- **XmNactivateCallback** callbacks, which the widget invokes when the user activates it, and **XmNcascadingCallback** callbacks, which the widget invokes just before posting a PulldownMenu.
- A resource to provide a delay between the time the mouse enters the widget and the time it posts a menu.

- Translations and actions to activate the widget and to post and unpost PulldownMenus. In general, pressing Btn1 or dragging Btn1 into the widget posts the PulldownMenu. Releasing Btn1 in the widget causes the PulldownMenu to remain posted and enables keyboard traversal. When keyboard traversal is enabled, pressing osfActivate or osfSelect in the widget posts the PulldownMenu and enables keyboard traversal in that menu.

5.2.2.2 PushButtons

A PushButton or PushButtonGadget can appear either inside or outside a menu. It performs some action determined by the application. When a PushButton is armed, or ready to be activated, it changes its appearance so that it looks as if the user has pressed it in. When it is disarmed it reverts to the appearance of extending out. PushButtons provide the following behavior:

- Callbacks that the widget invokes when it is armed, disarmed, and activated. The application usually provides only an **XmNactivateCallback** procedure to perform the action associated with the button.
- Resources to provide a color or pixmap to be displayed when the button is armed and not inside a menu. When a button in a menu is armed, the top and bottom shadows switch colors.
- A resource to determine whether or not the widget considers multiple mouse clicks distinct from single mouse clicks.
- A resource to determine whether or not the button is marked as the default button when outside a menu. In a BulletinBoard, the default button is the one activated when the user presses osfActivate and no other button has keyboard focus. The default button has a distinctive shadow whose thickness is controlled by the **XmNdefaultButtonShadowThickness** resource and whose appearance is controlled by the **XmNdefaultButtonEmphasis** resource of **XmDisplay**.
- Translations to arm, disarm, and activate the button. In general, a button is activated upon pressing Btn1 while on a button, or dragging Btn1 or traversing to a button while in menu. Releasing Btn1 or pressing osfActivate (in a menu) or osfSelect (in the widget) activates and disarms the button.

5.2.2.3 ToggleButtons

ToggleButtons and ToggleButtonGadgets are typically in one of two states: they are either "set" or "unset." In addition, ToggleButtons and ToggleButtonGadgets can also be in an indeterminate state; that is, neither set nor unset. They can appear in menus or in nonmenu RowColumn WorkAreas, including RadioBoxes and CheckBoxes. In a RadioBox only one ToggleButton at a time can be on; in a CheckBox more than one ToggleButton can be on. ToggleButtons can have indicators with distinctive shapes to distinguish whether or not more than one button at a time can be set. However, it is the RowColumn parent, not the ToggleButton, that controls this behavior.

ToggleButtons have the following characteristics:

- Callbacks that the widget invokes when it is armed or disarmed and when it changes state. The widget invokes the **XmNvalueChangedCallback** callbacks when the button's state changes from set to unset or from unset to set.
- Resources to control the appearance of the indicator. If **XmNindicatorOn** is False or if **XmNvisibleWhenOff** is False and the button is in the unset state no indicator is displayed. Otherwise, **XmNindicatorType** determines whether the indicator shows that only one or more than one button at a time can be set.
- A color to be displayed when the button is armed and **XmNfillOnSelect** is True and another color to be displayed when the button is not set.
- Pixmaps to be displayed when the button is selected and the Label or LabelGadget superclass's **XmNlabelType** is **XmPIXMAP**.
- Translations to arm and disarm the button and to change its state. In general, a button is activated upon pressing Btn1 while on a button, or dragging Btn1 or traversing to a button while in menu. Releasing Btn1 or pressing osfActivate (in a menu) or osfSelect (in the widget) activates and disarms the button.

5.2.2.4 DrawnButtons

A DrawnButton is an empty button surrounded by a shadow border. It is intended to be used as a PushButton but with graphics drawn by the application. Like a PushButton, it has translations and actions to arm, disarm, and activate the button and invoke the corresponding callbacks. If **XmNpushButtonEnabled** is True, it draws the shadow so that the button appears pressed in when armed and popped out when disarmed.

Other than this, the application must manage the button's visual appearance. It has **XmNexposeCallback** and **XmNresizeCallback** callbacks to notify the application that the button has been exposed or resized and therefore needs to be redrawn. The application must be careful not to draw within the button's shadows or highlight areas. The application can use a clipping rectangle in the widget's graphics context that takes account of the button's **XmNhighlightThickness** and **XmNshadowThickness**.

5.2.2.5 ArrowButtons

An ArrowButton or ArrowButtonGadget is a button with an arrow graphic and a shadow. A resource controls the direction of the arrow. Unlike other buttons, it is not a subclass of **XmLabel** or **XmLabelGadget**, but it has some of the same behavior as other buttons. It has callbacks that the widget invokes when armed, disarmed, or activated. It has translations and actions similar to those of other buttons to arm, disarm, or activate the button.

5.2.3 Separators

A Separator or SeparatorGadget separates controls or groups of controls. It usually appears as a horizontal or vertical line and supports several styles of line drawing. Resources control its orientation and the type of line it draws. One line style consists of no line at all. This allows the application to control the appearance of the separator by setting its *XmNbackgroundColor* or **XmNbackgroundPixmap**.

5.3 ScrollBar

A widget can act as a viewport onto a virtual scroll. The ScrollBar is the control that moves the viewport horizontally or vertically relative to the underlying scroll. A ScrollBar consists of a rectangle, called the scroll region, representing the full size of the scroll. It has a smaller rectangle, called the slider, within the scroll region, representing the position and size of the viewport relative to the full scroll. The ScrollBar usually has arrow graphics at both ends of the larger rectangle.

A ScrollBar has translations and actions that allow the user to move the slider. By clicking on an arrow, the user moves the slider one small increment in the direction of the arrow. By clicking in the scroll region between an arrow and the slider, the user moves the slider a larger increment (the page increment) in the direction of the arrow. When the ScrollBar has keyboard focus, the user can use the keyboard to move the slider in this way. The user can also drag the slider using the mouse.

By itself, the ScrollBar does not have an association with a widget acting as a viewport onto a scroll. Most applications use a ScrolledWindow, a Manager widget with a child to be scrolled and possibly with one or two ScrollBars to control the scrolling. ScrolledWindow can automatically control the interaction between the scrolled child and the ScrollBars, or it can allow the application to control the interaction. For more information see Chapter 8.

ScrollBar has a number of resources that allow the application to use it to control scrolling:

- A minimum value (**XmNminimum**), representing the position of the slider at one end of the scroll region, and a maximum value (**XmNmaximum**), representing the position of the slider at the other end of the scroll region. These values can be in any integral units the application chooses, so long as the maximum is greater than the minimum.
- The length of the slider (**XmNsliderSize**) between 1 and (**XmNmaximum** – **XmNminimum**).
- A value (**XmNvalue**), ranging between **XmNminimum** and (**XmNmaximum** – **XmNsliderSize**), representing the current position of the slider between the maximum and minimum values.
- Values for the increment (**XmNincrement**) and page increment (**XmNpageIncrement**) by which the user can move the slider.
- A resource (**XmNprocessingDirection**) that determines whether the minimum value is on the left or right for horizontal ScrollBars or is on the bottom or top for vertical ScrollBars.
- Distinct callbacks that the widget invokes when the user moves the slider by one increment in either direction, by one page increment in either direction, or all the way to either end of the scroll region. The widget invokes other callbacks as the user drags the slider and when the user stops dragging the slider. The application does not have to provide routines for all these callback lists; if it provides only an **XmNvalueChangedCallback** procedure, the widget invokes that procedure

whenever the ScrollBar value changes (except during interactive dragging of the slider).

- Resources to control the color of the scroll region, whether the ScrollBar is horizontal or vertical, and whether or not the ScrollBar has arrows.
- Resources to control the delays before the widget moves the slider continuously as the user presses and holds Btn1 on an arrow or the scroll region.
- A resource (**XmNsnapBackMultiple** to specify the distance over which the ScrollBar slider snaps back to its original position when the user drags the mouse outside the ScrollBar edge.
- Resources to specify the appearance of the slider (**XmNsliderMark** and *XmNsliderVisual*).
- A resource (**XmNslidingMode**) to specify the mode the slider works in. For example, this resource could make the ScrollBar look like an old-fashioned mercury thermometer (**XmTHERMOMETER** mode) or like a carpenter's level (**XmSLIDER** mode).

Two convenience routines, **XmScrollBarGetValues** and **XmScrollBarSetValues**, allow the application to get and set the value, slider size, increment, and page increment in one call.

5.4 Scale

A Scale graphically displays a value between two end points. Its appearance and behavior are much like those of a ScrollBar without arrows. However, unlike a ScrollBar, a Scale can optionally display a title. In addition, a Scale can optionally display a textual version of its value to supplement the graphic display of its value.

Like a ScrollBar, a Scale has a minimum, a maximum, and a current value. The current value must be between between the minimum and maximum, inclusive.

Scales can either be editable (the default) or noneditable. The **XmNeditable** resource controls this attribute. Once set, a user cannot change the value in a noneditable Scale. An editable Scale, on the other hand, allows the user to supply a new value.

The increment by which the arrow keys move the slider is always 1, but the application can supply a multiple increment (**XmNscaleMultiple**) analogous to ScrollBar's **XmNpageIncrement**. Scale has two callback lists: **XmNvalueChangedCallback** is invoked when the user changes the value but is not in the process of dragging the slider, and **XmNdragCallback** is invoked when the user changes the value while dragging the slider.

Your application cannot control the size of the slider.

Scale has resources controlling whether the orientation is vertical or horizontal and which end of the Scale represents the minimum value. Other resources control aspects of the Scale's appearance, including the width and height, the title string, whether or not the Scale displays the current value next to the slider, the number of decimal places in the displayed value, and a render table for the title and value.

5.4.1 Drag and Drop in Scale

A Scale can be the source of a drag and drop operation, but it cannot be the destination. In other words, you can drag a value from a Scale, but you cannot drop a value into a Scale. The **XmNenableUnselectableDrag** resource of **XmDisplay** holds a Boolean value. If this value is True, then users can drag from a Scale. If this value is False, then users cannot drag from a Scale.

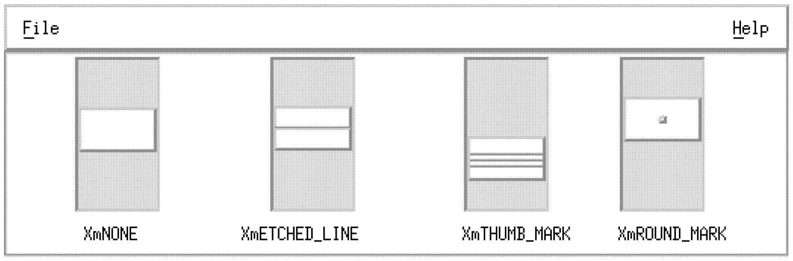
The only part of the Scale that a user can drag is the Scale's current value. More precisely, a user can drag a textual version of the Scale's current value, not the graphic version. The textual version can only be dragged if **XmNshowValue** is True.

5.4.2 Visuals Inside a Scale

The **XmNslidingMode** resource controls the anchoring of the slider. There are two possibilities. One possibility (**XmSLIDER**) is that the slider floats freely inside the Scale. Another possibility (**XmTHERMOMETER**) is that the slider appears tethered to one side of the Scale. The aptly named **XmTHERMOMETER** mode could allow your Scale widget to emulate an old-fashioned rising column of Mercury thermometer or barometer.

The **XmNsliderMark** resource controls the graphic that appears inside the slider. Figure 5-1 illustrates all the available slider marks.

Figure 5–1. Different Slider Marks Inside a Scale



Two convenience routines, **XmScaleGetValue** and **XmScaleSetValue**, allow the application to get and set the slider value.

5.4.3 Tic Marks

By default, a Scale has no labels or tic marks along the rectangle in which the slider moves. The application can add these by calling **XmScaleSetTicks**.

5.5 List

A List is an array of textual items from which the user selects one or more entries. Each item is a compound string. List has four modes, controlled by the **XmNselectionPolicy** resource, for selecting items:

Single Select	At most one item is selected. Performing the selection action on an item toggles the selection state of the item and deselects any other selected item.
Browse Select	At most one item is selected. Performing the selection action on an item selects the item and deselects any other selected item. Dragging Btn1 through the list moves the selection along with the cursor.

Multiple Select	Any number of items can be selected. Performing the selection action on an item toggles the selection state of the item but does not deselect any other selected item.
Extended Select	Any number of items can be selected. The user can select either continuous or discontinuous ranges of items, depending on the mouse buttons used or, when using the keyboard, on whether the List is in Normal Mode or Add Mode.

When the user makes a selection, the List invokes one of four callback lists, depending on the selection policy:

Selection Policy	Callback List
Single Select	**XmNsingleSelectionCallback**
Browse Select	**XmNbrowseSelectionCallback**
Multiple Select	**XmNmultipleSelectionCallback**
Extended Select	**XmNextendedSelectionCallback**

By default, the List does not invoke a callback list when the List is in Browse Select or Extended Select mode and the user drags the mouse cursor over a new item. It does invoke the callbacks when the user releases the mouse button. If **XmNautomaticSelection** is True, the List invokes the callbacks while the user is dragging the mouse.

The widget data passed to selection callback routines contains both the selected items—the compound strings—and integers representing the positions within the list of the selected items. The first item in the list is at position 1, the second item at position 2, and so on.

List has another callback list, **XmNdefaultActionCallback**, which it invokes when the user double clicks or presses osfActivate on an item. The widget data passed to these callback routines contains only the item at the location cursor and its position, not the selected items. When the user performs the default action via a double click, the List calls the appropriate selection callbacks on the first click and the **XmNdefaultActionCallback** callbacks on the second click.

List includes several other sets of resources:

- Arrays and counts of the List items and selected items
- The number of items, **XmNvisibleItemCount**, that the list can display at one time, and the position in the List of the first visible item
- Several resources that affect the appearance of the list items: render table, justification (**XmNstringDirection**), spacing between items, and margins between the items and the List border
- The maximum time interval between clicks for a double click
- A resource (**XmNlistSizePolicy**) that determines what the List does when an item is too wide to fit into the List: it can keep its size and, if it is a ScrolledList, add a horizontal ScrollBar; grow to accommodate the item; or try to grow and, if it fails to accommodate the item but is a ScrolledList, add a ScrollBar
- A resource that determines whether the ScrollBars in a ScrolledList are displayed at all times or only when needed

A ScrolledList is a List inside a ScrolledWindow. The application can use **XmCreateScrolledList** to create one.

In addition to its resources, List has a variety of convenience routines that allow the application to add, remove, select, and deselect items; specify the first or last visible item; find the position of an item or the positions of the selected items; set Add Mode; and scroll the List horizontally.

5.6 Text and TextField

The Text widget displays and, optionally, edits text. When the widget is editable and the user presses a key that represents a text character, that character is inserted into the text. Other translations and actions allow the user to navigate or to select, cut, copy, paste, or scroll the text.

The Text widget is used to edit simple text, which can only use a single font or fontset. The TextField widget is a text editor optimized for editing a single line of text. It also only uses simple text. This widget is largely used for entering commands and filenames.

For more information on Text and TextField, see Chapter 10.

Chapter 6

Menus and Options

A menu is a widget that allows the user to make a choice among actions or states. When the menu is visible, the user makes a choice by activating a button in the menu, usually by pressing Btn1, osfSelect, or osfActivate on the button. Some buttons also have mnemonics that allow the user to activate them by pressing the mnemonic keys when the menu is visible. Buttons can also have accelerators, which activate the buttons whether or not the menu is visible.

Like any other widget, a menu must be controlled by some parent widget. However, some types of Motif menu may be shared by more than one widget. (Only one widget will be the parent, but the same menu can be accessed from any other widget on the menu's "post-from" list.)

Motif has four basic kinds of menu:

- MenuBar. This menu is normally always managed within some component of an application, often the MainWindow. It usually consists of a row of CascadeButtons. When the user activates a button in the menu, a PulldownMenu menu appears with one set of top-level choices that apply to the application component.

- PopupMenu. This menu contains a set of choices that apply to a component of the application. The menu is not visible until the user takes an action that posts it, usually pressing Btn3 in the associated component or pressing osfMenu when the component has keyboard focus. A PopupMenu can contain buttons that take action or change state directly. It can also contain CascadeButtons that cause PulldownMenus to appear.
- PulldownMenu. This menu is associated with a CascadeButton in a MenuBar, a PopupMenu, or another PulldownMenu. The menu is not visible until the user posts it by activating the associated CascadeButton. Like a PopupMenu, a PulldownMenu can contain buttons that take action or change state directly. It can also contain CascadeButtons that cause other PulldownMenus to appear.
- OptionMenu. This menu allows the user to choose among one set of choices, usually mutually exclusive attributes or states. It consists of a label, a selection area, and a PulldownMenu. The selection area is a CascadeButtonGadget whose label shows the currently selected option. The PulldownMenu contains the set of options. The user posts the PulldownMenu by activating the CascadeButtonGadget or by pressing **MAlt** along with a mnemonic. When the user activates a button in the PulldownMenu, that button becomes the newly selected option.

RowColumn is the widget that Motif uses as a menu. A RowColumn can also be a nonmenu WorkArea. One use for a WorkArea is to contain a set of ToggleButtons constituting a RadioBox or a CheckBox. When the user selects a ToggleButton, its state changes from on to off or from off to on. In a RadioBox, only one ToggleButton at a time can be on; in a CheckBox, more than one ToggleButton can be on.

RowColumn performs special geometry management to align and lay out its children in a variety of ways. An application can use a RowColumn WorkArea to take advantage of the RowColumn geometry management for a set of widgets. For details see Chapter 14.

In addition to menus, users can also select choices by interacting with a ComboBox or a SpinBox.

A ComboBox is a combination of a List child and a TextField child. This combination gives users two ways to select a choice. That is, the user can either click on a displayed item in the List or type the choice directly into the TextField.

A SpinBox displays a combination of arrows and one or more textual widgets (usually Labels and TextFields). The user makes selections by clicking on the arrows. The arrow clicking increments or decrements the values displayed in the textual widgets. For example, a SpinBox would be an excellent widget to use when the user needs to select a date. The user could click on SpinBox arrows to adjust month and day fields.

6.1 Menu Components: Buttons, RowColumn, MenuShell

A menu is a three-level hierarchy:

- Buttons represent the menu selections.
- A RowColumn widget is the manager that contains the buttons.
- A MenuShell envelops each PulldownMenu and PopupMenu.

6.1.1 Buttons

The user makes a choice in a menu by activating one of the buttons in the menu. CascadeButtons, PushButtons, and ToggleButtons and their gadget variants are most commonly used in menus.

Note: Motif does not support DrawnButtons or ArrowButtons in menus, though they can appear in a RowColumn WorkArea. To give a menu button a distinctive appearance, use a PushButton with a label type of **XmPIXMAP** and supply **XmNlabelPixmap** and **XmNlabelInsensitivePixmap** resources.

The application learns of the user’s choice through the appropriate button callback lists:

- When the user activates a CascadeButton, the button calls the **XmNcascadingCallback** callbacks. If the button has an attached PulldownMenu after these callbacks return, the button posts the menu. Otherwise, the button calls the **XmNactivateCallback** callbacks.
- When the user activates a PushButton, the button calls the **XmNactivateCallback** callbacks.
- When the user activates a ToggleButton, the button calls the **XmNvalueChangedCallback** callbacks.

Buttons in a menu have translations and actions that arm, disarm, and activate the buttons. These actions also post and unpost menus in the hierarchy at appropriate times. The buttons inherit menu traversal translations and actions from **XmLabel**. These actions allow the user to move from button to button within a menu and from menu to menu within the menu hierarchy.

Use the **XmNenableEtchedInMenu** resource of **XmDisplay** to help control the shadowing of activated menu buttons.

6.1.2 RowColumn

The parent of the buttons in a menu is a RowColumn widget. RowColumn interacts with its button children in these ways:

- In a menu (but not a WorkArea), it ensures that all children are CascadeButtons, PushButtons, ToggleButtons, Labels, or Separators (or their gadget variants). If the **XmNisHomogeneous** resource is True, it ensures that all children are of the class specified by **XmNentryClass**.
- It lays out its children and, if **XmNisAligned** is True, aligns the labels of children that are **XmLabel** or **XmLabelGadget** subclasses.
- It stores the widget ID of the last menu item selected in the **XmNmenuHistory** resource.
- It allows the application to supply a single callback list for all button children. If **XmNentryCallback** is not NULL, it disables the **XmNactivateCallback** and **XmNvalueChangedCallback** callbacks for its button children and arranges for the buttons to call the **XmNentryCallback** callbacks instead.
- If **XmNradioBehavior** is True, it ensures that only one ToggleButton at a time is normally selected. It also changes the default values for **XmNindicatorType** and **XmNvisibleWhenOff** for its ToggleButton children to the one-of-many, always-displayed style.
- It has additional resources for MenuBars and OptionMenus, described in the following sections.

In addition to **XmNentryCallback**, RowColumn also has **XmNmapCallback** and **XmNunmapCallback** callbacks. These callbacks apply only to PopupMenus and PulldownMenus. The **XmNmapCallback** callbacks are called just before the menu is posted, and the **XmNunmapCallback** callbacks are called just after the menu is unposted. They are useful for changing the menu to reflect the current state of the application. For example, an **XmNmapCallback** callback can use **XtSetSensitive** to make some menu items insensitive if they are not applicable in the current state of the program.

6.1.3 MenuShell

The windows associated with PopupMenus and PulldownMenus are top-level windows. That is, the parent window of such a menu is the root window of the screen, not the window associated with the parent widget. This allows the menu to appear anywhere on the screen without being clipped by the parent widget's window.

The parent widget of each PopupMenu and PulldownMenu RowColumn must be a MenuShell. It is actually the MenuShell's window that is the top-level window. **XmMenuShell** is a subclass of OverrideShell, so the window manager ignores MenuShell's windows.

A MenuShell is often invisible to the application. The Motif convenience routines for creating PopupMenus and PulldownMenus automatically create MenuShell parents for these menus. When a PulldownMenu is the child of a PopupMenu or another PulldownMenu, the child's MenuShell is actually the child of the parent's MenuShell. The convenience routines for creating PulldownMenus manage these relations automatically.

Motif arranges for the RowColumn's window to coincide with the MenuShell's window. Setting **XmNheight**, **XmNwidth**, or **XmNborderWidth** for either a MenuShell or its child sets that resource to the same value in both the parent and the child. For a child of a MenuShell, setting **XmNx** or **XmNy** sets the corresponding resource of the parent but does not change the child's position relative to the parent. **XtGetValues** for the child's **XmNx** or **XmNy** yields the value of the corresponding resource in the parent. The x and y coordinates of the child's upper left outside corner relative to the parent's upper left inside corner are both zero minus the value of **XmNborderWidth**.

To change any geometry-related resources of a PopupMenu or PulldownMenu, an application should always specify these resources for the RowColumn child, not the MenuShell parent.

If an application needs to create a MenuShell explicitly, it should create the MenuShell as a popup child of its parent (using **XtCreatePopupShell** or **XtVaCreatePopupShell**). All Motif convenience routines that create MenuShells do this automatically, and an application rarely needs to create a MenuShell directly.

6.2 MenuBar

All children of a MenuBar must be CascadeButtons or CascadeButtonGadgets. The MenuBar attempts to place its button children in a single row. If it does not have enough room, it tries to wrap the remaining children into additional rows.

An application should treat specially the button, if any, that pulls down a help menu. The application should set the MenuBar RowColumn's **XmNmenuHelpWidget** to the widget ID of this button. The MenuBar attempts to place this button at one of the lower corners of the MenuBar, as specified by the *CDE 2.1/Motif 2.1—Style Guide and Glossary*.

In a MenuBar, all buttons typically have associated PulldownMenus. Each PulldownMenu associated with a button in a MenuBar must be a child of the MenuBar. (More precisely, each PulldownMenu's MenuShell must be a child of the MenuBar.) Each button's **XmNsubMenuId** resource must be set to the widget ID of the associated PulldownMenu. Set **XmNsubMenuId** to the widget ID of the PulldownMenu RowColumn, not of the PulldownMenu's MenuShell.

The routines **XmCreateMenuBar**, **XmCreateSimpleMenuBar**, and **XmVaCreateSimpleMenuBar** all create MenuBars.

6.3 PopupMenu

A PopupMenu is normally invisible. When the user takes some action—usually pressing Btn3 or osfMenu—in a widget that has a PopupMenu, the menu is posted. The user moves from item to item in the menu by dragging Btn3 or, when keyboard traversal is enabled, by keyboard traversal actions. Motif unposts the menu when the user activates an item in the menu system (other than a CascadeButton), presses osfCancel, or releases or clicks Btn3 outside a menu item.

A PopupMenu RowColumn must have a MenuShell parent. The parent of the MenuShell is the widget with which the PopupMenu is associated. Because the MenuShell is a popup child of its parent, the parent can be any widget (but not a gadget); it does not have to be a subclass of **Composite**. The Motif convenience routines that create PopupMenus automatically create a MenuShell as the parent of the PopupMenu RowColumn.

Several different widgets may share a given Popup menu. One widget must serve as the parent of the MenuShell, but other widgets may be nominated with the **XmAddToPostFromList**(3) function.

The PopupMenu's **XmNmenuPost** resource specifies the button event that posts the menu. This event will make the menu visible in any widget on the menu's list of eligible widgets. The event must be a button press, possibly with modifiers. However, you should not set **XmNmenuPost** to **BTransfer Press** because many button widgets use **BTransfer Press** as a default binding to initiate a Drag operation.

When a user creates a PopupMenu, Motif installs an event handler on the menu's widget parent. Called **PopupMenuEventHandler**, this routine performs most of the necessary setup for a PopupMenu, and manages the RowColumn widget as well.

On receipt of an event that should post a PopupMenu, this event handler searches the widget hierarchy and the popup list for the appropriate menu to post. The appropriate menu will have its **XmNmenuPost** resource match the triggering event, will have its **XmNpopupEnabled** resource set to **XmPOPUP_AUTOMATIC** or **XmPOPUP_AUTOMATIC_RECURSIVE**, and will have been created before any other PopupMenu that satisfies the same criteria. If the popup menu is found in a parent of the target widget, the **XmNpopupEnabled** resource must be set to **XmPOPUP_AUTOMATIC_RECURSIVE**.

Note: In older versions of Motif, developers were required to write their own event handlers in order to implement PopupMenus. In the upgrade to version 2.0, substantial changes were made to the popup menu system to simplify the creation and management of PopupMenus. However, applications using this approach will still work correctly under the current version of Motif.

Once a menu selection has been made, control shifts to the **popupHandlerCallback** procedure in the menu's parent widget, if there is one. The callback allows a more specific menu to be posted, if necessary, or can even perform the selected menu function, if desired.

A pointer to the following structure is passed to each callback for **XmNpopupHandlerCallback**:

```
typedef struct
{
        int     reason;
        XEvent  * xevent;
        Widget  menuToPost;
        Boolean postIt;
        Widget  target;
} XmPopupHandlerCallbackStruct;
```

reason	Indicates why the callback was invoked. **XmCR_POST** implies that this is a regular posting request, while **XmCR_REPLAY** indicates that the menu was just unposted, and that this callback was invoked on a replay.
xevent	Points to the *XEvent* that triggered the handler.
menuToPost	Specifies the PopupMenu that the menu system is to post. The application may modify this field.
postIt	Indicates whether the posting process should continue. The application may modify this field.
target	The target widget or gadget.

Posting a PopupMenu through the keyboard is controlled by the PopupMenu’s **XmNmenuAccelerator** and **XmNpopupEnabled** resources. **XmNmenuAccelerator** specifies a key event that may post the menu. **XmNpopupEnabled** specifies whether or not this event actually posts the menu. It also determines whether or not accelerators and mnemonics in the PopupMenu and its submenus are enabled.

An application can have only one active PopupMenu at a time for a particular widget. If the widget has more than one PopupMenu, the application should set **XmNpopupEnabled** to True for the active menu and set **XmNpopupEnabled** to False for all inactive menus.

6.4 PulldownMenu

A PulldownMenu is always associated with another RowColumn. It becomes visible when the user activates a CascadeButton in the associated RowColumn. It becomes invisible when the user traverses upward or laterally in the menu hierarchy, activates a button in the hierarchy (other than a CascadeButton in the menu or a descendant), presses osfCancel, or clicks or releases a mouse button outside a menu item.

A PulldownMenu must have the following relations with other widgets:

- It must be the value of the **XmNsubMenuId** resource of the CascadeButton that is to post the menu.
- It must have a MenuShell as its parent. The Motif convenience routines that create PulldownMenus create MenuShell parents automatically. As with the PopupMenu, several different widgets may share a given Pulldown menu. One widget must serve as the parent of the MenuShell, but other widgets may be nominated with the **XmAddToPostFromList**(3) function.
- The MenuShell must have the proper parent, depending on the kind of RowColumn with which the PulldownMenu is associated. The MenuShell is a popup child of its own parent. Following are the required parents of the MenuShell:
  - — If the PulldownMenu is to be pulled down from a MenuBar, the parent must be the MenuBar.
  - — If the PulldownMenu is to be pulled down from a PopupMenu or another PulldownMenu, the parent must be that PopupMenu or PulldownMenu. Actually, the parent is the other menu's MenuShell; but the *parent* parameter to the Motif convenience routines that create PopupMenus must be the other menu itself (the RowColumn), not its MenuShell parent.
  - — If the PulldownMenu is to be pulled down from an OptionMenu, the parent must be the parent of the OptionMenu.

6.5 OptionMenu

An OptionMenu lets the user choose among a set of usually mutually exclusive options. The OptionMenu is always visible. It consists of a label (a LabelGadget), a selection area (a CascadeButtonGadget), and an associated PulldownMenu. The label of the CascadeButtonGadget displays the currently selected option, one of the items in the PulldownMenu. When the user activates the CascadeButtonGadget, the PulldownMenu becomes visible with the currently selected item directly above the selection area. When the user activates an item in the PulldownMenu, the PulldownMenu is unposted and the item the user chose becomes the currently selected option.

The PulldownMenu normally contains only PushButtons. It must not contain any ToggleButtons, and Motif does not support CascadeButtons.

RowColumn has a number of resources for use specifically with an OptionMenu:

XmNlabelString
The text of the label. Setting this resource also sets the **XmNlabelString** of the LabelGadget.

XmNmnemonic
A keysym that, when pressed along with the **MAlt** modifier, posts the PulldownMenu. Motif underlines the first character in the label string that matches the mnemonic and that is in a segment whose font list element tag matches **XmNmnemonicCharSet**. Setting this resource also sets the **XmNmnemonic** of the LabelGadget.

XmNmnemonicCharSet
The font list element tag used for underlining the mnemonic. Setting this resource also sets the **XmNmnemonicCharSet** of the LabelGadget.

XmNsubMenuId
The widget ID of the PulldownMenu. Setting this resource also sets the **XmNsubMenuId** of the CascadeButtonGadget.

If the application needs to get or set any of these four resources for the LabelGadget or CascadeButtonGadget, it should always get or set it in the OptionMenu RowColumn, not the gadget itself. To get or set other resources for the gadgets, the application should use **XmOptionLabelGadget** or **XmOptionButtonGadget** and then call **XtGetValues** or **XtSetValues** on the returned widget ID. A user or application can also specify resource values in resource files by using the names of the gadgets, "OptionLabel" and "OptionButton".

Setting the **XmNmenuHistory** resource also has a special effect in OptionMenus. Setting **XmNmenuHistory** to an item in the PulldownMenu makes that item the currently selected option. It updates the label of the CascadeButtonGadget and causes the PulldownMenu to appear, when posted, with the selected item over the CascadeButtonGadget.

XmCreateOptionMenu creates an OptionMenu RowColumn and its LabelGadget and CascadeButtonGadget children. It does not create the associated PulldownMenu.

The following example creates a simple OptionMenu with three options:

```
Widget          parent, pulldown, option, pb1, pb2, pb3;
Arg             args[10];
Cardinal        n;
...
n = 0;
pulldown = XmCreatePulldownMenu(parent, "option_pd",
                                  args, n);
pb1 = XmCreatePushButtonGadget(pulldown, "option_pb1",
                                  args, n);
pb2 = XmCreatePushButtonGadget(pulldown, "option_pb2",
                                  args, n);
pb3 = XmCreatePushButtonGadget(pulldown, "option_pb3",
                                  args, n);
XtSetArg(args[n], XmNsubMenuId, pulldown);       n++;
XtSetArg(args[n], XmNmenuHistory, pb2);          n++;
option = XmCreateOptionMenu(parent, "option_rc", args, n);
...
```

The following application-class defaults file provides labels and mnemonics for an English-language locale:

```
*option_pb1.labelString:   Option 1
*option_pb2.labelString:   Option 2
*option_pb3.labelString:   Option 3
*option_rc.labelString:   Options
*option_rc.mnemonic:   O
```

6.6 RadioBox and CheckBox

RadioBoxes and CheckBoxes are collections of ToggleButtons. The primary difference is that in a RadioBox only one ToggleButton at a time can be set; in a CheckBox more than one ToggleButton can be set. In a RadioBox, a ToggleButton cannot be in an indeterminate state; in a CheckBox, a ToggleButton can be in an indeterminate state.

RadioBoxes and CheckBoxes are usually implemented as WorkAreas, though it is possible to implement them as menus. Usually the application intends for the box to remain visible after the user sets a ToggleButton, particularly in a CheckBox. The application can implement a transient RadioBox or CheckBox by placing a WorkArea inside a dialog.

The following RowColumn resources specifically control the behavior of a RadioBox or CheckBox:

XmNradioBehavior
When True, the RowColumn ensures that at most one ToggleButton is set at a time. Setting this resource to True also causes the ToggleButton resource **XmNindicatorType** to default to **XmONE_OF_MANY** and **XmNvisibleWhenOff** to default to True.

XmNradioAlwaysOne
When both this resource and **XmNradioBehavior** are True, RowColumn ensures that one ToggleButton is always set. The user is not allowed to unset a ToggleButton when no other ToggleButton is set.

For a RadioBox implemented as a WorkArea, the default value for **XmNisHomogeneous** is True, and by default RowColumn allows only ToggleButton and ToggleButtonGadget children.

Note that the application can foil the RowColumn's enforcement of **XmNradioBehavior** and **XmNradioAlwaysOne**, even when these resources are True. The application can use **XtSetValues** to set the state of the ToggleButtons, and it can manage and unmanage ToggleButtons regardless of their state. The behavior of a RadioBox is undefined if the application takes actions that contradict **XmNradioBehavior** or **XmNradioAlwaysOne**.

XmCreateRadioBox creates a WorkArea RadioBox and initializes **XmNradioBehavior** to True.

A CheckBox is most often a collection of ToggleButtons in a WorkArea with **XmNradioBehavior** set to False. By default, the ToggleButton **XmNindicatorType** is **XmN_OF_MANY** and **XmNvisibleWhenOff** is True.

6.7 TearOffMenus

An application can allow the user to "tear off" a PulldownMenu or PopupMenu. When the user tears off a menu, Motif unposts that menu and any posted menu descendants.

It gives the menu a TransientShell parent and then maps the parent as a top-level window. The torn-off menu has window-manager decorations, and its title can be specified with the **XmNtearOffTitle** resource of the RowColumn menu widget. If the title is not so specified, it will be the same as the label of the CascadeButton that posts the menu in the original menu system.

The user can interact with the torn-off menu just as in the menu hierarchy. When the user activates buttons in a torn-off menu, the actions take effect but the torn-off menu remains posted. When the user takes an action that unposts the torn-off menu, such as pressing osfCancel, the menu returns to its original position in the menu hierarchy. If the user reposts the original menu from the menu hierarchy while the torn-off menu is posted, an inactive representation of the torn-off menu remains visible, but the menu itself is unposted and then reposted within the menu hierarchy.

When a menu in a menu system can be torn off, a distinctive tear-off button appears at the beginning of the menu. The user can tear off the menu by activating the tear-off button as with any other button in the menu. The user can also tear off the menu by pressing Btn2 in the tear-off button. The user can then drag the torn-off menu to another position on the screen and fix its position by releasing Btn2.

Menus cannot be torn off by default. The application must allow the user to tear off a menu by setting the RowColumn resource **XmNtearOffModel** to **XmTEAR_OFF_ENABLED**. When the user tears off a menu, the **XmNtearOffMenuActivateCallback** callbacks are invoked just before the **XmNmapCallback** callbacks. When the user unposts a torn-off menu, the **XmNtearOffMenuDeactivateCallback** callbacks are invoked just after the **XmNunmapCallback** callbacks.

6.8 ComboBox

The ComboBox widget manages both a List widget and a TextField widget. This combination of List and TextField gives users two ways to choose an item. The user can choose an item either by selecting one of the entries displayed by the List widget or by typing the selection directly into the TextField widget.

By default, the ComboBox displays both the TextField widget and the List widget. However, it is possible to hide the List widget so that it will only be displayed when the user requests it. If the List is hidden, the user can make it visible by clicking on a displayed arrow.

The TextField can either be editable or non-editable. If the TextField is non-editable, the user must pick one of the choices displayed by the List.

6.8.1 ComboBox Types

Motif provides three types of ComboBox widgets. The widget type is specified using the **XmNcomboBoxType** resource on ComboBox. The possible settings for this resource are as shown in Table 6-1.

Table 6–1. XmNcomboBoxType Resource Values

Value	**TextField**	**List Widget**
XmCOMBO_BOX	Editable	Always displayed
XmDROP_DOWN_COMBO_BOX	Editable	Hidden
XmDROP_DOWN_LIST	Non-editable	Hidden

XmDROP_DOWN_LIST is the default value.

ComboBoxes with hidden Lists are usually more desirable when screen space is limited. ComboBoxes with hidden Lists are also recommended when users are more likely to enter text into the TextField than to choose an item from the List.

6.8.2 Creating and Manipulating ComboBox

Motif provides the following three convenience functions for creating ComboBoxes:

- **XmCreateComboBox**
- **XmCreateDropDownComboBox**
- **XmCreateDropDownList**

Each function creates an instance of a ComboBox widget and returns its associated widget ID.

Each function requires an *arglist* parameter. The resource values specified in *arglist* will be passed not only to the ComboBox, but to the List and TextField as well. Thus, an application can specify resources for the List such as **XmNitems** and **XmNvisibleItemCount** by including them in the *arglist* parameter passed to the ComboBox.

The List part of the ComboBox expects an array of compound strings to fill the list. If an array is not passed to the List at creation time, the application must provide this array later. Each string becomes an item in the list, with the first string becoming the item in position 1, the second string becoming the item in position 2, and so on. Your application can obtain the **XmList** widget ID by specifying ***List** as an argument to **XtNameToWidget( )**. Similarly, the TextField widget in the ComboBox can be accessed by passing ***Text** to **XtNameToWidget( )**. You can then call a convenience function, such as **XmListAddItem**, to add more List items. You can also access the **List** and **TextField** widgets through the **XmNlist** and **XmNtextField** resources, but these resources are read-only.

If the TextField widget is editable, the user can type directly in the text field to enter a selection. If the application wishes to validate the entered text, it can do so by installing the **XmNmodifyVerifyCallback** on the TextField widget directly.

The following example illustrates how to create a drop-down ComboBox containing five items. It illustrates how to use the XmCreateComboBox( )function, and how to pass its children resources at creation time. This example uses **XtNameToWidget( )**to manipulate the List child, setting the List child’s **XmNvisibleItemCount** to 5.

```
{
#define NUM_LIST_ITEMS 5
   Widget          comboBox;
   Arg             args[10];
   Cardinal        n, i;
   XmString        ListItem[NUM_LIST_ITEMS];
   static char    *ListString[] = { "kiwi",
                                    "raspberry",
                                    "carambola",
                                    "litchi",
                                    "coconut" };
   /* Create a list of XmStrings for the ComboBox List child */
     for (i=0; i < NUM_LIST_ITEMS; i++)
       ListItem[i] = XmStringCreate (ListString[i], XmSTRING_DEFAULT_CHARSET);


   /* Create a ComboBox of type XmDROP_DOWN_COMBO_BOX. */
   /* Resources passed to ComboBox are passed on to the
    * children of ComboBox.  So, in the argument list
    * below, the resources, XmNitems, and XmNitemCount
    * will be passed on to the List child of ComboBox.  */
   n=0;
   XtSetArg (args[n], XmNcomboBoxType, XmDROP_DOWN_COMBO_BOX); n++;
   XtSetArg (args[n], XmNarrowSpacing, 5); n++;
   XtSetArg (args[n], XmNitems, ListItem); n++;
   XtSetArg (args[n], XmNitemCount, NUM_LIST_ITEMS); n++;
   comboBox = XmCreateComboBox (parent, "ComboBox", args, n);
   XtManageChild (comboBox);
   XtAddCallback (comboBox, XmNselectionCallback, SelectionCB,
      (XtPointer)NULL);

   /* Example of manipulating a child widget directly to set the
    * visibleItemCount on the list.  */
   n=0;
   XtSetArg (args[n], XmNvisibleItemCount, 5); n++;
   XtSetValues (XtNameToWidget (comboBox,"*List"), args, n);
}
```

6.8.3 ComboBox Items

You may add and manipulate Combobox items using the functions

- **XmComboBoxAddItem**
- **XmComboBoxSelectItem**
- **XmComboBoxSetItem**
- **XmComboBoxDeletePos**

All of these functions take a widget ID that specifies the ComboBox; none returns a value.

- **XmComboBoxAddItem** takes an **XmString** specifying the new item and assigns to the item a position in the list.
- **XmComboBoxDeletePos** deletes an item by reference to its position.
- **XmComboBoxSelectItem** selects an item in the **XmList** of a ComboBox widget.
- **XmComboBoxSetItem** causes a specified item to be the first visible item in the list.

6.8.4 Controlling the Arrow

If a ComboBox has a hidden List, then the ComboBox displays an arrow. The arrow will always be displayed adjacent to the TextField. However, the placement of this arrow depends on the **XmNlayoutDirection** resource of the VendorShell (or subclass of) in which the ComboBox is contained, as follows:

- If the value of **XmNlayoutDirection** is **XmLEFT_TO_RIGHT**, ComboBox places the arrow to the right of the TextField field.
- If the value of **XmNlayoutDirection** is **XmRIGHT_TO_LEFT**, ComboBox places the arrow to the left of the TextField field.

Use **XmNarrowSize** to set the requested size of the arrow. Use **XmNarrowSpacing** to set the spacing between the arrow and the TextField.

Note that **XmNarrowSize** is a size request, not a size guarantee. For example, if the requested arrowSize is larger than the current size of the ComboBox, ComboBox will have to make a geometry request to its parent. Whether the geometry request is wholly, partially, or not accepted depends on the parent. If this request is accepted, the arrow size request can be granted. If the ComboBox's request to grow is only partially granted or not granted at all, ComboBox will make the arrow size the maximum that will fit inside the ComboBox's allowed size.

6.8.5 ComboBox Matching Behavior

If the ComboBox does not have an editable TextField, any text the user types will not appear in the TextField. Instead, typing text may cause a matching algorithm to be invoked that attempts to match the entered text with an item in the list. If a match is found, the item is selected, and the item appears in the TextField. Whether a specific matching algorithm is applied or not is determined by the value of the **XmNmatchBehavior** resource on ComboBox. The **XmNmatchBehavior** resource can accept two values: **XmNONE** and **XmQUICK_NAVIGATE**. A value of **XmNONE** indicates that no matching algorithm will be invoked; **XmQUICK_NAVIGATE** indicates that when the List widget has focus, a one-character navigation is supported. In this algorithm, if the typed character is the initial character of some item in the **XmList**, that item is navigated to and selected. Subsequently typing the same character will cycle among items with the same first character.

6.8.6 ComboBox Callbacks

When a selection occurs in ComboBox, the **XmNselectionCallback** callbacks are called. For example, the code fragment appearing earlier in this section established a selection callback procedure named **SelectionCB**. Here is the code for that callback:

```
void
SelectionCB (Widget w, XtPointer client_data, XtPointer call_data)
{
 XmComboBoxCallbackStruct *cb = (XmComboBoxCallbackStruct *)call_data;
 XmStringCharSet        charset;
 XmStringDirection      direction;
 XmStringContext        context;
 Boolean                separator;
 char                  *item;

   /* This callback procedure prints the item that was just selected. */

   /* Retrieve the selected text string from XmString. */
     item = XmStringUnparse(cb->item_or_text, NULL, XmCHARSET_TEXT,
                            XmCHARSET_TEXT, NULL, 0, XmOUTPUT_ALL);
     printf ("ComboBox SelectionCB: item = %s\n\n", item);
}
```

ComboBox allows a single item to be selected in many ways, including through a matching behavior. A list item can more directly be selected by scrolling to it with either the mouse or keyboard, and selecting the item directly. ComboBox supports the Browse Select selection method of **XmList**. If the ComboBox has an editable text field, the selection can be typed directly into the TextField entry field. Regardless of the selection mechanism used, when an item in the list is selected, the list item is highlighted by displaying it in reverse colors, and the selected item is displayed in the TextField field in the ComboBox.

Additionally, if the user performs an action that moves focus away from the ComboBox, the *XmNselectionCallbacks* are called to notify the application of the user's choice. The *item_or_text* field in the call data structure passed to the callback will contain the contents of the TextField field at the time of the callback. The application then takes whatever action is required for the specified selection.

The **XmComboBoxUpdate** function resynchronizes the internal data structures of a specified ComboBox widget. This function is useful when an application manipulates ComboBox's child widgets, possibly changing data structures. For example, you might want to use the **XmComboBoxUpdate** function after a ComboBox List child selection policy has been changed without notification.

6.9 SpinBox

A SpinBox lets a user select one choice from a set of mutually exclusive options. A SpinBox contains an increment and a decrement ArrowButton, which allow the user to change the displayed choice. You might think of a SpinBox as a "ring" of choices that operates similar to a digital clock. It displays choices consecutively, and, at the last item, wraps around to the first (or, going in the other direction, wraps from the first item to the last). As long as the user holds down an ArrowButton, the current position in the ring of choices changes continuously, with a "spinning" effect.

To control when spinning begins, set the **XmNinitialDelay** resource to specify the time, in milliseconds, that elapses before the displayed value changes. To control the rate of spinning, set the **XmNrepeatDelay** resource to specify the time, in milliseconds, that elapses between subsequent changes in the displayed value.

A SpinBox may include traversable text children, which display the choices, and label and separator children, which are used for informative or decorative purposes only.

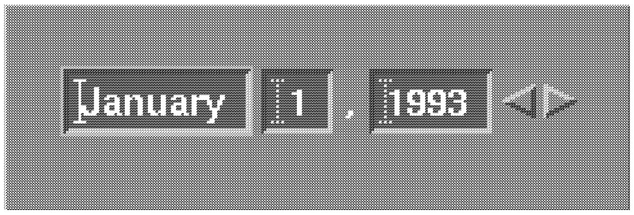
There are two types of traversable text children allowed in a SpinBox: numeric and string.

- For a numeric child, define the choices by passing in integers for a minimum value, a maximum value, and an incremental value. By setting the constraint resource, **XmNdecimalPoints**, you can display decimal values. The increment arrow increases the displayed SpinBox value by the given incremental value. The decrement arrow decreases the displayed value by the given incremental value.
- For a string child, provide an array of compound strings, using **XmStringTable**. The increment arrow displays the next string, moving toward the end of the array. The decrement arrow displays the previous string, moving toward the beginning of the array.

To create a SpinBox with one ring of choices, use **XmCreateSpinBox** and create a child (either numeric or string) using **XmCreateText** or **XmCreateTextField**.

The SpinBox, however, may manage multiple traversable text children, as illustrated by the SpinBox in Figure 6-1. Three traversable text children display months, days, and years. A decoration child (the comma) separates the displayed day and year. The two ArrowButtons are part of every SpinBox.

Figure 6–1. A SpinBox with Multiple Children



Because the text children are traversable, the user can tab from one field to the next to change focus. Only one child at a time can take focus. The ArrowButtons spin only choices in the text field with focus. It is possible to "chain" the choices so that spinning choices in one text field affects what is displayed in other text fields. For example, when a user spins the day child in Figure 6-1 up to 31, **XmNvalueChangedCallback** is called for each change in position. When the user spins from 31 to 1, the **XmNvalueChangedCallback** function can use **XtSetValues** to change the **XmNposition** resource for the month child from 1 to 2 so that February is displayed.

February has fewer days than January. Therefore, when the month changes from January to February, the **XmNmodifyVerifyCallback** procedure must change the **XmNmaximumValue** of the day child from 31 to 28 (or 29 if leap year). If the year rolls the calendar backward from February to January, then the **XmNmodifyVerifyCallback** procedure must reset the **XmNmaximumValue** of the day child back to 31.

SpinBox provides two callbacks:

XmNmodifyVerifyCallback

Whenever an ArrowButton is pressed but before the position of the SpinBox changes, **XmNmodifyVerifyCallback** is invoked. This callback lets you control the next SpinBox position or perform other actions before the SpinBox value changes. The callback function can modify two members of the callback structure: *doit* and *position*. Set these as follows:

- To prevent the spinning action, set *doit* to False.
- To allow normal consecutive spinning, do not change *doit* or *position*.
- To spin to another value in the ring of choices, change *position* and leave *doit* set to True.

When *doit* is False, the SpinBox position and value do not change. You would set *doit* to False if you wanted the SpinBox to stop at the first or last choice without wrapping around the ring of choices. When *doit* is True, spinning continues to the position returned in the callback structure, and **XmNvalueChangedCallback** is invoked.

XmNvalueChangedCallback

Whenever an ArrowButton is pressed and the SpinBox position has just changed, **XmNvalueChangedCallback** is invoked. This callback lets you perform any action based on the change in the callback structure members, *position* and *value*. In addition to returning a reason, this callback indicates the event that triggered the callback, which child is affected, and the SpinBox position. For **XmSTRING** children, *value* contains the displayed string.

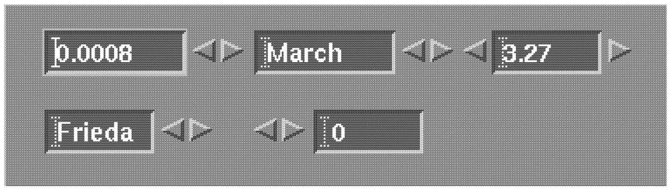
The SpinBox widget also includes translations to arm and disarm the SpinBox, increase and decrease the SpinBox position, and go to the first and last position.

The following SpinBox resources control arrow geometry, sensitivity and response:

XmNdefaultArrowSensitivity
: Specifies whether one or both ArrowButtons are sensitive (perform spinning) or insensitive (displayed as stippled and do not respond) by default. This allows you to turn off an arrow button; for example, you may want to stipple the increment arrow when the SpinBox reaches the last or maximum choice. The constraint resource **XmNarrowSensitivity** can override this resource setting for a particular child.

XmNarrowLayout
: Sets the ArrowButton layout. Arrows can be displayed side by side, either to the left or to the right of a text child, or split at either end of the text child. They may also be displayed one on top of the other, before or after the children; or side by side, before or after the children. Arrow layout depends on **XmNlayoutDirection**. When layout direction is left to right, beginning arrows are positioned to the left. When layout direction is right to left, beginning arrows are positioned to the right. Figure 6-2 illustrates the arrow layouts when **XmNlayoutDirection** is left to right. The first two SpinBoxes (**.0008** and **March**) show layout at the end of the child. The third SpinBox (**3.27**) shows a split layout. The last SpinBox (**0**) shows layout at the beginning of the child.

Figure 6–2. SpinBox Arrow Layouts



XmNarrowOrientation
: Sets the orientation of the ArrowButtons: vertical or horizontal.

XmNarrowSize
: Sets both the width and height of the two ArrowButtons.

XmNinitialDelay
: Sets the length of time the mouse button can be held down before automatic spinning begins. This resource and the **XmNrepeatDelay** resource allow you to control the rate of spinning.

XmNrepeatDelay
Sets a delay between each change in the SpinBox choice position.

The following constraints affect each SpinBox traversable child individually:

XmNarrowSensitivity
Sets the sensitivity of one or both ArrowButtons for a specific traversable child. This resource overrides the SpinBox **XmNdefaultArrowSensitivity** resource.

XmNspinBoxChildType
Specifies whether the ring of choices is of numeric or string type.

XmNposition
Specifies the current position in the range of valid numbers or in the string array. The interpretation of this resource depends on the **XmNpositionType** resource.

XmNpositionType
Specifies how the value in the **XmNposition** resource is to be interpreted: as its index or as the data value.

XmNvalues
Defines an array of strings for a SpinBox string type child.

XmNnumValues
The number of strings in the **XmNvalues** array.

XmNminimumValue
Sets the minimum value for a SpinBox numeric type child.

XmNmaximumValue
Sets the maximum value for a SpinBox numeric type child.

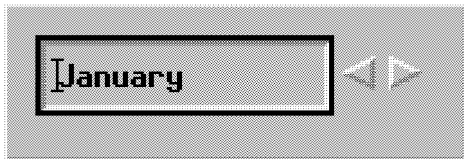
XmNincrementValue
Sets the value by which to raise or lower the displayed value in a SpinBox numeric type child.

XmNdecimalPoints
Specifies the number of decimal places to display for a SpinBox numeric type child.

If the Core resources **XmNwidth** and **XmNheight** are not specified, the SpinBox widget attempts to grow to accommodate large arrows or long text fields. It does not shrink if arrows or children are very small.

The following code creates a SpinBox with one string type child, as shown in Figure 6-3. The choices are values from January to December. This code invokes only the **XmNvalueChangedCallback**, which is called after the displayed value has changed.

Figure 6–3. A SpinBox with One Child



```
...
/*****  Create SpinBox parent  *****/
n = 0;
XtSetArg(argList[n], XmNy, nextY); n++;

spin0 = XmCreateSpinBox(parent, "spin0", argList, n);

/*****  Create XmString array of month names  *****/
setMonths();

/*****  Create TextField child  *****/
n = 0;
XtSetArg(argList[n], XmNvalues, monthValues); n++;
XtSetArg(argList[n], XmNnumValues, 12); n++;
XtSetArg(argList[n], XmNspinBoxChildType, XmSTRING); n++;
XtSetArg(argList[n], XmNselectionPolicy, XmSINGLE_SELECT); n++;
XtSetArg(argList[n], XmNeditable, False); n++;

spin0_text = XmCreateTextField(spin0, "spin0_text", argList, n);

/*****  Manage SpinBox  *****/
XtManageChild(spin0);

/*****  Call changedSpin0 AFTER displayed value has changed  *****/
XtAddCallback(spin0, XmNvalueChangedCallback, changedSpin0, (XtPointer) 0);

/*****  Manage SpinBox child  *****/
XtManageChild(spin0_text);
```

```
...
```

The **setMonths** routine creates an **XmString** array of month names; **setMonths** consists of the following code:

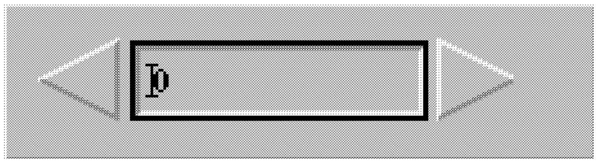
```
void
setMonths(void)
{
XmString tempString;
int      monthLoop;

    for (monthLoop = 0; monthLoop < NUM_MONTHS; monthLoop++) {
        tempString = XmStringCreate(months[monthLoop],
                                    XmFONTLIST_DEFAULT_TAG);
        monthValues[monthLoop] = tempString;
        }
}
```

The following callback code for **XmNvalueChangedCallback** modifies the variable *thisYear* whenever the boundary is crossed.

```
void
changedSpin0(Widget w, XtPointer client, XtPointer call)
{
XmSpinBoxCallbackStruct *user;
static int              thisYear = 1994;

    user = (XmSpinBoxCallbackStruct *)call;

    if (user->crossed_boundary)
        {
        if (user->reason == XmCR_SPIN_NEXT)
          thisYear++;
        else
          thisYear--;
        }
}
```

The code for the SpinBox shown in Figure 6-4 includes an *XmModifyVerifyCallback* to change resources and prevent wrapping below the minimum value.

Figure 6–4. A SpinBox that Does Not Wrap



```
...
/*****  Create SpinBox parent  *****/
n = 0;
XtSetArg(argList[n], XmNarrowSize, 35); n++;
XtSetArg(argList[n], XmNrepeatDelay, 250); n++;
XtSetArg(argList[n], XmNinitialDelay, 500); n++;
XtSetArg(argList[n], XmNarrowLayout, XmARROWS_SPLIT); n++;

spin4 = XmCreateSpinBox(parent, "spin4", argList, n);

n = 0;
XtSetArg(argList[n], XmNmaximumValue, 4); n++;
XtSetArg(argList[n], XmNspinBoxChildType, XmNUMERIC); n++;

spin4_text = XmCreateTextField(spin4, "spin4_text",
                               argList, n);


/*****  Manage SpinBox  *****/
XtManageChild(spin4);

/*****  Call modifySpin4 BEFORE displayed value is changed  *****/
XtAddCallback(spin4, XmNmodifyVerifyCallback, modifySpin4,
              (XtPointer) 0);

/*****  Manage SpinBox child  *****/
XtManageChild(spin4_text);
...
```

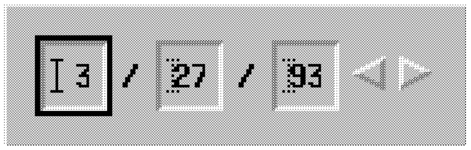
The callback code for the previous call to **XmNmodifyVerifyCallback** uses *doit* to stop the SpinBox from wrapping at the minimum value:

```
void
modifySpin4(Widget w, XtPointer client, XtPointer call)
{
XmSpinBoxCallbackStruct *user;
int                     newHigh;
Cardinal                n;
Arg                     argList[5];

    user = (XmSpinBoxCallbackStruct *)call;

    if (user->crossed_boundary)
        {
        if (user->reason == XmCR_SPIN_NEXT)
            {
            n = 0;
            XtSetArg(argList[n], XmNmaximumValue, &newHigh); n++;
            XtGetValues(user->widget, argList, n);

            newHigh++;

            n = 0;
            XtSetArg(argList[n], XmNmaximumValue, newHigh); n++;
            XtSetValues(user->widget, argList, n);
            user->position = 0;
            }
        else if (user->reason == XmCR_SPIN_PRIOR)
                user->doit = False;
        }
}
...
```

The SpinBox shown in Figure 6-5 has multiple children, including two decoration children. In addition, this SpinBox includes ’chaining’, the process where a change in one child causes values to change in the child and on or more other children. Chaining is performed by the valueChanged callback.

The code for this SpinBox is as follows.

Figure 6–5. A SpinBox with Multiple, Chained Children



```
...
/*****  Create SpinBox parent  *****/
n = 0;
XtSetArg(argList[n], XmNy, nextY); n++;
XtSetArg(argList[n], XmNinitialDelay, 0); n++;
XtSetArg(argList[n], XmNrepeatDelay, 150); n++;

spin6 = XmCreateSpinBox( parent, "spin6", argList, n );


/*****  Increment Y position for next SpinBox  *****/
nextY +=  Y_OFFSET;

/*****  Create SpinBox child  *****/
n = 0;
XtSetArg(argList[n], XmNwidth, 30); n++;
XtSetArg(argList[n], XmNposition, thisMM - 1); n++;
XtSetArg(argList[n], XmNminimumValue, 1); n++;
XtSetArg(argList[n], XmNmaximumValue, 12); n++;
XtSetArg(argList[n], XmNspinBoxChildType, XmNUMERIC); n++;

spin6_text1 = XmCreateTextField( spin6, "spin6_text1",
                                 argList, n );


/*****  Create SpinBox decoration child  *****/
n = 0;
decoString = XmStringCreateLtoR("/", XmSTRING_DEFAULT_CHARSET);
XtSetArg(argList[n], XmNlabelString, decoString); n++;

spin6_deco1 = XmCreateLabel(spin6, "spin6_deco1", argList, n);

/*****  Create SpinBox child  *****/
```

```
n = 0;
XtSetArg(argList[n], XmNwidth, 30); n++;
XtSetArg(argList[n], XmNposition, thisDD - 1); n++;
XtSetArg(argList[n], XmNminimumValue, 1); n++;
XtSetArg(argList[n], XmNmaximumValue, 31); n++;
XtSetArg(argList[n], XmNspinBoxChildType, XmNUMERIC); n++;

spin6_text2 = XmCreateTextField( spin6, "spin6_text2",
                                 argList, n );

/*****  Create SpinBox decoration child  *****/
n = 0;
decoString = XmStringCreateLtoR("/", XmSTRING_DEFAULT_CHARSET);
XtSetArg(argList[n], XmNlabelString, decoString); n++;
spin6_deco2 = XmCreateLabel( spin6, "spin6_deco2", argList, n );


XmStringFree(decoString);

/*****  Create SpinBox child  *****/
n = 0;
XtSetArg(argList[n], XmNwidth, 30); n++;
XtSetArg(argList[n], XmNposition, thisYY); n++;
XtSetArg(argList[n], XmNmaximumValue, 99); n++;
XtSetArg(argList[n], XmNspinBoxChildType, XmNUMERIC); n++;


spin6_text3 = XmCreateTextField(spin6, "spin6_text3", argList, n);

/*****  Manage SpinBox  *****/
XtManageChild(spin6);

/*****  Call changedSpin6 AFTER displayed value has changed  *****/
XtAddCallback(spin6, XmNvalueChangedCallback, changedSpin6,
              (XtPointer) 0);

/*****  Manage SpinBox children  *****/
XtManageChild(spin6_text1);
XtManageChild(spin6_deco1);
XtManageChild(spin6_text2);
XtManageChild(spin6_deco2);
XtManageChild(spin6_text3);
...
```

The callback code for the previous call to **XmNvalueChangedCallback** uses **XtSetValues** calls to chain the day and the month children, and the month and the year children. It also changes **XmNmaximumValue** for the day child whenever the month child changes, as follows:

```
...
void
changedSpin6(Widget w, XtPointer client, XtPointer call)
{
XmSpinBoxCallbackStruct *user;
Cardinal                n;
Arg                     argList[5];
int                     saveMM;
int                     saveDD;
int                     saveYY;

    user = (XmSpinBoxCallbackStruct *)call;

    if (user->widget == spin6_text1)
        {
        thisMM = user->position + 1;

        if (thisDD <=3)
            setMaxDay(spin6_text2, thisMM -1);
        else if (thisDD > 27)
            setMaxDay(spin6_text2, thisMM);

        saveYY = thisYY;

        if (user->crossed_boundary)
            if (user->reason == XmCR_SPIN_NEXT)
                thisYY++;
            else
                thisYY--;

        if (thisYY != saveYY)
```

```
                {
                if (thisYY < 0)
                    thisYY = 99;
                else if (thisYY > 99)
                    thisYY %= 100;

                n = 0;
                XtSetArg(argList[n], XmNposition, thisYY);
                n++;

                XtSetValues(spin6_text3, argList, n);
                }
            }
        else if (user->widget == spin6_text2)
            {
            thisDD = user->position + 1;

            if (thisDD <=3 && user->reason == XmCR_SPIN_PRIOR)
                setMaxDay(spin6_text2, thisMM -1);
            else if (thisDD > 27 && user->reason == XmCR_SPIN_NEXT)
                setMaxDay(spin6_text2, thisMM);

            saveMM = thisMM;
            saveYY = thisYY;

            if (user->crossed_boundary)
                if (user->reason == XmCR_SPIN_NEXT)
                    thisMM++;
                else
                    thisMM--;

            if (thisMM != saveMM)
                {
                if (thisMM < 1)
                    {
                    thisMM = 12;
                    thisYY--;
                    }
                else if (thisMM > 12)
                    {
```

```
                thisMM = 1;
                thisYY++;
                }

            n = 0;
            XtSetArg(argList[n], XmNposition, thisMM - 1);
            n++;

            XtSetValues(spin6_text1, argList, n);

            if (thisYY != saveYY)
                {
                n = 0;
                XtSetArg(argList[n], XmNposition, thisYY);
                n++;

                XtSetValues(spin6_text3, argList, n);
                }
            }
        }
    else if (user->widget == spin6_text3)
        {
        thisYY = user->position;

        if (user->reason == XmCR_OK)
            {
            if (thisDD <= 3)
                setMaxDay(spin6_text2, thisMM - 1);
            else if (thisDD > 27)
                setMaxDay(spin6_text2, thisMM);
            }
        }
}
```

The **XmSpinBoxValidatePosition** function is available for use by applications that need to implement a policy for tracking user modifications to editable SpinBox children of type **XmNUMERIC**. It is up to the application to specify when and how the user modifications take effect.

6.10 SimpleSpinBox

The **XmSimpleSpinBox** widget, a subclass of **XmSpinBox**, is a user interface control to increment and decrement an arbitrary TextField. For example, it can be used to cycle through the months of the year or days of the month.

The XmSimpleSpinBox widget redefines some of its XmSpinBox superclass constraint resources as normal resources for the class: **XmNdecimalPoints**, **XmNincrementValue**, **XmNmaximumValue**, **XmNminimumValue**, **XmNnumValues**, **XmNposition**, **XmNpositionType**, **XmNspinBoxChildType**, **XmNvalues**. Widget subclassing is not supported for the XmSimpleSpinBox widget class.

SimpleSpinBox provides the following functions:

- **XmCreateSimpleSpinBox** — creates an instance of SimpleSpinBox.
- **XmSimpleSpinBoxAddItem** — takes an **XmString** specifying the new item and assigns to the item a position in the list.
- **XmSimpleSpinBoxDeletePos** — deletes an item by reference to its position.
- **XmSimpleSpinBoxSetItem** — causes a specified item to be the first visible item in the list.

Chapter 7

Dialogs

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

Dialogs are similar to menus. Both seek input from the user. Like PopupMenus and PulldownMenus, dialogs appear in top-level windows and are more or less transient. Making a selection typically unposts a PopupMenu or PulldownMenu and often pops down a dialog. "PopupMenu" "in DialogBoxes" "PulldownMenu" "in DialogBoxes" There are two chief differences:

- Unless torn off, menus are usually *modal*; that is, the user must make a selection from the menu or unpost it before interacting with other parts of the application. Dialogs can be either modal or *modeless*. In a modeless dialog, the user can interact with other parts of the application before returning to the dialog.
- Menu components are limited to buttons, labels, and separators. Dialogs can contain other, sometimes arbitrary, kinds of widgets, such as List and Text. Dialogs permit more complex interaction with the user and allow the application to solicit a broader range of information.

Menus are well suited to allowing the user to make a single choice from a constrained set. Dialogs are appropriate for displaying information about a transient or unusual state of the program and for obtaining complex input from the user. Whether to use a dialog or a menu is not always clear. In fact, a TearOffMenu combines aspects of both. For more information on using menus and dialogs, see the *CDE 2.1/Motif 2.1—Style Guide and Glossary*.

7.1 BulletinBoard and DialogShell

From the application’s point of view, a dialog must meet the following criteria:

- A dialog is a widget that is a subclass of **XmBulletinBoard**.
- A dialog’s parent is a DialogShell.

BulletinBoard is intended to be the usual superclass for a dialog widget. The dialog widget can be either a BulletinBoard itself or one of its more specialized subclasses. BulletinBoard is a container with no automatically created children; it supplies general behavior needed by most dialogs. Its subclasses provide child widgets and specific behavior tailored to particular types of dialogs.

BulletinBoard and its subclasses can also function outside a DialogShell, as part of the application’s main window. One subclass, Form, is particularly useful in providing constraint-based geometry management for a collection of child widgets.

7.1.1 BulletinBoard

BulletinBoard provides the following resources and behavior:

- Activation and cancellation of the dialog. BulletinBoard installs accelerators for osfActivate and osfCancel. Unless focus is in another button, osfActivate activates the **XmNdefaultButton** if it is sensitive. The osfCancel activates the **XmNcancelButton** if it is sensitive. Subclasses set the **XmNdefaultButton** and **XmNcancelButton**.
- A resource, **XmNdialogStyle**, that determines whether the dialog is modal or modeless. Three modal styles exist:

primary application modal
: Among the dialog and its ancestors, input goes only to the dialog, but the user can interact with other parts of the application or with other applications.

full application modal
: Within the application, input goes only to the dialog, but the user can interact with other applications.

system modal
: Input goes only to the dialog; the user cannot interact with other applications. System modal should seldom be used.

- Callbacks invoked when the BulletinBoard is mapped and unmapped and when it gains input focus.
- Geometry-management resources and class methods that implement several resizing policies and that allow the BulletinBoard to interact with its subclasses in managing complex collections of descendant widgets. The geometry-related resources are **XmNmarginHeight**, **XmNmarginWidth** and **XmNresizePolicy**. For more information on BulletinBoard's geometry management, see Chapter 10.

7.1.2 Activation, Cancellation, and Help

Often a dialog has one or more actions, associated with buttons, that apply to the dialog as a whole. Some common actions are "activate," "cancel," and "help." BulletinBoard deals specially with activation and cancellation. BulletinBoard allows the user to "activate" or "cancel" the dialog from anywhere within the BulletinBoard (except, in the case of activation, when a button has the focus).

BulletinBoard has a resource, **XmNdefaultButton**, whose value is a button descendant that represents the default activation action. The dialog renders the default button so that it looks somewhat different from the other buttons. Use the **XmNdefaultButtonEmphasis** resource of **XmDisplay** to change the highlighting of this default button. When the user presses osfActivate in a button that has keyboard focus, that button's osfActivate actions are called. If the user presses osfActivate and no button has focus, BulletinBoard calls the osfActivate actions for **XmNdefaultButton** if it is sensitive. If the user presses osfActivate in a List, Text, or TextField descendant, the osfActivate actions for that widget are invoked first, and then BulletinBoard calls the osfActivate actions for **XmNdefaultButton**.

BulletinBoard has another resource, **XmNcancelButton**, whose value is a button descendant that represents the default cancellation action. When the user presses osfCancel anywhere within the BulletinBoard, BulletinBoard calls the osfActivate actions for **XmNcancelButton** if it is sensitive.

The help action works differently. Often the application represents help for the dialog as a whole by providing a Help button. When the user activates this button, the application provides help for the dialog. In general, the application can provide help through an **XmNactivateCallback** procedure for the Help button. Some BulletinBoard subclasses create Help buttons automatically. These widgets add a procedure to the Help button’s **XmNactivateCallback** list that invokes the dialog’s **XmNhelpCallback** procedures when the Help button is activated. In these cases, the application can provide help through the dialog’s **XmNhelpCallback** procedures.

If the user presses osfHelp elsewhere in the BulletinBoard, this action usually invokes the **XmNhelpCallback** callbacks for the widget with the focus. If this widget has no **XmNhelpCallback** procedures, Motif looks up the widget hierarchy for the first ancestor with a non-NULL **XmNhelpCallback** list and invokes those procedures. By providing an **XmNhelpCallback** procedure for the dialog itself, the application can ensure that the user sees help for the dialog as a whole when the descendant widget with focus has no help information of its own.

7.1.3 DialogShell

DialogShell is the Motif shell widget that contains dialogs. It is a subclass of TransientShell, which is a subclass of VendorShell. DialogShell inherits much of VendorShell’s behavior in interacting with the window manager and in providing geometry management for off-the-spot input methods.

DialogShell cooperates extensively with BulletinBoard, and some of DialogShell’s features for containing dialogs assume that its child is a BulletinBoard or BulletinBoard subclass. Often the application does not need to deal directly with the DialogShell at all. The Motif convenience routines that create dialogs automatically create a DialogShell as the popup child of the parent shell.

To pop up the dialog, the application does not call **XtPopup** on the DialogShell, but instead manages the child of the DialogShell. DialogShell’s **change_managed** procedure pops up the dialog when the child is managed and pops it down when the child is unmanaged, providing that the child’s **XmNmappedWhenManaged** resource is True. If a BulletinBoard child’s **XmNautoUnmanage** resource is initialized to True, the BulletinBoard is automatically unmanaged when its OK and cancel buttons are activated.

DialogShell notifies its BulletinBoard child by using the **XmNmapCallback** and **XmNunmapCallback** procedures when the child is about to be mapped and unmapped.

Like VendorShell, DialogShell ensures that when no off-the-spot input method exists the DialogShell window remains coincident with the child window. Setting **XmNx** and **XmNy** for the child sets these resources for the shell, without changing the child’s position relative to the child. Setting **XmNheight**, **XmNwidth** and **XmNborderWidth** for the child usually sets these resources to the same value in the DialogShell. When a BulletinBoard child is managed with its **XmNdefaultPosition** resource set to True, DialogShell centers the dialog with respect to the parent.

BulletinBoard has two resources that allow the user or application to customize a parent DialogShell’s interaction with the window manager. **XmNdialogTitle** provides a title for the window manager, and **XmNnoResize** determines whether or not the dialog MWM frame includes resize controls. To affect other aspects of interaction with the window manager, the user or application must set the appropriate DialogShell resources.

XmCreateBulletinBoardDialog creates a BulletinBoard and a parent DialogShell.

7.1.4 Initial Focus

When the **XmNkeyboardFocusPolicy** of a shell is **XmEXPLICIT**, Motif uses the Manager resource **XmNinitialFocus** in determining which component of a manager receives initial focus in these circumstances:

- When the manager is the child of a shell and the shell hierarchy receives focus for the first time
- When focus is inside the shell hierarchy, the manager is a composite tab group, and the user traverses to the manager using the keyboard

Following are the default values of **XmNinitialFocus** for BulletinBoard and its subclasses:

- For BulletinBoard, Form, and MessageBox, the default is the value of **XmNdefaultButton**
- For SelectionBox and its subclasses, the default is the text edit area

7.2 Making a Selection: SelectionBox

SelectionBox is a BulletinBoard subclass that generally allows the user to select an item from a list. By default, a SelectionBox includes the following children:

- A scrolling list of alternatives
- An editable text field for the selected alternative
- Labels for the list and text field
- Three or four buttons

The default buttons are OK, Cancel, and Help. By default, an Apply button is also created. If the parent of the SelectionBox is a DialogShell, it is managed; otherwise, it is unmanaged.

An application can add additional children to the SelectionBox. The first child is used as a work area. The value of **XmNchildPlacement** determines whether the work area is placed above or below the Text area, or above or below the List area. Additional children are laid out in the following manner:

MenuBar	The first MenuBar child is placed at the top of the window.
Buttons	All **XmPushButton** widgets or gadgets and their subclasses are placed after the OK button in the order of their creation.
Others	The layout of additional children that are not in these categories is undefined.

The user can select an item in two ways: by scrolling through the list and selecting the desired item or by entering the item name directly into the text edit area. Selecting an item from the list causes that item name to appear in the selection text edit area. SelectionBox installs accelerators, the value of **XmNtextAccelerators**, on the text edit widget. The default accelerators bind osfUp, osfDown, osfBeginLine, osfEndLine, and osfRestore events in the text edit widget to SelectionBox actions that select an item in the list and replace the text edit widget value with that list item.

SelectionBox provides **XmNokCallback**, **XmNcancelCallback**, **XmNhelpCallback**, and **XmNapplyCallback** lists, which the SelectionBox invokes when the corresponding button is activated. Activation of the OK button may invoke either the **XmNokCallback** list or the **XmNnoMatchCallback** list. When the user activates the OK button and either the **XmNmustMatch** resource is False or the text in the text edit area matches a list item, SelectionBox invokes the **XmNokCallback** procedures. When the user activates the OK button, **XmNmustMatch** is True, and the text in the text edit area does not match a list item, SelectionBox invokes the **XmNnoMatchCallback** procedures.

SelectionBox has two subclasses, FileSelectionBox and Command, which are described in later sections. **XmCreateSelectionDialog** creates a standard SelectionBox and a DialogShell parent. **XmCreatePromptDialog** creates a variant SelectionBox dialog containing a text edit area and label and OK, Cancel, and Help buttons. A PromptDialog has an unmanaged Apply button, and it has no List or List label. It is intended for the application to prompt the user for brief text input.

The **XmNdialogType** resource determines which of the standard SelectionBox children are created and managed. The value usually depends on the application's use of the SelectionBox:

- **XmDIALOG_SELECTION** usually indicates a standard SelectionBox dialog.
- **XmDIALOG_WORK_AREA** indicates a SelectionBox outside a DialogShell. The Apply button is unmanaged.
- **XmDIALOG_PROMPT** indicates a PromptDialog.
- **XmDIALOG_COMMAND** indicates a Command subclass.
- **XmDIALOG_FILE_SELECTION** indicates a FileSelectionBox subclass.

SelectionBox has resources for supplying text, label strings, and list items for its children. The widget IDs of the children of a SelectionBox and its subclasses are not available as resources. The application can retrieve the widget IDs of the automatically created children by using **XtNameToWidget** or by calling one of the convenience routines Motif provides for this purpose: **XmSelectionBoxGetChild**, **XmFileSelectionBoxGetChild**, and **XmCommandGetChild**.

7.3 Choosing a Pathname: FileSelectionBox

FileSelectionBox is a subclass of SelectionBox designed for finding and selecting files. By default, a FileSelectionBox contains the same children as a standard SelectionBox, with the addition of a second ScrolledList and a second text edit area. Unlike a SelectionBox, the default Apply button in a FileSelectionBox is labeled "Filter" in a C locale.

One of the text areas, the directory mask area, holds a directory mask specifying a base directory to be searched and a search pattern. However, if the **XmNpathMode** resource is set to **XmPATH_MODE_RELATIVE**, FileSelectionBox splits this text area into two separate TextFields. One of these TextFields holds the base directory to be searched and the other holds the search pattern.

Another text area, the selection area, holds the name of the selected file.

One of the Lists, the directory list, displays the subdirectories of the current base directory. The other List, the file list, displays all the files, subdirectories, or both in the base directory that match the search pattern. The **XmNfileListItems** resource holds the selected items in this list.

The **XmNfileFilterStyle** resource determines whether or not hidden files (those starting with a period, usually) will be displayed in the directory and file lists.

The user can select a new base directory to examine by scrolling through the list of directories and selecting the desired directory or by editing the directory mask. Selecting a new directory from the directory list does not change the search pattern. A user can select a new search pattern by editing the directory mask. Double clicking or pressing osfActivate on a directory in the directory list initiates a search for files and subdirectories in the new directory, using the current search pattern.

Activating the Filter button, the directory list, or the directory mask text area causes the FileSelectionBox to initiate a file search. The FileSelectionBox uses three procedures, each the value of a resource, in conducting the search: **XmNqualifySearchDataProc**, **XmNdirSearchProc** and **XmNfileSearchProc**. The **XmNqualifySearchDataProc** procedure extracts the base directory and the search pattern from the directory mask. The **XmNdirSearchProc** procedure uses the data returned by **XmNqualifySearchDataProc** to update the directory list. The **XmNfileSearchProc** procedure uses the data returned by **XmNqualifySearchDataProc** to update the file list.

The user can select a file by scrolling through the list of filenames and selecting the desired file or by entering the filename directly into the text edit area. Selecting a file from the list causes that filename to appear in the file selection text edit area. The user confirms the selection by activating the OK button, the file list, or the selection text area.

FileSelectionBox uses the SelectionBox callback lists to notify the application when the user activates one of the buttons. The application can also provide one or more of the three procedures that FileSelectionBox uses to conduct a search. For a specification of the input to and output from these routines, see the **XmFileSelectionBox**(3) reference page in the *Motif 2.1—Programmer's Reference*.

The application can remove the directory list, the file list, or both. The application must unmanage the ScrolledWindow parent of the List and the corresponding label. An application can also add additional children to a FileSelectionBox, which manages any additional children in the same way as SelectionBox.

XmCreateFileSelectionDialog creates a FileSelectionBox and a parent DialogShell.

7.4 Command

Command is a SelectionBox subclass intended for entering a command. It contains the SelectionBox text edit area, List, and List label, but no buttons. The application can add only one additional work area child to the Command. A Command usually appears as part of the application's main window rather than as a dialog.

The user specifies a command by adding text to the text area or by selecting an item from the List, which represents the command history. Whenever the text edit area changes, Command invokes the **XmNcommandChangedCallback** procedures. The user enters a command by activating the List or the text edit area. When the user enters a command, Command appends the command to the history list and invokes the **XmNcommandEnteredCallback** procedures.

Command has a number of resources that are aliases for SelectionBox resources dealing with the List and text edit area. Command also has an **XmNhistoryMaxItems** resource, which specifies the maximum length of the history list. After the list reaches this length, Command deletes the first item in the list before appending a newly entered command.

7.5 MessageBox

MessageBox is a BulletinBoard subclass intended for a dialog consisting of a single user interaction. By default, a MessageBox has the following components:

- A LabelGadget with a pixmap label symbolizing the type of interaction the MessageBox represents
- A LabelGadget with a compound string label representing the text of the message
- A SeparatorGadget separating the message symbol and text from the other children
- Three buttons: OK, Cancel, and Help

Typically the message symbol and text are on top and the buttons on the bottom, with the separator between. The application can add additional children to a MessageBox. Additional children are laid out in the following manner:

- The first MenuBar child is placed at the top of the window.
- All **XmPushButton** widgets or gadgets, and their subclasses, are placed after the OK button in the order of their creation.
- A child that is not in these categories is treated as a work area and is placed above the row of buttons. If a message label exists, the child is placed below the label. If a message pixmap exists, but a message label is absent, the child is placed on the same row as the pixmap. The child behaves as a work area and grows or shrinks to fill the space above the row of buttons. The layout of multiple work area children is undefined.

Several convenience routines create MessageBox widgets with DialogShell parents for particular kinds of interactions. For most of these routines, the principal difference in the type of MessageBox they create is that each uses a distinct default symbol pixmap. When it creates the symbol pixmap, MessageBox uses **XmGetPixmapByDepth** to find a pixmap with a name that corresponds to the type of interaction. Each dialog type is also associated with a value of the **XmNdialogType** resource. The following table shows the correspondence between creation routine, **XmNdialogType**, and symbol pixmap name:

Table 7–1. MessageBox Routines, Dialog Types, and Pixmaps

Convenience Routine	**XmNdialogType**	**Pixmap Name**
XmCreateErrorDialog	**XmDIALOG_ERROR**	**xm_error**
XmCreateInformationDialog	**XmDIALOG_INFORMATION**	**xm_information**
XmCreateMessageDialog	**XmDIALOG_MESSAGE**	
XmCreateQuestionDialog	**XmDIALOG_QUESTION**	**xm_question**
XmCreateTemplateDialog	**XmDIALOG_TEMPLATE**	
XmCreateWarningDialog	**XmDIALOG_WARNING**	**xm_warning**
XmCreateWorkingDialog	**XmDIALOG_WORKING**	**xm_working**

A MesssageDialog and a TemplateDialog have no default symbol pixmap. A TemplateDialog is a special MessageBox variant that is intended for application customization and that, by default, has no children except the separator.

Like SelectionBox, MessageBox has **XmNokCallback**, **XmNcancelCallback**, and **XmNhelpCallback** lists to inform the application when the user activates a button. MessageBox has resources for supplying label strings and the symbol pixmap for its children. The widget IDs of the children of a MessageBox are not available as resources. The application can retrieve the widget IDs of the automatically created children by using **XtNameToWidget** or by calling **XmMessageBoxGetChild**.

7.6 Form

Form is a BulletinBoard subclass whose main purpose is to provide constraint-based geometry management for arbitrary children. Form has a number of constraint resources that it uses to place children with respect to the Form, positions within the form, and other children. Most Form-specific behavior is related to this geometry management. Form has no default children of its own. But as a BulletinBoard subclass, Form is an appropriate container for use in dialogs. **XmCreateFormDialog** creates a Form and a DialogShell parent.

For information on Form’s geometry management, see Chapter 14.

Chapter 8

Scrolling, Panes, Frames, Containers, and Notebooks

Chapters 6 and 7 discuss the Motif Manager widgets used to construct menus and dialogs. Motif also provides more general-purpose managers intended for use in main application windows and some dialogs. This chapter discusses widgets that perform the following functions:

- Establishing a viewport for a larger underlying scroll
- Providing a main application window with a combination of standard and custom components
- Presenting information about objects to users using different views
- Organizing child widgets as either a page or component to access or describe pages
- Placing a shadowed frame around a widget and an optional title at the top
- Creating multiple subwindows for a composite with adjustable boundaries between the subwindows

8.1 ScrolledWindow

Frequently a collection occupies an area that is too large to display within an application or that may grow or shrink as the user adds or deletes data. Examples include text in a Text widget, items in a List, and graphical objects in a DrawingArea or other canvas. Three approaches exist for handling this problem:

- Set a fixed size for the widget. The disadvantage of this approach is that when the collection grows beyond the bounds of the widget, part of the collection is not visible.
- Allow the widget to make geometry requests to expand or contract, perhaps up to some maximum or down to some minimum size. The disadvantages of this approach are that it may disrupt the application's visual layout and that the widget is able to grow only within limits, perhaps not at all.
- Treat the collection as a virtual scroll, with the widget acting as a (more or less) fixed-size viewport onto the scroll. The user can move the viewport to expose obscured portions of the scroll.

The ScrolledWindow widget implements the last approach. It is a Manager with one or two ScrollBar children, a child widget that acts as the virtual scroll, and in some cases another child that acts as a viewport onto the scroll. By using the ScrollBars or keyboard scrolling commands, the user moves the viewport to expose part of the scroll.

8.1.1 Automatic and Application-Defined Scrolling

ScrolledWindow implements two scrolling models: automatic and application defined.

In automatic scrolling, the application creates a widget to serve as the virtual scroll, and the ScrolledWindow creates the ScrollBars and a widget to serve as a fixed-size viewport onto the scroll. The application adjusts the size of the scroll widget as necessary to contain the entire collection. The ScrolledWindow adjusts the appropriate ScrollBar resources so that the size and position of the slider reflect the position of the viewport in relation to the scroll and the proportion of the scroll's entire size that the viewport represents. The ScrolledWindow also handles the user's interaction with the ScrollBars, moving the viewport in relation to the scroll as the user manipulates the ScrollBars. Usually the application need have no interaction with the ScrollBars or the widget that serves as the viewport.

In application-defined scrolling, the application must create the ScrollBars as well as the widget that acts as the virtual scroll and, if necessary, a separate viewport widget. The application must determine how large to make the viewport widget and what portion of the data to display in the viewport. The application handles all interaction with the ScrollBars. It must adjust the appropriate ScrollBar resources if it wants the size and position of the slider to reflect the relation of the viewport to the underlying scroll. It must also move the viewport in relation to the scroll as the user interacts with the ScrollBars.

The ScrolledWindow resource **XmNscrollingPolicy** determines the scrolling model. Possible values for this resource are **XmAUTOMATIC** and **XmAPPLICATION_DEFINED**. The default value is **XmAPPLICATION_DEFINED**.

8.1.2 Other Resources

In addition to **XmNscrollingPolicy**, ScrolledWindow has two sets of resources.

One set of resources holds the components of the ScrolledWindow. An application usually does not have to set any of these resources; the ScrolledWindow examines the class and other characteristics of each child as it is created and sets the appropriate resource. If the application needs to supply a new ScrollBar or scroll widget after creating the initial component, it can use either **XtSetValues** or **XmScrolledWindowSetAreas**.

Following are the resources that hold components of the ScrolledWindow:

XmNclipWindow
The value is the ID of the viewport widget created by the ScrolledWindow in automatic scrolling. This resource applies only when the **XmNscrollingPolicy** is **XmAUTOMATIC**. It is a read-only resource; the application cannot set a new value.

XmNhorizontalScrollBar
The value is the ID of the horizontal ScrollBar. The ScrolledWindow creates this ScrollBar and sets the value of this resource when

the **XmNscrollingPolicy** is **XmAUTOMATIC**. In application-defined scrolling, the application must create and manage the ScrollBar, but the ScrolledWindow automatically sets the value of this resource to its widget ID.

XmNverticalScrollBar
The value is the ID of the vertical ScrollBar. The ScrolledWindow creates this ScrollBar and sets the value of this resource when the **XmNscrollingPolicy** is **XmAUTOMATIC**. In application-defined scrolling, the application must create and manage the ScrollBar, but the ScrolledWindow automatically sets the value of this resource to its widget ID.

XmNworkWindow
The value is the ID of the widget that serves as the scroll. The application has to create and manage this widget, but it usually does not have to set this resource. When the application creates a child of the ScrolledWindow that is not a ScrollBar, the ScrolledWindow automatically sets the value of this resource to its widget ID.

The second set of resources specifies the layout of the ScrolledWindow:

XmNscrollBarDisplayPolicy
This resource determines whether the ScrolledWindow always displays managed ScrollBars or displays them only when the corresponding dimensions of the scroll exceed those of the viewport. Possible values are **XmAS_NEEDED** and **XmSTATIC**. The value is forced to **XmSTATIC** when the scrolling policy is **XmAPPLICATION_DEFINED** and defaults to **XmAS_NEEDED** when the scrolling policy is **XmAUTOMATIC**.

XmNscrollBarPlacement
This resource determines where the ScrolledWindow places the horizontal and vertical ScrollBars. The possible values are constants that specify on which sides of the viewport the ScrolledWindow places the two ScrollBars: **XmTOP_LEFT**, **XmTOP_RIGHT**, **XmBOTTOM_LEFT**, and **XmBOTTOM_RIGHT**.

XmNscrolledWindowMarginHeight
This resource specifies the margins between the top and bottom sides of the ScrolledWindow and the first child on each side.

XmNscrolledWindowMarginWidth
This resource specifies the margins between the left and right sides of the ScrolledWindow and the first child on each side.

XmNspacing
This resource specifies the distance between each ScrollBar and the viewport.

8.2 Automatic Scrolling

In the automatic scrolling model, the ScrolledWindow creates a fixed-size viewport and handles all interaction with the ScrollBars. The application usually needs to take only the following steps:

- Create and manage a ScrolledWindow, supplying a value of **XmAUTOMATIC** for **XmNscrollingPolicy** in the argument list passed to the creation function
- Create and manage a widget child of the ScrolledWindow to serve as the scroll
- Adjust the size of the scroll widget, typically using **XtSetValues** of **XmNheight** and **XmNwidth**, as necessary to contain all the data in the scroll

The ScrolledWindow automatically creates a widget to serve as the viewport and sets **XmNclipWindow** to the ID of this widget. It also creates horizontal and vertical ScrollBars and sets **XmNhorizontalScrollBar** and **XmNverticalScrollBar** to the appropriate IDs of the ScrollBars. The ScrolledWindow attaches callback procedures to the ScrollBars to handle user interaction with the ScrollBars.

The ScrolledWindow sets the ScrollBar resource **XmNincrement** to a small fraction of the height or width of the viewport. It sets the ScrollBar resource **XmNpageIncrement** to a large fraction of the height or width of the viewport. If the ScrolledWindow resizes the viewport, it recomputes the values of these resources.

The ScrolledWindow sets the ScrollBar resources **XmNmaximum**, **XmNminimum**, and **XmNsliderSize** so that the size of the slider reflects the proportion of the entire scroll that the viewport represents. If the application resizes the scroll or if the ScrolledWindow resizes the viewport, the ScrolledWindow recomputes the values of some or all of these resources.

If the value of **XmNscrollBarDisplayPolicy** is **XmAS_NEEDED**, as it is by default in automatic scrolling, the ScrolledWindow displays a ScrollBar only if the size of the scroll exceeds the size of the viewport in the relevant dimension. If the value of **XmNscrollBarDisplayPolicy** is **XmSTATIC**, the ScrolledWindow always displays both ScrollBars.

As the user manipulates a ScrollBar and changes its **XmNvalue**, the ScrolledWindow moves the scroll with respect to the viewport. For example, if the user moves the slider down in a vertical ScrollBar, the ScrolledWindow moves the scroll up with respect to the viewport.

The ScrolledWindow may need to move the scroll (and set a ScrollBar's **XmNvalue**) in circumstances other than the user's interaction with the ScrollBar. For example, if the viewport is at the bottom of the scroll and the application reduces the height of the scroll, the ScrolledWindow must move the scroll down with respect to the viewport. In this case, it reduces the ScrollBar's **XmNmaximum** and **XmNvalue**.

In automatic scrolling, the application should not try to set any of the following resources:

- Any geometry resources of the viewport (the **XmNclipWindow**)
- The **XmNmaximum**, **XmNminimum**, **XmNvalue**, **XmNincrement**, or **XmNpageIncrement** of a ScrollBar

The application can add callbacks of its own to a ScrollBar but, because the ScrolledWindow adds its own callbacks, the application must not call **XtRemoveAllCallbacks** for a ScrollBar.

The application or user can specify other resources, such as those that determine appearance, for the ScrolledWindow or its children. The names of the automatically created ScrollBars are "HorScrollBar" and "VertScrollBar".

8.2.1 Traversing to Obscured Widgets

By default, it is not possible to use keyboard traversal to move to a widget that is inside the ScrolledWindow but outside the viewport. For example, if the user presses osfNextField and the next field is not within the viewport, focus does not move to that field. The user must first use the ScrollBars or a scrolling command to position the viewport so that the target widget is no longer obscured.

ScrolledWindow has a callback list, **XmNtraverseObscuredCallback**, that allows an application to make it possible to traverse to widgets that are in the ScrolledWindow but not in the viewport. The callback list is invoked when the user tries to traverse to such a widget in a ScrolledWindow with automatic scrolling. The callback procedure is passed a pointer to an **XmTraverseObscuredCallbackStruct** structure, which contains the reason (**XmCR_OBSCURED_TRAVERSAL**), the event, the widget that is the target of the traversal, and the traversal direction passed to **XmProcessTraversal**.

Usually the callback procedure can allow traversal to the target widget simply by calling **XmScrollVisible**. This function takes as arguments the ScrolledWindow, the target widget, and requested margins between the target widget and the edges of the viewport. The function moves the work area with respect to the viewport to make the obscured widget visible. This function applies only to ScrolledWidgets with automatic scrolling.

When ScrolledWindows are nested and focus is in an inner ScrolledWindow, the **XmNtraverseObscuredCallback** callbacks of the inner ScrolledWindow are invoked first if necessary. If the destination widget remains outside the viewport of the first ancestor ScrolledWindow, that ScrolledWindow’s **XmNtraverseObscuredCallback** callbacks are invoked, and so on up the widget hierarchy.

8.2.2 Example of Automatic Scrolling

This section contains the scrolling-related portions of an example program that uses a ScrolledWindow with an automatic scrolling policy. The ScrolledWindow is actually a MainWindow, a subclass of ScrolledWindow that is often the containing manager for the primary window of an application. (MainWindow is discussed in Section 8.4.) The scroll widget is a DrawingArea.

The application allows the user to create a simple map in the DrawingArea. The user can use the mouse to establish points representing cities and to draw lines between the cities. The application contains a TextField that allows the user to enter the name of a city and then to create a button child of the DrawingArea located at the city and containing the city’s name as its label. The user can adjust the size of the DrawingArea by manipulating two Scales, one for the height of the DrawingArea and the other for the width. Other parts of the application save and retrieve the map data.

This section contains only the portions of the application that relate directly to creating and maintaining the ScrolledWindow. These include:

- Creating the MainWindow with an automatic scrolling policy
- Creating the DrawingArea child of the ScrolledWindow
- Resizing the DrawingArea in response to the user's interaction with the Scales
- Establishing an **XmNtraverseObscuredCallback** procedure

```
/*--------------------------------------------------------------
**  Create a Main Window with a menubar, a command panel
**  containing 2 Scales and a TextField, and a workarea.
**  Also put in the graphic structure the workarea info and the
**  textfield ids.
*/
void CreateApplication (
Widget          parent,
Graphic *       graph)
{
 Widget main_window, menu_bar, menu_pane, cascade,
           button, comw, scale;
 Arg args[5];
 int n;

    /*  Create automatic MainWindow.
     */
    n = 0;
    XtSetArg (args[n], XmNscrollingPolicy, XmAUTOMATIC);  n++;
    main_window = XmCreateMainWindow (parent, "main_window",
                    args, n);

    XtAddCallback (main_window, XmNtraverseObscuredCallback,
                   TravCB, (XtPointer)graph);

    XtManageChild (main_window);

...
```

```
/*  Create work_area in MainWindow
 */
n = 0;
XtSetArg (args[n], XmNresizePolicy, XmRESIZE_NONE); n++;
XtSetArg (args[n], XmNmarginWidth, 0); n++;
XtSetArg (args[n], XmNmarginHeight, 0); n++;
/* hardcode this one, since its is required for the motion handling */
xlations = XtParseTranslationTable(drawTranslations);
XtSetArg (args[n], XmNtranslations, xlations); n++;
graph->work_area = XmCreateDrawingArea(main_window, "work_area",
                                       args, n);
XtAddCallback (graph->work_area, XmNexposeCallback, DrawCB,
               (XtPointer)graph);
XtAddCallback (graph->work_area, XmNresizeCallback, DrawCB,
               (XtPointer)graph);
XtAddCallback (graph->work_area, XmNinputCallback, DrawCB,
               (XtPointer)graph);
XtManageChild (graph->work_area);


/*  Create a commandWindow in MainWindow with text and scales
 */
n = 0;
comw = XmCreateRowColumn(main_window, "comw", args, n);
XtManageChild (comw);
n = 0;
XtSetArg (args[n], XmNcommandWindow, comw);  n++;
XtSetValues (main_window, args, n);

/* find the initial size of the work_area and report to the scales */
n = 0;
XtSetArg (args[n], XmNwidth, &graph->old_width);  n++;
XtSetArg (args[n], XmNheight, &graph->old_height);  n++;
XtGetValues (graph->work_area, args, n);

n = 0;
XtSetArg (args[n], XmNorientation, XmHORIZONTAL);  n++;
XtSetArg (args[n], XmNshowValue, True);  n++;
XtSetArg (args[n], XmNvalue, graph->old_width);  n++;
```

```
        scale = XmCreateScale(comw, "scale_w", args, n);
           /* scale_w is the name */
        XtAddCallback (scale, XmNvalueChangedCallback, ValueCB,
                       (XtPointer)graph->work_area);
        XtManageChild (scale);

        n = 0;
        XtSetArg (args[n], XmNorientation, XmHORIZONTAL);  n++;
        XtSetArg (args[n], XmNshowValue, True);  n++;
        XtSetArg (args[n], XmNvalue, graph->old_height);  n++;
        scale = XmCreateScale(comw, "scale_h", args, n);
        XtAddCallback (scale, XmNvalueChangedCallback, ValueCB,
                       (XtPointer)graph->work_area);
        XtManageChild (scale);

        n = 0;
        graph->textf = XmCreateTextField(comw, "textf", args, n);
        XtManageChild (graph->textf);


        /*  Set MainWindow areas
         */
        XmMainWindowSetAreas (main_window, menu_bar, comw, NULL, NULL,
                              graph->work_area);

    }

    /*--------------------------------------------------------------
    **      TravCB          - callback for traverseObscure
    */
    void TravCB (
    Widget          w,              /*  widget id           */
    XtPointer       client_data,    /*  data from application   */
    XtPointer       call_data)     /*  data from widget class  */
    {
     XmTraverseObscuredCallbackStruct * tocs =
       (XmTraverseObscuredCallbackStruct *) call_data;
     Graphic * graph = (Graphic *) client_data;

        if (tocs->traversal_destination != graph->work_area)
            XmScrollVisible(w, tocs->traversal_destination, 20, 20);
    }

    /*-------------------------------------------------------------
    **      ValueCB         - callback for scales
    */
    void ValueCB (
    Widget          w,              /*  widget id           */
    XtPointer       client_data,    /*  data from application   */
    XtPointer       call_data)     /*  data from widget class  */
    {
     Arg args[5];
     int n;
     int value;
     Widget workarea = (Widget) client_data;

        /* get the value out of the Scale */
        n = 0;
        XtSetArg (args[n], XmNvalue, &value);  n++;
        XtGetValues (w, args, n);

        n = 0;
        if (strcmp(XtName(w), "scale_w") == 0) { /* width scale */
            XtSetArg (args[n], XmNwidth, value);  n++;
        } else {
            XtSetArg (args[n], XmNheight, value);  n++;
        }
        XtSetValues (workarea, args, n);
    }
```

8.3 Application-Defined Scrolling

In application-defined scrolling, the application is responsible for all aspects of the interactions among the scroll, the viewport, and the ScrollBars. The ScrolledWindow remains responsible for geometry and layout, but the application must adjust both the ScrollBars and the scroll position in response to the user's scrolling actions.

Because this model requires more work on the part of the application, it is most suitable for programs in which automatic scrolling is not adequate. For example, an application may contain a text editor or browser that reads only enough of a file to fill the viewport. This application must be informed of the user's scrolling actions so that it can read more of the file when necessary.

The application implements a scheme of its choosing for the relationship between the scroll and the viewport. Following are two common models:

- A fixed-size viewport widget as the parent of a variable-sized scroll widget that contains the data. The application resizes the scroll widget as necessary to contain all the data. As the user interacts with the ScrollBar, the application moves the scroll widget with respect to the viewport, which clips the scroll. This is the model that ScrolledWindow uses for automatic scrolling.
- A single widget that serves as the viewport, with the scroll contained in internal data structures or a combination of data structures and files. The application expands the internal structures as necessary to contain all the data. As the user interacts with the ScrollBar, the application retrieves the appropriate portion of the data from the internal structures or files and displays that portion of the data in the viewport. This is the model that the Motif ScrolledList and ScrolledText widgets use.

In both models, the application must be notified when the viewport is resized. It may need to adjust the scroll with respect to the viewport, and it must recompute ScrollBar resources to reflect the new relation between the viewport and the scroll. If the viewport is a DrawingArea, the application can use the **XmNresizeCallback** callbacks for this purpose. Otherwise, the application can establish an event handler for **ConfigureNotify** events.

The application needs to take the following steps to use application-defined scrolling:

- Create and manage a ScrolledWindow, horizontal and vertical ScrollBar children, and a child to serve as the viewport.
- If the application is using a separate widget as the scroll, create and manage that widget as a child of the viewport widget.
- Add callbacks to the ScrollBars to notify the application when the user interacts with the ScrollBars. The application should at least provide a procedure for the **XmNvalueChangedCallback** list.

- Add a callback (such as the DrawingArea **XmNresizeCallback**) or an event handler to the viewport widget to notify the application when the widget is resized.
- Based on the initial relationship between the viewport and the scroll, supply initial values for the ScrollBars' **XmNincrement**, **XmNpageIncrement**, **XmNmaximum**, **XmNminimum**, **XmNvalue**, and **XmNsliderSize** resources.
- Adjust the size of the scroll widget or internal data structures as necessary to contain the data in the scroll.
- As the data in the scroll changes, recompute the ScrollBars' **XmNmaximum** and **XmNsliderSize** and perhaps **XmNminimum** and **XmNvalue** to reflect the new relation between the viewport and the scroll.
- When the viewport is resized, reposition and resize the scroll with respect to the viewport if necessary. Recompute the ScrollBars' **XmNsliderSize** and **XmNpageIncrement** and possibly other resources to reflect the new relationship between the viewport and the scroll.
- As the user interacts with the ScrollBars, if a separate scroll widget exists, reposition the scroll with respect to the viewport. If no separate scroll widget exists, bring in additional data from files if necessary, recompute which portion of the data to make visible, and redisplay the viewport. If the size of the scroll has changed, recompute the ScrollBar resources to reflect the new relationship between the viewport and the scroll.

8.3.1 Example of Application-Defined Scrolling

This section contains the scrolling-related portions of an example program that uses a ScrolledWindow with an application-defined scrolling policy. As in the example of automatic scrolling, the ScrolledWindow is a MainWindow, and the scroll widget is a DrawingArea. In this example, the scroll widget also serves as the viewport widget, and the scroll data is maintained in internal data structures.

The application is a simple file browser for C source code. The user selects a filename. The program reads the file and parses it (in the C locale) into an internal table of lines. The application displays in the DrawingArea as many lines as will fit into the current dimensions of the DrawingArea.

The application uses only a vertical ScrollBar, which allows the user to browse through the file. After reading the file, the program sets the ScrollBar's **XmNminimum** and

XmNvalue to 0, its **XmNmaximum** to the number of lines in the file, and its **XmNsliderSize** to the lesser of the number of lines in the file and the number of lines that can be displayed in the viewport.

The program establishes a ScrollBar **XmNvalueChangedCallback** and a DrawingArea **XmNexposeCallback** that redisplay the lines in the viewport. The redisplay procedure fetches and displays lines from the internal data structure, starting with the line indicated by the ScrollBar's **XmNvalue** and proceeding to the last line that fits in the viewport. The program also establishes a DrawingArea **XmNresizeCallback** that recomputes the ScrollBar's **XmNsliderSize** and **XmNvalue** based on the number of lines that can be displayed in the viewport. The application does not resize the DrawingArea itself.

This section contains only the portions of the application that relate directly to creating and maintaining the ScrolledWindow. These include:

- Creating the MainWindow with an application-defined scrolling policy
- Creating the DrawingArea and vertical ScrollBar children of the ScrolledWindow
- Establishing an **XmNactivateCallback** callback for the OK button of the FileSelectionBox invoked from the file menu Open button
- Establishing a ScrollBar **XmNvalueChangedCallback** callback
- Establishing a DrawingArea **XmNexposeCallback** callback and an **XmNresizeCallback** callback

```
/*-------------------------------------------------------------
**      Internal data structure to hold file info.
*/
typedef struct {
    Widget work_area;
    Widget v_scrb;
    String file_name;
    XFontStruct * font_struct;
    GC draw_gc;
    char ** lines;
    int num_lines;
} FileData;

/*-------------------------------------------------------------
** Create a MainWindow with a MenuBar to load a file.
** Add the vertical scrollbar and the workarea to filedata.
*/
void CreateApplication (
Widget          parent,
FileData *      filedata)
{
    Widget main_window, menu_bar, menu_pane, cascade,
           button;
    Arg args[5];
    int n;

    /*  Create app_defined MainWindow.
     *  XmAPPLICATION_DEFINED is the default; however we set it here
     *  to override any values specified by a user in a resource file
     */
    n = 0;
    XtSetArg (args[n], XmNscrollingPolicy,
              XmAPPLICATION_DEFINED);  n++;
    main_window = XmCreateMainWindow (parent,
                                "main_window", args, n);
    XtManageChild (main_window);

    /*  Create MenuBar in MainWindow.
     */

...

    /* Create "File" PulldownMenu with Open and Quit buttons
     */

    n = 0;
    menu_pane = XmCreatePulldownMenu (menu_bar,
                                      "menu_pane", args, n);

    n = 0;
    button = XmCreatePushButton (menu_pane, "Open...",
                                 args, n);
    XtManageChild (button);

    /* pass the file data to the Open callback */
    XtAddCallback (button, XmNactivateCallback,
                   OpenCB, (XtPointer)filedata);
    n = 0;
    button = XmCreatePushButton (menu_pane, "Quit", args, n);
    XtManageChild (button);
    XtAddCallback (button, XmNactivateCallback, QuitCB, NULL);

    n = 0;
    XtSetArg (args[n], XmNsubMenuId, menu_pane);  n++;
    cascade = XmCreateCascadeButton (menu_bar, "File",
                                     args, n);
    XtManageChild (cascade);

    /*  Create "Help" PulldownMenu with Help button.
     */

...

    /*  Create vertical scrollbar only
     */
     n = 0;
    XtSetArg (args[n], XmNorientation, XmVERTICAL);  n++;
    filedata->v_scrb = XmCreateScrollBar (main_window,
                                          "v_scrb", args, n);
    XtAddCallback (filedata->v_scrb, XmNvalueChangedCallback,
                   ValueCB, (XtPointer)filedata);
    XtManageChild (filedata->v_scrb);

    /*  Create work_area in MainWindow
     */
    n = 0;
    filedata->work_area = XmCreateDrawingArea(main_window,
                                      "work_area", args, n);
    XtAddCallback (filedata->work_area, XmNexposeCallback,
                   DrawCB, (XtPointer)filedata);
    XtAddCallback (filedata->work_area, XmNresizeCallback,
                   DrawCB, (XtPointer)filedata);
    XtManageChild (filedata->work_area);

    /*  Set MainWindow areas
```

```
     */
    XmMainWindowSetAreas (main_window, menu_bar, NULL, NULL,
                          filedata->v_scrb,
                          filedata->work_area);

}

/*-------------------------------------------------------------
**      OpenCB                  - callback for Open button
*/
void OpenCB (
Widget          w,              /*  widget id           */
XtPointer       client_data,    /*  data from application   */
XtPointer       call_data)     /*  data from widget class  */
{
        static Widget fsb_box = NULL;

        if (!fsb_box) {
            fsb_box = XmCreateFileSelectionDialog (w,
                                        "Load file", NULL, 0);
            /* just propagate the file information */
            XtAddCallback (fsb_box, XmNokCallback, ReadCB,
                                                 client_data);
        }

        XtManageChild (fsb_box);
}

/*-------------------------------------------------------------
**      ReadCB  - callback for fsb activate
*/
void ReadCB (
Widget          w,              /*  widget id           */
XtPointer       client_data,    /*  data from application   */
XtPointer       call_data)     /*  data from widget class  */
{
    FileData * filedata = (FileData *) client_data;
    String file_name;
    Arg args[5];
    int n, slider_size;
    Dimension height;

    file_name = XmTextGetString(
                XmFileSelectionBoxGetChild(w, XmDIALOG_TEXT));

    if (!BuildLineTable(filedata, file_name)) {
        WarnUser (w, "Cannot open %s\n", file_name);
    } else {
        filedata->file_name = file_name;

        /* ok, we have a new file, so reset some values */
        n = 0;
        XtSetArg (args[n], XmNheight, &height);  n++;
        XtGetValues (filedata->work_area, args, n);

        slider_size = (height - 4) /
                             (filedata->font_struct->ascent
                            + filedata->font_struct->descent);
        if (slider_size <= 0) slider_size = 1;
        if (slider_size > filedata->num_lines)
            slider_size = filedata->num_lines;

        n = 0;
        XtSetArg (args[n], XmNsliderSize, slider_size);  n++;
        XtSetArg (args[n], XmNmaximum, filedata->num_lines);
                    n++;
        XtSetArg (args[n], XmNvalue, 0);  n++;
        XtSetValues (filedata->v_scrb, args, n);

        /* clear and redraw */
        XClearWindow(XtDisplay(filedata->work_area),
                     XtWindow(filedata->work_area));
        ReDraw (filedata);
    }
}

/*-------------------------------------------------------------
**      ValueCB         - callback for scrollbar
*/
void ValueCB (
Widget          w,              /*  widget id           */
XtPointer       client_data,    /*  data from application   */
XtPointer       call_data)     /*  data from widget class  */
{
    FileData * filedata = (FileData *) client_data;

    /* clear and redraw, dumb dumb.. */
    XClearWindow(XtDisplay(filedata->work_area),
                 XtWindow(filedata->work_area));
    ReDraw(filedata);
}

/*-------------------------------------------------------------
**      DrawCB                  - callback for drawing area
*/
void DrawCB (
Widget          w,              /*  widget id           */
XtPointer       client_data,    /*  data from application   */
XtPointer       call_data)     /*  data from widget class  */
{

    XmDrawingAreaCallbackStruct * dacs =
        (XmDrawingAreaCallbackStruct *) call_data;
    FileData * filedata = (FileData *) client_data;
    XSetWindowAttributes xswa;

    static Boolean first_time = True;

    switch (dacs->reason) {
    case XmCR_EXPOSE:
        if (first_time) {
            /* Change once the bit gravity of the
               Drawing Area; default is north west and we
               want forget, so that resize always
               generates exposure events */
            first_time = False;
            xswa.bit_gravity = ForgetGravity;
            XChangeWindowAttributes(XtDisplay(w), XtWindow(w),
                                    CWBitGravity, &xswa);
        }
```

```
            ReDraw(filedata);

            break;
        case XmCR_RESIZE:
            ReSize(filedata);

            break;
        }
}

void ReDraw(
FileData * filedata)
{
        /* Display as many line as slider_size actually shows,
           since slider_size is computed relative to the
           work_area height */

        Cardinal i;
        int value, slider_size;
        Arg args[5];
        int n;
        Position y;

        if (filedata->num_lines == 0) return;

        n = 0;
        XtSetArg (args[n], XmNvalue, &value);  n++;
        XtSetArg (args[n], XmNsliderSize, &slider_size);  n++;
        XtGetValues (filedata->v_scrb, args, n);

        for (i = value, y = 2 + filedata->font_struct->ascent;
             i < value + slider_size;
             i++, y += (filedata->font_struct->ascent
                        + filedata->font_struct->descent)) {
            XDrawString(XtDisplay(filedata->work_area),
                        XtWindow(filedata->work_area),
                        filedata->draw_gc,
                        4, y,
                        filedata->lines[i],
                    strlen(filedata->lines[i]));
    }
}

void ReSize(
FileData * filedata)
{
    /* Just update the scrollbar internals here, don’t
       bother to redisplay since the gravity is none */

    Arg args[5];
    int n;
    int value, slider_size;
    Dimension height;

    if (filedata->num_lines == 0) return;

    n = 0;
    XtSetArg (args[n], XmNheight, &height);  n++;
    XtGetValues (filedata->work_area, args, n);

    /* sliderSize is the number of visible lines */
    slider_size = (height - 4) /
                          (filedata->font_struct->ascent
                         + filedata->font_struct->descent);
    if (slider_size <= 0) slider_size = 1;
    if (slider_size > filedata->num_lines)
        slider_size = filedata->num_lines;

    n = 0;
    XtSetArg (args[n], XmNvalue, &value);  n++;
    XtGetValues (filedata->v_scrb, args, n);

    /* value shouldn’t change that often but there are cases
       where it matters */
    if (value > filedata->num_lines - slider_size)
        value = filedata->num_lines - slider_size;

    n = 0;
    XtSetArg (args[n], XmNsliderSize, slider_size);  n++;
    XtSetArg (args[n], XmNvalue, value);  n++;
    XtSetArg (args[n], XmNmaximum, filedata->num_lines); n++;
    XtSetValues (filedata->v_scrb, args, n);
}
```

8.4 MainWindow

Motif provides a widget, MainWindow, that serves as a template for the primary window of most applications. MainWindow is a subclass of ScrolledWindow. In addition to the viewport and ScrollBar components of the ScrolledWindow, MainWindow has an optional MenuBar, Command window, and Message window.

MainWindow lays out these components in a manner compliant with the *CDE 2.1/ Motif 2.1—Style Guide and Glossary* specifications for the primary window of an application. The MenuBar, if present, spans the top of the MainWindow horizontally. By default, the Command window, if present, spans the MainWindow horizontally just below the MenuBar. The ScrolledWindow viewport and ScrollBars are below the Command window, and the Message window is below the ScrolledWindow viewport or horizontal ScrollBar. If the MainWindow resource **XmNcommandWindowLocation** is set to **XmCOMMAND_BELOW_WORKSPACE** at the time the MainWindow is created, the Command window is located below the ScrolledWindow viewport or horizontal ScrollBar.

If the MainWindow resource **XmNshowSeparator** is True, the MainWindow automatically creates up to three SeparatorGadgets to separate the components. The names of these automatically created SeparatorGadgets are "Separator1", "Separator2", and "Separator3". The application can retrieve the widget IDs of the SeparatorGadgets by using the functions **XmMainWindowSep1**, **XmMainWindowSep2**, and **XmMainWindowSep3**.

In addition to the ScrolledWindow resources that hold the widget IDs of the ScrollBars, scroll widget, and viewport widget, MainWindow has resources that hold the widget IDs of the other MainWindow components:

XmNcommandWindow
: The value is the widget ID of the Command window. If a child is a Command widget and no Command window exists, MainWindow automatically sets the value of this resource to the child’s widget ID.

XmNmenuBar
: The value is the widget ID of the MenuBar. If a child is a MenuBar and no MainWindow MenuBar exists, MainWindow automatically sets the value of this resource to the child's widget ID.

XmNmessageWindow
: The value is the widget ID of the Message window. After creating the Message window, the application must use **XtSetValues** to set the value of this resource to the child's widget ID.

MainWindow has a convenience routine, **XmMainWindowSetAreas**, to establish both the MainWindow and the ScrolledWindow components. **XmMainWindowSetAreas** does not set the Message window; an application must use **XtSetValues** of **XmNmessageWindow** to set the Message window. An application that has no Message window and uses only standard components for the other MainWindow children may not need to call **XmMainWindowSetAreas** or **XtSetValues** for the component resources, but it is good practice to make these calls. If an application uses a Message window or has additional MainWindow children beyond the standard components, it must call **XmMainWindowSetAreas** and **XtSetValues** for **XmNmessageWindow**.

An application takes the following steps to use MainWindow:

1. Create and manage the MainWindow, usually as a child of the ApplicationShell. If the scrolling mode is to be automatic, supply an initial value of **XmAUTOMATIC** for **XmNscrollingPolicy**.
2. Create and manage the components of the MainWindow.
3. If necessary, call **XmMainWindowSetAreas** or **XtSetValues** for the MainWindow components.
4. Take any other actions needed to regulate the ScrolledWindow components. These actions are discussed in the previous descriptions of automatic and application-defined scrolling.

For examples of using MainWindow with both automatic and application-defined scrolling policies, see the ScrolledWindow examples in the previous sections.

8.5 Container

Container is a manager that accepts only widgets of class **XmIconGadget** or its subclasses as children. Applications use the IconGadget to display information in graphic and/or text form. Applications can use the Container to view the IconGadgets in different formats, to select, and to manipulate the IconGadgets.

Container has several layout options for displaying its IconGadget children. An application can use Container’s **XmOUTLINE** layout to convey hierarchical relationships between the information or objects represented by the children; for example, an organization chart or the taxonomy of a particular genus of plant. Container’s **XmDETAIL** layout allows the application to display additional information with each child lined up in vertical columns. Options within Container’s **SPATIAL** layout can be used by applications to display children in a grid-like configuration as in an icon-box for a window manager. Or it can display them in specific locations; for example, a Container application could display IconGadgets that represent workstations in a computer lab so that each IconGadget’s position in the Container corresponds to its location on the lab floor.

When Container’s **SPATIAL** layout is used, users can drag and drop IconGadgets within the Container to reposition them. An application where this might be useful is a chessboard, or an application to select planting locations within a garden plot.

Container supports the same four modes of selection as the List widget. The **XmNselectionPolicy** resource controls the selection mode. Applications using Container can allow users to select only one or more than one IconGadget child at a time. Container invokes callbacks associated with the **XmNselectionCallback** resource whenever a user action causes a change to the set of selected items. Container also invokes callbacks associated with the **XmNdefaultActionCallback** whenever a user performs an activate action on an IconGadget child. Using these callbacks, an application using Container can respond to user actions, as in the example below.

This example program uses IconGadgets to represent workers in a fictional organization. The code could be the front end to an office telephone system. It uses Container’s **XmOUTLINE** layout to arrange the information so that it corresponds to an organization chart. The **XmNselectionPolicy** is set to **XmBROWSE_SELECT** so that only one selection can be made at a time. If this application were to support an electronic mailing system, the application could use Container’s **XmEXTENDED** or **XmMULTIPLE** selection policy to allow multiple IconGadgets to be selected at once.

```
void
MakeACall(Widget        w,
          XtPointer     client_data,
          XtPointer     call_data)
{
 /* Ring that person’s phone when selected. */
  printf("Ring!\n");
}


void
CreateContainer(Widget parent_of_container)
{
 Widget  container1;
 Widget  president, vice_president, dir_of_sales, dir_of_rnd, dir_of_mfr;

   container1 = XtVaCreateWidget("Container",
                        xmContainerWidgetClass, parent_of_container,
                        XmNlayoutType, XmOUTLINE,
                        XmNselectionPolicy, XmBROWSE_SELECT,
                        XmNautomaticSelection, XmNO_AUTO_SELECT,
                        XmNentryViewType, XmSMALL_ICON,
                        NULL);
   XtAddCallback(container1, XmNselectionCallback, MakeACall,
                 (XtPointer)NULL);

   president = XtVaCreateManagedWidget("President",
                        xmIconGadgetClass, container1,
                        XmNoutlineState, XmEXPANDED,
                        NULL);
   vice_president = XtVaCreateManagedWidget("Vice-President",
                        xmIconGadgetClass, container1,
                        XmNentryParent, president,
                        XmNoutlineState, XmEXPANDED,
                        NULL);
   dir_of_sales = XtVaCreateManagedWidget("Director of Sales",
                        xmIconGadgetClass, container1,
                        XmNentryParent, vice_president,
                        NULL);
   dir_of_rnd = XtVaCreateManagedWidget("Director of R&D",
                        xmIconGadgetClass, container1,
                        XmNentryParent, vice_president,
                        NULL);
   dir_of_mfr = XtVaCreateManagedWidget("Director of Manufacturing",
                        xmIconGadgetClass, container1,
                        XmNentryParent, vice_president,
                        NULL);
   XtManageChild(container1);
}
```

The preceding example appears online in directory **demos/doc/programGuide/ch08/Container**.

This example could be expanded to use other Container features. Container’s **XmDETAIL** view could be used to give additional information about each of the workers like "job responsibilities" and "internal mail address." The application could use Container’s **XmNdefaultActionCallback** to enable users to edit this same information. And, assuming the fictional workers weren’t located on separate floors or separate sites, the application could support a switch to an **XmSPATIAL** view that has the office floorplan as the Container’s background pixmap and positions each IconGadget according to the corresponding worker’s office location.

8.6 Notebook

Notebook is a manager that treats all children as either a page, a component to access a page, or a component that describes a page.

Setting the **XmNnotebookChildType** constraint resource to **XmPAGE** when creating a child of Notebook indicates that it is to be treated as a page. Notebook will only display one page at any one time (determined by the **XmNcurrentPageNumber** resource) even though the Notebook may have several page-type widgets, all of which are managed and realized. This makes the Notebook a useful component for applications that maintain a large amount of information but only wish to allow the user to see and/or manipulate a small portion of the information at one time. An application that maintains a large set of images may wish to use Notebook to keep track of them and to allow only one at a time to be seen so that screen real estate and color resources are conserved. An application that maintains user-configurable data (such as name and address entries, Xserver options, book catalog information) can segment that data into pages for the Notebook. Each page-type widget has a **XmNpageNumber** constraint resource (Notebook will assign it a default one if one isn't given). When that **XmNpageNumber** constraint resource matches Notebook's **XmNcurrentPageNumber** resource, the page-type widget will be shown. Since the **XmNpageNumber** constraint resource can be changed by the application, a single widget can represent many "virtual" pages. For example, an application displaying help text can place all the text in a single ScrolledText widget and, by changing the **XmNpageNumber** constraint resource and changing the scroll position, allow a user to access the textual information by page instead of scrolling to find it.

An application can cause pages in Notebook to be displayed by changing the value of the **XmNcurrentPageNumber** resource. However, an application can also provide other child widgets to Notebook that will enable the user to access pages. By setting the constraint resource **XmNnotebookChildType** to **XmMAJOR_TAB**, **XmMINOR_TAB**, or **XmPAGE_SCROLLER** when creating a child of the Notebook, that child widget will be used by the Notebook to access pages. Each of these child widgets will access the page that corresponds to their **XmNpageNumber** constraint resource.

Notebook always contains one page-scroller. If one is not supplied, then Notebook will create it by default. If more than one is supplied, then Notebook will only display one of them. The page-scroller (**XmPAGE_SCROLLER**) is, by default, a SpinBox component and allows a user to access pages by changing the **XmNcurrentPageNumber** resource one increment at a time.

Applications can use major-tabs (**XmMAJOR_TAB**) and minor-tabs (**XmMINOR_TAB**) to allow users to access pages individually or within groups of pages. For example, an online photo album with one picture per page could add a major tab for every page; each major-tab could be a PushButton with a description of the corresponding picture in its label. The user can then access a particular photo by pressing the tab with the photo description instead of using the page-scroller to access each page one at a time. If the same photo album were, in fact, a class yearbook, there might be many of these labels. So, instead of major-tabs, PushButtons with descriptive labels (the person's name) could be attached to each page as minor-tabs. Then a major-tab could be added to the first person's photo in each class; the major-tab PushButton would have a label describing the class as a whole. In this application, the user would be able to use the major-tab components to quickly skip to the class of interest, and then use the minor-tabs to find the picture of interest. There might still be a large number of minor tabs; if Notebook does not have enough space to display all the major-tabs and/or minor-tabs that are to be shown, then Notebook will provide tab-scrollers (arrow buttons) so the user can scroll to the major-tab and/or minor-tab of interest.

Some applications might require all three of the component types that are used to access pages. A calendar application might have a page for each day. Major-tabs that display a particular year could be added to the first page (day) of each year. Minor-tabs that display the name of a month could be added to the first page (day) of each month. Users could quickly access the year and month of interest by activating the corresponding major-tab and minor-tab and then access the day of interest by using the page-scroller.

Setting the **XmNnotebookChildType** constraint resource to **XmSTATUS_AREA** when creating a child of Notebook indicates that it is used to describe a page. As with a page-type widget, Notebook will only display a status-area when its **XmNpageNumber** constraint resource is equal to Notebook's **XmNcurrentPageNumber** resource. Status-areas can be used by applications to describe pages. In the previously mentioned photo album example, a status-area widget could contain information about the photo that is not contained in the photo (page) itself or in the major-tab or minor-tab descriptive labels. It could describe the date the photo was taken, the shutter speed used, the type of film used, and so on.

Notebook invokes callbacks associated with the **XmNpageChangedCallback** resource whenever the value of the **XmNcurrentPageNumber** resource is changed. After invoking the callbacks, Notebook displays the children that have an **XmNpageNumber** constraint resource equal to Notebook's **XmNcurrentPageNumber** resource. However, if there are multiple children of the same type that share the same **XmNpageNumber**, then Notebook displays only one of that type. For example, if the application defines two minor tabs that both have an **XmNpageNumber** constraint resource equal to **XmNcurrentPageNumber**, then Notebook displays only one of the minor tabs. The callback information contains the page number that is about to be displayed. Since the callback is done first, applications don't have to maintain a separate child widget for each page number and, in fact, do not have to create a page widget or access the information needed for a particular page until that page is actually requested by the user. An application can use **XmNpageChangedCallback** to always update a single page and/or status-area widget to always have the current page number in its **XmNpageNumber** constraint resource; the information content rather than the widget itself can be changed. In the previous photo album example, the image data required to display a photo needn't be loaded into memory until the **XmNpageChangedCallback** indicates that the user wishes to see that photo.

Following are the relevant portions of a simple example program that creates a Notebook with 7 pages, numbered 1 through 7, all Label components. Each Label displays a simple string. The 7 pages are divided into two major tab categories, fruits and vegetables. The vegetables category is subdivided into two minor tab categories, green and orange. A status area appears in the Notebook when the current page number is 2. No page-scroller is created by the application, so the Notebook widget will create a default page-scroller.

In this example, all pages can be accessed by the page-scroller; pages 1 and 4 can be accessed by major-tab buttons; and pages 4 and 6 can be accessed by minor-tab buttons.

```
CreateNotebook(Widget parent_of_notebook)
{
#define PAGES_IN_NOTEBOOK 7
#define NUMBER_OF_MAJOR_TABS 2

 Widget notebook, frame;
 char buff[80];
 int i;
 static char *info[PAGES_IN_NOTEBOOK+1] = {
                                  "dummy",
                                  "apples are high in fiber",
                                  "bananas are high in Potassium",
                                  "oranges are high in Vitamin C",
                                  "celery",
                                  "lettuce",
                                  "sweet potato",
                                  "carrot"
                                 };

   notebook = XtVaCreateWidget("notebook", xmNotebookWidgetClass,
                              parent_of_notebook, NULL);

 /* Create the pages of the Notebook. */
   for (i=1; i<=PAGES_IN_NOTEBOOK; i++) {
      /* Create a frame on every page. */
        frame = XtVaCreateManagedWidget("frame",
                     xmFrameWidgetClass, notebook,
                     XmNnotebookChildType, XmPAGE,
                     XmNpageNumber,i,
                     NULL);

      /* Place the page contents (a string) on each page. */
        XtVaCreateManagedWidget(info[i],
                     xmLabelWidgetClass, frame,
                     NULL);
   }

   XtVaCreateManagedWidget("tropical only",
                     xmLabelWidgetClass, notebook,
                     XmNnotebookChildType, XmSTATUS_AREA,
                     XmNpageNumber, 2,
                     NULL);

 /* Create major tabs to divide the pages into categories. */
   XtVaCreateManagedWidget("fruits",
                xmPushButtonWidgetClass, notebook,
                XmNnotebookChildType, XmMAJOR_TAB,
                XmNpageNumber, 1,
                NULL);
   XtVaCreateManagedWidget("vegetables",
                xmPushButtonWidgetClass, notebook,
                XmNnotebookChildType, XmMAJOR_TAB,
                XmNpageNumber, 4,
                NULL);


 /* Create some minor tabs to divide the categories into
    subcategories. */
   XtVaCreateManagedWidget("green",
                xmPushButtonWidgetClass, notebook,
                XmNnotebookChildType, XmMINOR_TAB,
                XmNpageNumber, 4,
                NULL);
   XtVaCreateManagedWidget("orange",
                xmPushButtonWidgetClass, notebook,
                XmNnotebookChildType, XmMINOR_TAB,
                XmNpageNumber, 6,
                NULL);

   XtManageChild(notebook);

}
```

The preceding example appears online in directory **demos/doc/programGuide/ch08/Notebook**.

8.7 Frame

Frame is a simple manager that encloses a child and displays a shadow around it. An application usually uses a Frame to provide a shadow for a widget, such as a RowColumn WorkArea, that does not display a shadow itself. The Frame resource **XmNshadowType** determines the type of shadow to draw. The resources **XmNmarginHeight** and **XmNmarginWidth** specify the margin between the shadow and the border of the child.

Frame can also have one other child that serves as a title. Frame places the title above the principal child of the Frame. The following constraint resources determine the Frame's treatment of the child:

XmNframeChildType
The value is a constant that tells the Frame whether the child is the work area child, the title, or another kind of child. Following are the possible values:

XmFRAME_WORKAREA_CHILD
This value specifies that the child is the principal (work area) component. This is the default.

XmFRAME_TITLE_CHILD
This value specifies that the child is the Frame title.

XmFRAME_GENERIC_CHILD
This value specifies that the child is a component other than the work area and the title. When the value is

XmFRAME_GENERIC_CHILD, Frame does not include the child in its layout.

XmNchildHorizontalAlignment

The value specifies the alignment of the title with respect to the left and right inner edges of the Frame (determined by the child's **XmNchildHorizontalSpacing**). Following are the possible values:

XmALIGNMENT_BEGINNING

This value specifies that the title is placed at the left inner edge when the Frame's **XmNstringDirection** has the value **XmSTRING_DIRECTION_L_TO_R**; otherwise, the title is placed at the right inner edge. This is the default.

XmALIGNMENT_END

This value specifies that the title is placed at the right inner edge when the Frame's **XmNstringDirection** has the value **XmSTRING_DIRECTION_L_TO_R**; otherwise, the title is placed at the left inner edge.

XmALIGNMENT_CENTER

This value specifies that the title is centered between the edges.

XmNchildHorizontalSpacing

The value is the minimum distance between the title and the shadow along the left and right edges of the Frame. The default is the Frame's **XmNmarginWidth**.

XmNchildVerticalAlignment

The value specifies the alignment of the title with respect to the shadow along the top edge of the Frame. Following are the possible values:

XmALIGNMENT_BASELINE_BOTTOM

The baseline of the last line of text in the title is even with the shadow along the top edge of the Frame.

XmALIGNMENT_BASELINE_TOP

The baseline of the first line of text in the title is even with the shadow along the top edge of the Frame.

XmALIGNMENT_CENTER
The center of the title is even with the shadow along the top edge of the Frame. This is the default.

XmALIGNMENT_WIDGET_BOTTOM
The bottom edge of the title is even with the shadow along the top edge of the Frame.

XmALIGNMENT_WIDGET_TOP
The top edge of the title is even with the shadow along the top edge of the Frame.

Following is a UIL specification for an example Frame with a Label title and a Form child (not defined here):

```
object exampleFrame: XmFrame {
  controls {
      XmLabel  { arguments {
          XmNframeChildType = XmFRAME_TITLE_CHILD;
          XmNchildHorizontalSpacing = 4;
          XmNchildVerticalAlignment = XmALIGNMENT_WIDGET_BOTTOM;
        }; };
        XmForm exampleForm;
    };
};
```

8.8 PanedWindow

PanedWindow is a manager that by default lays out its children vertically from top to bottom and places a separator between each pair of children. Each child spans the width of the PanedWindow, which resizes children to be as wide as the widest child. When possible, the PanedWindow grows to accommodate the width of the widest child and the heights of all the children.

Usually PanedWindow allows the user to adjust the height of each pane. When a pane is adjustable, PanedWindow creates a control called a sash and places it below the pane that it controls. By manipulating the sash with the mouse or keyboard commands, the user changes the height of the pane above. This may also change the height of a pane below the sash.

PanedWindow has the following resources to control general appearance:

XmNmarginHeight
The value specifies the margin between the PanedWindow’s top and bottom shadows and the children nearest those shadows.

XmNmarginWidth
The value specifies the margin between the PanedWindow’s left and right shadows and the children nearest those shadows.

XmNorientation
The value specifies the layout as either vertical or horizontal.

XmNseparatorOn
The value determines whether or not PanedWindow displays a separator between each pair of panes.

XmNspacing
The value is the distance between each pane.

The following PanedWindow resources control the appearance of the sashes:

XmNsashHeight
The value specifies the height of each sash.

XmNsashIndent
The value specifies the distance between each sash and the inner margin of the left or right side of the PanedWindow. If the value is positive, the sash is offset from the near (left) side of the PanedWindow. If the value is negative, the sash is offset from the far (right) side of the PanedWindow. If the value is greater than the width of the PanedWindow minus the width of the sash, the sash is placed flush against the near side of the PanedWindow.

XmNsashShadowThickness
The value specifies the shadow thickness for each sash.

XmNsashWidth
The value specifies the width of each sash.

PanedWindow has one other resource, **XmNrefigureMode**. When this resource is set to False, the PanedWindow does not recompute its layout when either the user or the application resizes a pane or when the PanedWindow is resized.

PanedWindow children have a number of constraint resources that PanedWindow uses to determine the positions and size limitations of the panes:

XmNallowResize
: The value specifies whether the PanedWindow grants resize requests from the pane. When the value is False (the default) and the pane is realized, PanedWindow refuses such requests, but it allows the user to resize the pane if it is adjustable. For example, if the application attempts to change the height or width of the pane using **XtSetValues**, PanedWindow does not allow the change. If the value is True or if the pane is not realized, PanedWindow grants requests by the pane to change its size if possible.

XmNpaneMaximum
: The value is the maximum height to which the user or application can resize the pane. If this value is the same as the value of **XmNpaneMinimum**, the pane cannot be resized at all, and PanedWindow does not display a sash at the bottom of the pane.

XmNpaneMinimum
: The value is the minimum height to which the user or application can resize the pane. If this value is the same as the value of **XmNpaneMaximum**, the pane cannot be resized at all, and PanedWindow does not display a sash at the bottom of the pane.

XmNpositionIndex
: The value is the ordinal position of the pane in the PanedWindow's list of pane children. The application or user can specify the value as an integer between 0 and the number of children already in the list, or as the value **XmLAST_POSITION** (the default), which means the child is inserted at the end of the list. If specifying a new value causes the order of children in the list to change, PanedWindow recomputes its layout according to the new order of children: the first pane is displayed at the top of the PanedWindow, the second child below the first, and so on.

XmNskipAdjust
: The value specifies whether or not the PanedWindow resizes the pane when the PanedWindow itself is resized or when the user resizes another pane. When the value is True, PanedWindow does not resize this pane under these circumstances, but the user can still resize the pane if **XmNpaneMaximum** is greater than **XmNpaneMinimum**. The default is False.

Chapter 9
Compound Strings

A compound string is simply a way to encode text independent of the font or the color used to display it. By pairing encoded text with an entry in a table of "renditions," Motif allows the application programmer great flexibility in the display of text information. By using renditions from a resource file, for example, a Motif programmer can create one application that can be useful in many different countries without being recompiled. Motif uses compound strings to display all text, except in the Text and TextField widgets.

This chapter introduces the structure of a compound string and its components. The data types associated with compound strings are reviewed, and the functions used to create, modify, and handle these strings are covered as well. Though this chapter will serve as an introduction to these data types and functions, they are not described completely here. For the complete definitions of each function and data type, please refer to the *Motif 2.1—Programmer's Reference*.

9.1 The Structure of a Compound String

A compound string is an opaque data structure consisting of one or more pieces of encoded text, and information about how to display each piece.

- **XmString**

This information consists of the direction in which to draw the text, and the name (also called the "tag") of one or more "renditions," which specify attributes of the drawn text, such as the font, the color of the text and its background, whether the text should be underlined or struck through, and a list of tab stops. The data type for a compound string is **XmString**.

A compound string is divided into "components," which identify the various parts of the compound string. A compound string component could be the text itself, without the rendition and direction, or it could be the direction indicator alone, or the rendition tag itself.

There are several different kinds of compound string components.

- **XmStringComponentType**
- **XmStringDirection**
- **XmStringTag**

Each of them consists of a component identifier (data type **XmStringComponentType**), and the component value, usually in the form of a text string. The text string might contain the text of the compound string, but it also might contain the tag of a rendition to use, the name of a character set, or the name of the current locale. For a complete list of the possible values of **XmStringComponentType**, and the order in which they should appear, please refer to the **XmStringComponentType** page of the *Motif 2.1—Programmer's Reference*.

The direction of a compound string is stored so that the data structure will be equally useful for describing text in left-to-right languages such as English, Spanish, French, and German, and for text in right-to-left languages such as Hebrew and Arabic. The direction is stored as data type **XmStringDirection**, which may have the values **XmSTRING_DIRECTION_L_TO_R** or **XmSTRING_DIRECTION_R_TO_L**. A widget may itself have a default direction, called its "layout direction." For information about the interaction between a widget's layout direction and the string direction of a compound string to be displayed in that widget, please see chapter 11 of this manual, "Internationalization."

Another important compound string component is the "separator," which contains no text. When displayed, a separator usually maps to a line break. Do not confuse these with the Separator widget, which is also called **XmSeparator**.

An **XmStringTag** indicates the name, or "tag," of a particular rendition. The rendition describes several attributes to use when rendering a compound string, such as the font to use, the color, a list of tab stops, and whether the text is to be displayed underlined or struck out. Each rendition contains an **XmStringTag** that must match the tag included in the string itself.

Note: The rendition tag should not be confused with the charset tag, which exists for compatibility with previous releases of Motif, or the locale tag, which specifies the locale for internationalized applications. In a compound string, these tags are specified with the **XmSTRING_COMPONENT_TAG** component, while the rendition tags are specified in the **XmSTRING_COMPONENT_RENDITION_BEGIN** and **XmSTRING_COMPONENT_RENDITION_END** components. The charset tag was called the fontlist tag in previous releases of Motif.

Several renditions together make up a render table, and a compound string component can actually be associated with more than one tag, indicating more than one rendition. Multiple renditions will be merged when the text is displayed. (See section 9.2.3 for a more complete description of the merging process.)

A rendition may also be indicated through a locale specification. See the chapter of this manual on Internationalization for more information about locales.

The rendition table and rendition tag structure parallels the old font list and font list element tag structure of earlier releases of Motif. The rendition structure has augmented, not supplanted, the font list. Font Lists should be considered obsolete, and their use should be avoided. However, for the purposes of compatibility with existing software, the font management function calls and data structures still exist. If both a font list and a render table are specified for some widget, the render table will take precedence, and the font list will be ignored.

9.1.1 Creating a Compound String

There are several different functions available to create a compound string.

- **XmStringGenerate**
- **XmStringComponentCreate**

The most basic such function is **XmStringGenerate**, which accepts as arguments the text to be encoded, and a rendition tag to use to display it. It also accepts information about the locale or charset to use in rendering the text. The function returns an encoded **XmString**.

The **XmStringComponentCreate** function is used to create individual components of a compound string, such as the text, the tag, the direction, or the separator, which can then be concatenated into a single longer string.

There are a few functions that create a compound string which are obsolete, but are included for compatibility with earlier releases of Motif.

- **XmStringCreate**
- **XmStringSegmentCreate**
- **XmStringSeparatorCreate**

XmStringCreate may also be used to create a compound string of charset type with a specified font tag, with no rendition specified. The **XmStringCreate** function has been superseded, but is included for backward compatibility. Its function can be easily duplicated by calling **XmStringGenerate** with the text type **XmCHARSET_TEXT**, a charset tag, and a NULL rendition tag. **XmStringSegmentCreate** functions in the same manner to **XmStringCreate**, but allows the programmer to specify the string direction, and whether a separator should be appended to the returned **XmString**.

Earlier releases of Motif also included the functions **XmStringCreateLtoR** and **XmStringCreate**. These are still available, but again only for purposes of backward compatibility with existing code. Use **XmStringGenerate** instead.

The **XmStringSeparatorCreate** function may also be used instead of **XmStringSegmentCreate** to create a separator component for a compound string. In display, the separator would typically be mapped to a line break. Note that this function will not do anything more than **XmStringSegmentCreate** would if called with an empty text string.

The application that calls the compound string creation functions is responsible for performing its own memory management. The creation functions all must allocate space in which to return the strings.

- **XmStringFree**

When an application is finished with a compound string, the **XmStringFree** function should be used to free the memory that had been used for that string, and to make it available for some other purpose.

The following code sample demonstrates the use of **XmStringGenerate** and **XmStringFree** to create and display a compound string in a PushButton gadget.

```
void RendCB(Widget, XtPointer, XtPointer);
XmString        rend_label1, rend_label2;
widget  rend_button1, rend_button2;
XmRenderTable RenderTable;
...

/* RenderTable created with two renditions:
 * Rendition1 and Rendition2 */

...
RendPullDown = (Widget)XmCreatePulldownMenu(menubar, "RendPullDown",
                                            NULL, 0);

rend_label1=XmStringGenerate("render1", NULL, XmCHARSET_TEXT,
                             "Rendition1");
rend_label2=XmStringGenerate("render2", NULL, XmCHARSET_TEXT,
                             "Rendition2");

rend_button1 = XtVaCreateManagedWidget("rend1", xmPushButtonGadgetClass,
                                RendPullDown,
                                XmNrenderTable, RenderTable,
                                XmNlabelString, rend_label1,
                                NULL);
rend_button2 = XtVaCreateManagedWidget("rend2", xmPushButtonGadgetClass,
                                RendPullDown,
                                 XmNrenderTable, RenderTable,
                                 XmNlabelString, rend_label2,
                                 NULL);
XtAddCallback(rend_button1, XmNactivateCallback, RendCB, (XtPointer) 0);
XtAddCallback(rend_button2, XmNactivateCallback, RendCB, (XtPointer) 1);

XmStringFree(rend_label1);
XmStringFree(rend_label2);
```

Note that the description of **Rendition1** and **Rendition2** must be explicitly sent to the widgets by using the **XmNrenderTable** resource to specify the render table containing these renditions. See section 9.2 about renditions and render tables. Also note that **rend1** and **rend2** are the *names* of the PushButton widgets; they are not the *labels* that will appear on those widget.

9.1.1.1 Compound Strings and Resource Files

In an internationalized program, the labels and messages used to communicate with the user should be stored independently of the binary code of the compiled application, generally in external files. Under these conditions, adapting a program to different locales becomes as simple as replacing the external files. The compound strings used in Motif button labels and menu titles can be retrieved from X resource files.

In the preceding code example, the button label could have come from a resource file instead of being coded into the program. For this example, assume that the PushButtons are children of a widget called **RendPullDown**. The following example specifies two compound strings with the same rendition.

```
*RendPullDown.rend_button1.labelString:  render1
*RendPullDown.rend_button2.labelString:  render2
```

Here, Motif’s string-to-compound-string converter produces a compound string from the resource file text. If the rendition is not specified, as in this example, the converter uses the rendition corresponding to the tag **_MOTIF_DEFAULT_LOCALE**. The format for specifying a rendition tag is analogous to the format for specifying a widget. The following example will draw the **render1** label with the rendition tagged by **rendition1**, and draw the **render2** label with the **rendition2** rendition. A later section of this chapter describes how a rendition and render table can be specified in a resource file.

```
*RendPullDown.rend_button1.renderTable.rendition1:  render1
*RendPullDown.rend_button2.renderTable.rendition2:  render2
```

Note that, when a compound string is specified in this manner, the string direction is inherited from the widget **XmNlayoutDirection** resource.

9.1.2 Compound String Manipulation

To make working with compound strings as simple as using regular character strings, Motif contains a number of functions analogous to the basic string manipulation functions generally available with C.

- **XmStringCompare**
- **XmStringConcat**
- **XmStringConcatAndFree**
- **XmStringCopy**
- **XmStringEmpty**
- **XmStringIsVoid**

Use **XmStringCompare** to determine whether its two input compound strings are identical. **XmStringConcat** creates a third string by concatenating its two input strings. **XmStringConcatAndFree** does the same thing, except that it also deletes the two input strings and frees the memory occupied by them. **XmStringCopy** allocates enough memory to hold a copy of its input string, and returns a copy there. The **XmStringEmpty** function returns a Boolean value of True if there is no printable text in the input compound string. The **XmStringIsVoid** function is the same as the **XmStringEmpty** function except that it will only return True if there is no text at all in the string, printable or unprintable.

The following example takes an existing compound string called *string*, and sandwiches it with a new rendition, called *rendition*. This may also be accomplished with the **XmStringPutRendition** function (see section 9.2.3.1).

```
XmString  string;
XmString  tmp1, tmp2;
...
```

```
tmp1 = XmStringComponentCreate(
             XmSTRING_COMPONENT_RENDITION_BEGIN, "rendition");
tmp2 = XmStringComponentCreate(
             XmSTRING_COMPONENT_RENDITION_END, "rendition");

string = XmStringConcat(tmp1, string);
XmStringFree(tmp1);
string = XmStringConcat(string, tmp2);
XmStringFree(tmp2);
```

Display of the new string will use a composite rendition formed through a merge of the pre-existing rendition with the *rendition* rendition. See section 9.2.3 on render tables for the rules of merging renditions.

9.1.3 Reading a Compound String

Motif provides two different functions to read the contents of a compound string.

- **XmStringGetNextTriple**
- **XmStringPeekNextTriple**

The **XmStringGetNextTriple** function is the most basic of the compound string readers. Given an initialized "compound string context," the function returns successive of the nominated compound string on successive calls. Use **XmStringPeekNextTriple** to determine the type of the next component without reading it, and without advancing the counter. A subsequent call to **XmStringGetNextTriple** would return the value of the component just peeked at.

In order to read a compound string with **XmStringGetNextTriple**, or to peek forward with **XmStringPeekNextTriple**, a program must first establish a "compound string context" to use. This is just a state structure used to keep track of how far into a compound string the program has read.

- **XmStringInitContext**
- **XmStringFreeContext**

Motif provides the **XmStringInitContext** function to read a compound string and allocate a buffer long enough to handle any of its components. As with all of the other Motif calls that allocate memory, the calling application is responsible for doing its own memory management. Motif provides a function, **XmStringFreeContext**, to free the memory associated with the context.

Two convenient functions are also provided to do two commonly needed tasks: searching for a substring and counting the lines of text.

- **XmStringHasSubString**
- **XmStringLineCount**

The **XmStringHasSubString** function takes two compound strings, the target string and the search string, as its arguments. The function returns True if the search string (which can only have one text component) is found in the target string.

The **XmStringLineCount** function simply returns the number of separators found in the input compound string, plus one. Since the separators usually map to a line terminator, this would be the number of lines in the displayed compound string.

9.1.3.1 Storing a Compound String in a File

The **XmString** data type is opaque to the application programmer. In addition to the text, a compound string also contains data to tell a widget how to render the string. Some of this data may be in the form of pointers to various addresses, while other data may contain values.

If a program needs to store a compound string, or pass it to some other program, it must first convert the string to some more portable format, independent of any memory references. Motif provides two functions to convert a compound string into a byte stream, and to reverse the process.

- **XmCvtXmStringToByteStream**
- **XmCvtByteStreamToXmString**
- **XmStringByteStreamLength**

The **XmCvtXmStringToByteStream** function converts the text of a compound string and all its ancillary information, including rendition tags, direction indicators, and separators, into a byte sequence in an ASN.1-compliant format. This output string can then be written to a file, or passed to some other program. The size of the returned byte stream is internally specified in the ASN.1-format header, and can be returned with the **XmStringByteStreamLength** function. Once in hand, a program can call **XmCvtByteStreamToXmString** to convert the given byte stream back into a compound string.

Note that the compound string may contain rendition tags, and these are not resolved into the renditions they indicate. The text of the rendition tags is actually transcribed into the byte stream output of **XmCvtXmStringToByteStream**. For any other program to make sense of the given data, it may need a copy of the render table, too. Motif provides two functions to convert render tables to and from a portable format:

- **XmRenderTableCvtFromProp**
- **XmRenderTableCvtToProp**

The **XmRenderTableCvtToProp** function takes a render table and outputs a character string containing the resource functions and values for each of the render table's renditions. Use **XmRenderTableCvtFromProp** to convert such a character string table back into a Motif render table. Refer to section 9.2.3 for more information about renditions and render tables.

9.1.4 Displaying a Compound String

In most cases, a programmer need not worry about displaying a compound string. If the string is submitted to a widget, that widget is responsible for figuring out how to draw its own labels and text. Therefore, the functions described in this section are not used by most Motif applications. However, if a programmer chooses to *write* a widget, these functions will be quite useful.

- **XmStringDraw**
- **XmStringDrawImage**
- **XmStringDrawUnderline**

The **XmStringDraw** function draws a compound string in a specified window, in a specified display. The **XmStringDrawImage** function is nearly identical, except that, when drawing the string, it uses both the foreground and background bits of each character, where **XmStringDraw** only uses the specified foreground color.

XmStringDrawUnderline is the same as **XmStringDraw**, with the addition of one more argument, also a compound string. If this compound string matches any pattern in the compound string to be drawn, that part of the drawn string will be underlined.

In order to draw a string properly, it may be necessary to predict the drawn string's dimensions.

- **XmStringExtent**
- **XmStringHeight**
- **XmStringWidth**
- **XmStringBaseLine**

The **XmStringExtent** function returns the height and width of the smallest rectangle that could enclose the given compound string. The **XmStringHeight** and **XmStringWidth** functions return only the height or width of the string. Use **XmStringBaseLine** to find the number of pixels between the tallest possible letter and the baseline of the first line of text in the input compound string.

9.1.5 Compound String Tables

Motif now provides support for a table of compound strings. This is simply an array of compound strings, typically to be used to display or accept tabular information. Motif provides functions for creating and manipulating such arrays.

- **XmStringToXmStringTable**
- **XmStringTableToXmString**

The **XmStringToXmStringTable** function takes a single compound string as its input, along with a compound string component, *break_component*. Upon return, the string is broken up, wherever the specified *break_component* appears, and the function returns an array (table) of the resulting compound strings. The component marking the breaks will not appear in the output table.

Use **XmStringTableToXmString** to convert a table of compound strings back into a single compound string. This function also uses a *break_component* argument, which it inserts into the resulting compound string between each of the elements of the input table.

The following two functions convert regular text to a table of compound strings and back again. They both use the parsing capabilities of Motif, which are described in section 9.4.

- **XmStringTableParseStringArray**
- **XmStringTableUnparse**

The **XmStringTableParseStringArray** function accepts an array of simple text strings, calls **XmStringParseString** on each one with the given parse table, and puts the result into a table of compound strings. The **XmStringTableUnparse** function also uses an input parse table, to convert the elements of the input compound string table back into regular text strings. See section 9.4 for more information about parsing and parse tables.

9.2 Renditions

A rendition (data type **XmRendition**) is an opaque data type used to specify data used in rendering compound strings.

- **XmRendition**

A rendition has two parts: the set of data used to render a compound string (such as fonts, colors, and tabs), and a name, or "tag" (data type **XmStringTag**), by which to identify it.

9.2.1 Creating a Rendition

A rendition is neither a widget nor a gadget. However, the style used to specify resources for widgets and gadgets is a simple and familiar way to set data values, and it has been implemented for renditions well. To change the foreground color for a rendition, the application programmer must set a "resource" called **XmNrenditionForeground** for that rendition, in a manner identical to that of a programmer changing a color resource for some widget. For a complete description of the XmRendition resource set, see the XmRendition page in the *Motif 2.1—Programmer's Reference*.

- **XmRenditionCreate**

XmRenditionCreate takes an argument list (use **XtSetArg** to create the list) of resource name and value pairs as input, and returns a rendition, which can then be entered into the render table.

The following code fragment establishes a rendition that would display a compound string in the 8x16 font, in blue, and underlined with a single line.

```
int n;
XmRendition Rendition;
XmStringTag RenditionTag;

   XtVaGetValues(parent, XmNcolormap, &cmap, NULL);
   if (XAllocNamedColor(XtDisplay(parent),cmap,"blue",&color,&unused))
    {
      pixel_color = color.pixel;
    } else {
      pixel_color = XmUNSPECIFIED_PIXEL;
    }

   n = 0;
   XtSetArg( args[n], XmNrenditionForeground, pixel_color); n++;
   XtSetArg( args[n], XmNfontName, "8x16" ); n++;
   XtSetArg( args[n], XmNfontType, XmFONT_IS_FONT ); n++;
   XtSetArg( args[n], XmNunderlineType, XmSINGLE_LINE );  n++;
   RenditionTag = (XmStringTag) "Rendition1";
   Rendition = XmRenditionCreate( parent, RenditionTag, args, n );
```

9.2.2 Editing a Rendition

These functions are used to read and modify a rendition.

- **XmRenditionRetrieve**
- **XmRenditionUpdate**

The **XmRenditionUpdate** function works in a similar way to **XmRenditionCreate**, merging the given argument list into the definition of the rendition. Resources in the input ArgList that were **XmAS_IS** will be set to the given values, and resources in the input ArgList that were already set will be reset to the new value. Resources not in the input ArgList will remain unchanged. The **XmRenditionRetrieve** function also uses an ArgList, but the user must specify the address (instead of the value) of each resource value to be returned.

The following code retrieves a rendition. If the rendition was underlined, the underline is removed. If the rendition was not underlined, the underline is inserted.

```
int n;
unsigned char underline;

n = 0;
XtSetArg( args[n], XmNunderlineType, &underline ); n++;
XmRenditionRetrieve( Rendition, args, n );

if (( underline == XmAS_IS) || ( underline == XmNO_LINE ))
     underline = XmSINGLE_LINE;

else if ( underline == XmSINGLE_LINE)
      underline = XmNO_LINE;

n = 0;
XtSetArg( args[n], XmNunderlineType, underline ); n++;
XmRenditionUpdate( Rendition, args, n );
```

The modified rendition must still be added to the render table, and the modified render table sent back to the widget, before it is accessible to a widget needing to display a compound string.

- **XmRenditionFree**

The **XmRenditionCreate** function allocates memory for the created rendition structure. The application calling this function is responsible for maintaining its own memory management scheme. Use **XmRenditionFree** to recover memory space allocated to a rendition when the application is done with it. The application may dispose of the rendition as soon as it has been added to a render table.

9.2.3 Render Tables

A collection of renditions, each identified by a rendition "tag" (type **XmStringTag**), constitutes a render table (type **XmRenderTable**). Each rendition contains specifications describing how to display text, including the font, the color, the tab stops, and other information.

- **XmRenderTable**

A compound string can specify its rendition in more than one way. The typical way is simply for each compound string text component to be associated with a rendition tag, which is matched with the entry in the render table with the same tag. However, an application may use the default rendition for the current locale (tag=**_MOTIF_DEFAULT_LOCALE**). Alternatively, older applications may specify a charset tag, which will identify a font to use to display some text. The charset and the locale options may not both be specified, but both may exist with one or more rendition tags.

Renditions are accumulated as a compound string is read. If there have been three rendition begin tags and only one corresponding rendition end tag preceding a particular segment of text, then there are two renditions associated with that text. The *last* rendition specified that is still in effect is the primary rendition. If a segment of text is associated with more than one rendition, and if there are unspecified values in the primary rendition, the widget must create an effective rendition for that segment. This is formed by using the previous (active) rendition to fill in the unspecified values of the primary rendition. If this effective rendition still has unspecified values, then the next active rendition back is used, and so on. Finally, if the resulting rendition still has resources with unspecified values and the segment has a locale or charset tag (these are optional and mutually exclusive) this tag is matched with a rendition in the render table, and the missing rendition values are filled in from that entry.

If no matching rendition is found for a particular tag, then the **XmNoRenditionCallback** of the **XmDisplay** object is called and the render table is searched again for that tag.

If the resulting rendition does not specify a font or fontset, then for segments whose text type is **XmCHARSET_TEXT**, the render table will be searched for a rendition tagged with **XmFONTLIST_DEFAULT_TAG**, and if a matching rendition is found, it will be merged into the current rendition. If the resulting rendition contains no font or fontset, the **XmNnoFontCallback** will be called with the default rendition and "" as the font name. If no rendition matches or no font was found after the callback, then the first rendition in the render table will be merged into the current rendition. If this rendition still has no font, then the segment will not be rendered and a warning will be issued.

For segments whose text type is **XmMULTIBYTE_TEXT** or **XmWIDECHAR_TEXT**, the render table will be searched for a rendition tagged with **_MOTIF_DEFAULT_LOCALE**, and if a matching rendition is found, it will be merged into the current rendition. If the resulting rendition contains no font, the **XmNnoFontCallback** will be called with the default rendition and "" as the font name. If no rendition matches or no font was found after the callback, then the segment will not be rendered and a warning will be issued.

For example, imagine a render table containing three renditions. One belongs to the default locale, and has the tag **_MOTIF_DEFAULT_LOCALE**, while the other two renditions are called *fred* and *susan*. Further suppose the default locale rendition specifies a font, but the foreground and background color resources are **XmUNSPECIFIED_PIXEL**. Rendition *fred* specifies the foreground color as red, but the background field is **XmUNSPECIFIED_PIXEL**, and rendition *susan* gives blue for the foreground and purple for the background. Following is a schematic representation of such a render table:

```
_MOTIF_DEFAULT_LOCALE:
      XmNfontName:   variable
      XmNfontType:   XmFONT_IS_FONT
      XmNrenditionForeground: XmUNSPECIFIED_PIXEL
      XmNrenditionBackground: XmUNSPECIFIED_PIXEL
      ...
fred:
      XmNfontName:   XmAS_IS
      XmNfontType:   XmAS_IS
      XmNrenditionForeground: red
      XmNrenditionBackground: XmUNSPECIFIED_PIXEL
      ...
susan:
      XmNfontName:   XmAS_IS
      XmNfontType:   XmAS_IS
      XmNrenditionForeground: blue
      XmNrenditionBackground: purple
      ...
```

A text component associated with the *fred* and *susan* tags (in that order) will have red type on a purple background. But a component using *susan* and *fred* will have blue type on a purple background. Both components will use the font corresponding to the locale rendition, *variable* in this case. Note that the order of the renditions in the merge sequence is the opposite of the order in which the corresponding **XmSTRING_COMPONENT_RENDITION_BEGIN** components appear in the compound string. That is, the primary rendition for some piece of text corresponds to the *last* rendition component read.

If the search of the render table results in no font or fontset, then if there is a rendition in the render table with a tag of **_MOTIF_DEFAULT_LOCALE**, and if that rendition specifies a font, then that font will be used. If no font or fontset is specified at this point, the text component will not be rendered and a warning message will be displayed.

9.2.3.1 Creating a Render Table

Before creating a render table, an application program must first have created at least one of the renditions that will be part of the table.

- **XmRenderTableAddRenditions**

As the name implies, this function is also used to augment a render table with new renditions. To create a new render table, call the **XmRenderTableAddRenditions** function with a NULL argument in place of an existing render table. The following code creates a render table from three renditions.

```
n = 0;
XtSetArg( args[n], XmNtopAttachment, XmATTACH_FORM ); n++;
XtSetArg( args[n], XmNwidth, LIST_WIDTH ); n++;
XtSetArg( args[n], XmNvisibleItemCount, NUM_ITEMS ); n++;
List = XmCreateList( Manager, "List", args, n );
XtManageChild(List);

n = 0;
XtSetArg( args[n], XmNfontName, "fixed" ); n++;
XtSetArg( args[n], XmNfontType, XmFONT_IS_FONT ); n++;
XtSetArg( args[n], XmNunderlineType, XmNO_LINE );  n++;
Renditions[0] = XmRenditionCreate( List, (XmStringTag)tags[0],
                  args, n );

n = 0;
XtSetArg( args[n], XmNfontName, XmAS_IS ); n++;
XtSetArg( args[n], XmNfontType, XmFONT_IS_FONT ); n++;
XtSetArg( args[n], XmNunderlineType, XmSINGLE_LINE );  n++;
Renditions[1] = XmRenditionCreate( List, (XmStringTag)tags[1],
                  args, n );

n = 0;
XtSetArg( args[n], XmNfontName, "variable" ); n++;
XtSetArg( args[n], XmNfontType, XmFONT_IS_FONT ); n++;
XtSetArg( args[n], XmNunderlineType, XmDOUBLE_LINE );  n++;
Renditions[2] = XmRenditionCreate( List, (XmStringTag)tags[2],
                  args, n );

RenderTable =
    XmRenderTableAddRenditions( NULL, Renditions, 3,
                            XmREPLACE );

n = 0;
XtSetArg( args[n], XmNrenderTable, RenderTable ); n++;
XtSetValues( List, args, n );
```

Note that the list widget must specify the resulting render table in its **XmNrenderTable** resource in order for that widget to have access to the render table’s rendition data.

- **XmStringPutRendition**

To create a new compound string with a new rendition out of an old compound string, use the **XmStringPutRendition** function. Given an **XmString** and the tag of a rendition from a render table, this function places **XmSTRING_COMPONENT_RENDITION_BEGIN** and **XmSTRING_COMPONENT_RENDITION_END** components containing the input rendition tag around a copy of the old string. This function does not strip old rendition data out of the string, so the resulting string will merge the existing rendition with the new one when it is rendered. See the example in section 9.1.2.

9.2.3.2 Editing a Render Table

The **XmRenderTableAddRenditions** may also be used to update an existing render table by adding, replacing, or merging new renditions into the table. The following functions also provide important editing functionality.

- **XmRenderTableGetRenditions**
- **XmRenderTableRemoveRenditions**
- **XmRenderTableCopy**
- **XmRenderTableGetTags**

These functions are used to read and modify render tables. Use the **XmRenderTableGetRenditions** function to call up one or more renditions for a table. A program might do this in order to modify some of the renditions in the table. The **XmRenderTableRemoveRenditions** function will delete any unneeded renditions from a given table. A related function is **XmRenderTableCopy**, which will copy only renditions matching the input list of rendition tags into a new render table. Use **XmRenderTableGetTags** to retrieve a list of tags from a render table.

The following code uses the render table created in the last section, and modifies the *variable* rendition to include a single line "strike through." (This produces a horizontal line through the drawn text, as if the text had been crossed out.)

```
int n;
XmRendition Rendition;

     Rendition = XmRenderTableGetRenditions(RenderTable,
                      (XmStringTag) "variable", 1);

     n = 0;
     XtSetArg( args[n], XmNstrikethruType, XmSINGLE_LINE ); n++;
     XmRenditionUpdate( Rendition, args, n );

     XmRenderTableAddRenditions(RenderTable, Rendition, 1,
                      XmMERGE_NEW);

     n = 0;
     XtSetArg( args[n], XmNrenderTable, RenderTable ); n++;
     XtSetValues( List, args, n );
```

Note that the modified render table must be sent again to the widgets in which it will be used before becoming effective.

- **XmRenderTableFree**

Although the **XmRenderTableAddRenditions** and **XmRenderTableCopy** functions allocate memory for a render table, the application program is responsible for its own memory management. Use the **XmRenderTableFree** function to free memory when a render table is no longer needed. Note, however, that the **XmRenderTableAddRenditions** function automatically frees the memory occupied by its input render table.

9.2.3.3 Creating a Render Table in a Resource File

Renditions and render tables may be specified in resource files. For a properly internationalized application, in fact, this is the preferred method. When the render tables are specified in a file, the program binaries are made independent of the particular needs of a given locale, and may be easily customized to local needs. Render tables are specified in resource files with the following syntax:

```
resource_spec: [ tag [, tag ]* ]
```

where *tag* is some string suitable for the **XmNtag** resource of a rendition. This line creates an initial render table containing one more renditions than the number of tags specified. The renditions are attached to the specified tags, with the untagged rendition going with the tag **_MOTIF_DEFAULT_LOCALE**. If no tags are specified, then a render table will be created that contains only a rendition with a tag of **_MOTIF_DEFAULT_LOCALE**.

Specific values for specific rendition resources are specified using the following syntax:

```
resource_spec [*|.] rendition[*|.] resource_name:value
```

where *resource_spec* specifies the render table, *rendition* is either the class Rendition or a tag, *resource_name* is either the call or name of a particular resource, and *value* is the specification of the value to be set.

Any resource line that consists of just a resource name or class component with no rendition component or loose binding will be assumed to specify resource values for a rendition with a tag of **_MOTIF_DEFAULT_LOCALE**.

For example, the following:

```
*List.renderTable: green, variable
*List.renderTable.green.renditionForeground: Green
```

```
*List.renderTable.green.fontName: AS_IS
*List.renderTable.variable.underlineType: SINGLE_LINE
*List.renderTable.variable.renditionForeground: Red
*List.renderTable.variable.fontName: variable
*List.renderTable.variable.fontType: FONT_IS_FONT
*List.renderTable.renditionForegound: black
*List.renderTable.fontName: fixed
*List.renderTable.fontType: FONT_IS_FONT
*List.renderTable.variable.underlineType: NO_LINE
*List.renderTable*tabList: 1in, +1.5in, +3in
```

would set the **renderTable** resource of **List** to a render table consisting of three renditions tagged with **_MOTIF_DEFAULT_LOCALE**, *green*, and *variable* with values for resources set as described in the resource specifications. Note that the **tabList** resource will be shared by all three renditions.

9.2.3.4 Widget Resources and Render Tables

Several widget resource sets have been modified in Motif version 2.0 to accommodate the use of renditions and render tables. Several widgets have added the **XmNrenderTable** resource to their set. Several others have had more complex changes. The following list outlines the new resources for the affected widget classes. For complete information about the resource set of any widget, please refer to the appropriate section of the *Motif 2.1—Programmer's Reference*.

XmList, **XmLabel**, **XmLabelGadget**, **XmScale**, **XmText**, **XmTextField**
These widget classes use the **XmNrenderTable** resource to nominate their render table. The resource may be set at any time.

XmMenuShell
The **XmMenuShell** widget allows different render tables to be specified for different descendants. For button descendants, use the **XmNbuttonRenderTable** resource; and for label descendants, use the **XmNlabelRenderTable** resource. These resources may be set and reset at any time.

VendorShell, **XmBulletinBoard**
These widgets also allow different render tables for different kinds of descendants. The **XmNbuttonRenderTable** resource controls the render table for button descendants, the **XmNlabelRenderTable** controls any

label descendants, and the **XmNtextRenderTable** resource covers the text descendants. These resources may be set and reset at any time.

XmDisplay The **XmDisplay** widget has two resources that provide callback branches in case a descendant widget finds itself without enough information to render any given text. The **XmNnoFontCallback** resource nominates a callback routine to be invoked if a widget cannot find a font to use for any given compound string, and the **XmNnoRenditionCallback** resource nominates a callback in case a valid rendition cannot be found. These resources are of type **XtCallbackList**, and may only be set at the creation of the widget.

Upon execution of the callback routine nominated by either of these resources, a pointer to the following structure is passed to the **XmNnoFontCallback** and **XmNnoRenditionCallback** callbacks:

```
typedef struct
{
        int     reason;
        XEvent  *event;
        XmRendition     rendition;
        char    *font_name;
        XmRenderTable   render_table;
        XmStringTag     tag;
}XmDisplayCallbackStruct;
```

reason	Indicates why the callback was invoked. The possible values are **XmCR_NO_FONT** and **XmCR_NO_RENDITION**.
event	Points to the **XEvent** that triggered the callback. It can be NULL.
rendition	Specifies the rendition with the missing font.
font_name	Specifies the name of the font or font set which could not be loaded.
render_table	Specifies the render table with the missing rendition.
tag	Specifies the tag of the missing rendition.

9.2.3.5 Porting Render Tables

The render table is an opaque data type. If a programmer wishes to save a render table for some future use, or to pass the table to some other program, the application using the render table must first translate the render table into a portable format. Motif provides two functions to convert render tables to and from such a format:

- **XmRenderTableCvtFromProp**
- **XmRenderTableCvtToProp**

The **XmRenderTableCvtToProp** function takes a render table and outputs a character string containing the resource meanings and values for each of the render table's renditions. Use **XmRenderTableCvtFromProp** to convert such a character string table back into a Motif render table.

9.2.4 Font Lists

The font list paradigm of Motif 1 has been supplanted by renditions and render tables. The font list management routines are provided only for backward compatibility with older applications written. They should not be used for new applications, which can use renditions and render tables for the same effect. Following is a list of the remaining font list management functions:

- **XmFontListAppendEntry**
- **XmFontListCopy**
- **XmFontListEntryCreate**
- **XmFontListEntryFree**
- **XmFontListEntryGetFont**
- **XmFontListEntryGetTag**
- **XmFontListEntryLoad**
- **XmFontListFree**
- **XmFontListFreeContext**
- **XmFontListGetNextFont**

- **XmFontListInitFontContext**
- **XmFontListNextEntry**
- **XmFontListRemoveEntry**
- **XmFontListAdd** (superseded by **XmFontListAppendEntry**)
- **XmFontListCreate** (superseded by **XmFontListAppendEntry**)

9.3 Horizontal Tabs

To control the placement of text, a compound string can contain <tab> characters. To interpret those characters on display, a widget will refer to the rendition in effect for that compound string, where it will find a list of tab stops.

- **XmTab**
- **XmTabList**

A tab stop is encoded in an **XmTab** data type, and a **XmTabList** is an array of one or more **XmTab**s. A tab stop contains a tab value, the units of the value (such as inches, centimeters, pixels), the offset model, and the alignment model. The offset model dictates whether the tab measurement is absolute, measured from the left margin of the compound string display (or the right margin if the layout direction is right-to-left), or relative, measured from the previous tab stop. The tab stop information also contains information about the alignment of the text and the tab stop. A user could choose to align the tab stop with the beginning, center, or end of the text appearing at the stop. However, the alignment model is included only for future use, and the only alignment model supported by Motif will align the beginning of the following text with the tab stop.

Note that only the tab list associated with the first text component on any displayed line is used. If a rendition contains a NULL tab list, the default tab list is used. This consists of ten tabs, each of eight font units (**XmFONT_UNIT**).

9.3.1 Creating a Tab Stop

The following functions are used to create a single tab stop.

- **XmTabCreate**
- **XmTabGetValues**
- **XmTabSetValue**

Use the **XmTabCreate** convenience function to create a tab stop from a tab value, the tab units, and the offset model. To specify the offset model, there are the defined constants **XmABSOLUTE** and **XmRELATIVE**. Note that the tab value must be a positive number, and that, if a tab stop would cause a text component to overwrite the previous component, the x position of the component will be moved (the tab is ignored) to follow the first component. There is also an input argument to **XmTabCreate** to define the alignment of the text relative to the tab stop. Presently, **XmALIGNMENT_BEGINNING** is the only valid value.

Tab stops may be measured in any of the following units:

- **XmPIXELS**
- **XmINCHES**
- **XmCENTIMETERS**
- **XmMILLIMETERS**
- **XmPOINTS**
- **XmFONT_UNITS**

Font units are calculated from the dimensions of the font used. Horizontal font units (used in horizontal tabs) are calculated as follows:

- If the font has an **AVERAGE_WIDTH** property, the horizontal font unit is the **AVERAGE_WIDTH** property divided by 10.
- If the font has no **AVERAGE_WIDTH** property but has a **QUAD_WIDTH** property, the horizontal font unit is the **QUAD_WIDTH** property.
- If the font has no **AVERAGE_WIDTH** or **QUAD_WIDTH** property, the horizontal font unit is the sum of the font structure's *min_bounds.width* and *max_bounds.width* divided by 2.3.

The **XmTabGetValues** function separates a tab stop into its constituent parts, as outlined above, and the **XmTabSetValue** function is used to change only the *value* field, or the actual measurement of an existing tab stop. **XmTabSetValue** has no effect on the units, the offset model, or the alignment of a tab stop.

9.3.2 Creating a Tab List

Motif provides convenience functions to create tab lists in an application.

- **XmTabListInsertTabs**
- **XmStringTableProposeTablist**

The **XmTabListInsertTabs** function adds one or more **XmTab**s to an already existing **XmTabList**. Use it with an empty input tab list to create a tab list.

The **XmStringTableProposeTablist** function takes an **XmStringTable** structure containing tabbed compound strings and, using its rendition information and input about the desired padding between columns, returns a proposed tab list. This tab list is defined so that, if used to render the strings in the table, it would cause the strings to line up in columns with no overlap and with the specified amount of padding between the widest item in each column and the start of the next column. Each tab in the tab list would have the same unit type, offset model, and alignment type, specified on input.

Tab stops and tab lists may also be specified in a resource file. To create a tab stop in a resource file, use the following syntax:

```
resource_spec: tab [, tab ]*
```

The resource value string consists of one or more tabs separated by commas. Each *tab* identifies the value of the tab, the unit type, and whether the offset is relative or absolute. The tab stop specification uses the following syntax:

```
tab = [ + ]  float [ units ]
```

The tab contains a decimal number and an indication of the units to be used. If no units are specified, the default unit is pixels. The presence or absence of a sign indicates, respectively, a relative offset or an absolute offset model. A relative tab stop is measured from the previous tab stop in the list, while an absolute tab stop is measured from the left margin (or the right margin if the layout direction is right-to-left).

For example, the following line of a resource file sets a tab list for a rendition (**rend1**) in a render table (**render_table**) used by a widget called **List**. It specifies a tab list consisting of a 1-inch absolute tab followed by a 1-inch relative tab (equivalent to a 2-inch absolute tab, unless and until a third tab is inserted between the two):

```
*List.render_table.rend1.tabList: 1in, +1in
```

The recognized units and their abbreviations include:

pixels	*pix*, *pixel*, *pixels*
inches	*in*, *inch*, *inches*
centimeter	*cm*, *centimeter*, *centimeters*
millimeters	*mm*, *millimeter*, *millimeters*
points	*pt*, *point*, *points*
font units	*fu*, *font_unit*, *font_units*

See section 9.3.1 for more information about font units.

9.3.3 Editing a Tab List

These are the functions provided to maintain and modify a tab list:

- **XmTabListRemoveTabs**
- **XmTabListCopy**
- **XmTabListReplacePositions**
- **XmTabListTabCount**
- **XmTabListGetTab**

To remove tabs from a list, a program may use either **XmTabListRemoveTabs**, **XmTabListCopy**, or **XmTabListReplacePositions**. The first of these actually deletes the indicated **XmTab** entries from the list. The second and third functions create a new **XmTabList** using a subset of the original list. For **XmTabListCopy**, the subset must be in order in the original list, while **XmTabListReplacePositions** can pick an assortment of tab stops out of the old list to put in the new one.

The **XmTabListGetTab** function returns a copy of the tab stop at the indicated position in the list, and **XmTabListTabCount** simply returns the number of tab stops in a given tab list.

- **XmTabFree**
- **XmTabListFree**

The functions that create tabs and tab lists allocate memory in which to put them. When an application is finished with the tabs, it should use **XmTabFree** and **XmTabListFree** to free that memory for other uses.

9.4 Parse Tables

A *parse mapping* (data type **XmParseMapping**) is a fragment of multibyte text paired with a compound string component. When translating text into a compound string, using the **XmParse** functions, any occurrences of the parse mapping text will be translated into the compound string component with which it is paired in a parse mapping. Likewise, when converting a compound string back into regular text, each compound string component that matches a parse mapping will be translated into the corresponding text string. A *parse table* (data type **XmParseTable**) is nothing more than an ordered array of **XmParseMapping**s.

9.4.1 Structure of a Parse Mapping

Though the parse mapping is at heart nothing more than the text/compound string pairs, Motif stores each parse mapping with *resources* that indicate how the parsing is to proceed. Note that these are not widget resources, but the widget resource model provides a convenient way to specify and recall parse mapping instructions.

- **XmParseMappingCreate**
- **XmParseMappingGetValues**
- **XmParseMappingSetValues**

Following this model, an application uses the **XtSetArg** function to build a resource-style argument list to submit to the **XmParseMappingCreate**,

XmParseMappingGetValues, or **XmParseMappingSetValues** in the same manner that would be used to create or change the resource values of some widget.

The parse mapping "resources" are outlined briefly below. For more information about the **XmParseMapping** data type and its resources, please see the *Motif 2.1—Programmer's Reference*.

XmNpattern

Specifies a pattern to be matched in the text being parsed. At present, this is a maximum of one character.

XmNpatternType

Specifies the type of the pattern that is the value of **XmNpattern**. Following are the possible values:

- **XmCHARSET_TEXT** (default)
- **XmMULTIBYTE_TEXT**
- **XmWIDECHAR_TEXT**

XmNsubstitute

Specifies the compound string to be included in the compound string being constructed when **XmNincludeStatus** is **XmINSERT** or **XmTERMINATE**.

XmNincludeStatus

Specifies how the result of the mapping is to be included in the compound string being constructed. Unless the value is **XmINVOKE**, the result of the mapping is the value of **XmNsubstitute**. Following are the possible values for **XmNincludeStatus**:

XmINSERT Concatenate the result to the compound string being constructed and continue parsing. This is the default value.

XmINVOKE

Invoke the **XmNinvokeParseProc** on the text being parsed and use the returned compound string instead of **XmNsubstitute** as the result to be inserted into the compound string being constructed. The include status returned by the parse procedure (**XmINSERT** or **XmTERMINATE**) determines how the returned compound string is included.

XmTERMINATE
Concatenate the result to the compound string being constructed and terminate parsing.

XmNinvokeParseProc
Specifies the parse procedure to be invoked when **XmNincludeStatus** is **XmINVOKE**. This defaults to NULL. See the following section about the parse procedure.

XmNclientData
This is a **XtPointer** used to indicate data to be used by the parse procedure. The default value is NULL.

9.4.1.1 Parse Procedures

The Motif parsing mechanism allows for the execution of an arbitrarily complex parsing algorithm. If the simple substitution of compound string components for text strings is not powerful enough for a particular application, a programmer can define a *parse procedure* that will be invoked when certain text characters are read.

While parsing a text string, if an **XmNpattern** attached to a non-NULL **XmNinvokeParseProc** resource, matches part of the input string, the nominated parse procedure is invoked.

The **XmNinvokeParseProc** resource is a function of type **XmParseProc**, which is defined as follows:

```
XmIncludeStatus (*XmParseProc)(
        XtPointer       *text_in_out,
        XtPointer       text_end,
        XmTextType      type,
        XmStringTag     tag,
        XmParseMapping  entry,
        int     pattern_length,
        XmString        *str_include,
        XtPointer       call_data)
```

The input text is a pointer to the first byte of the pattern that was matched to trigger the call to the parse procedure. The parse procedure may use, modify, or remove as many bytes of the input string as it needs, as long as upon return the input text pointer is advanced to the following byte. It returns a compound string to be included in the compound string being constructed, as well as an **XmIncludeStatus**, corresponding to the parse mapping resource, indicating how the returned compound string should be handled.

text_in_out Specifies the text being parsed. The value is a pointer to the first byte of text matching the pattern that triggered the call to the parse procedure. When the parse procedure returns, this argument is set to the position in the text where parsing should resume—that is, to the byte following the last character parsed by the parse procedure.

text_end Specifies a pointer to the end of the *text_in_out* string. If *text_end* is NULL, the string is scanned until a NULL character is found. Otherwise, the string is scanned up to but not including the character whose address is *text_end*.

type Specifies the type of text and the tag type. If a locale tag should be created, *type* has a value of either **XmMULTIBYTE_TEXT** or **XmWIDECHAR_TEXT**. If a charset should be created, *type* has a value of **XmCHARSET_TEXT**.

tag Specifies the tag to be used in creating the result string. The type of string tag created (charset or locale) depends on the text type and the *tag* value passed to the parse procedure. If the *tag* value is NULL and if *type* indicates that a charset tag should be created, the string tag has the value **Xm_FONTLIST_DEFAULT_TAG**. If *type* indicates a locale tag, the string tag has the value **_MOTIF_DEFAULT_LOCALE**.

entry Specifies the parse mapping that triggered the call to the parse procedure.

pattern_length
Specifies the number of bytes in the input text, following *text_in_out*, that constitute the matched pattern.

str_include Specifies a pointer to a compound string. The parse procedure creates a compound string to be included in the compound string being constructed. The parse procedure then returns the compound string in this argument.

call_data Specifies data passed by the application to the parsing routine.

The parse procedure returns an **XmIncludeStatus** indicating how *str_include* is to be included in the compound string being constructed. Following are the possible values:

XmINSERT Concatenate the result to the compound string being constructed and continue parsing.

XmTERMINATE
Concatenate the result to the compound string being constructed and terminate parsing.

Upon return, the function that invoked the parse procedure will continue looking for matching patterns from the position of the text pointer. If the parse procedure has not advanced the pointer, and the text and parse mappings also remain the same, the parse procedure will be called again, perhaps infinitely.

Two parse procedures are provided as part of Motif.

- **XmeGetNextCharacter**
- **XmeGetDirection**

The **XmeGetNextCharacter** procedure consumes the byte the input pointer indicates, and the following byte, and returns a compound string component containing only the second byte. The effect is to copy the byte following the trigger byte directly into the output compound string, even if the normal parsing would not have done so. In other words, the trigger byte has served as an "escape" character. The **XmeGetNextCharacter** return value is **XmINSERT**, so parsing will continue normally from the same spot.

The other parse procedure included as a part of Motif is **XmeGetDirection**. This function simply returns an **XmString** component of type **XmSTRING_COMPONENT_DIRECTION**, whose value depends on the input text.

9.4.2 Use of a Parse Table

The following functions use a parse table to translate text into the compound string format and the reverse.

- **XmStringParseText**
- **XmStringUnparse**

To use a parse table, send it and a companion text string to the **XmStringParseText** function. This function also accepts a pointer to data that may be passed to a parse procedure, if one exists. The function will walk through the input text string, copying characters from it into a compound string; when it finds a pattern (specified by **XmNpattern** in the parse mappings) to match a pattern in the input parse table, it will not copy the pattern, but write the corresponding compound string (indicated by **XmNsubstitute** in the parse mapping) to the output **XmString**.

In the event it is desirable to translate a compound string back into a text string, the **XmStringUnparse** function is provided. This provides the precise converse to the **XmStringParseText** function, stepping through the input compound string and substituting text patterns for **XmString** components that match the **XmNsubstitute** resources in the parse table. If there is no parse procedure nominated in the parse table, the **XmStringUnparse** function called on the output of the **XmStringParseText** function will equal the input of **XmStringParseText**. A compound string created with a parse table that includes a parse procedure will not be correctly translated into simple text by a call to **XmStringUnparse**.

- **XmParseMappingFree**
- **XmParseTableFree**

The routines that create parse mappings allocate memory in which to store those mappings. It is the responsibility of the application, however, to free that memory when the mappings and tables are no longer needed. Use **XmParseMappingFree** and **XmParseTableFree** to free the memory for other uses.

9.5 Compound Strings in UIL

Text strings in UIL may be specified as simple string literals or as compound strings. Compound strings are created with the **COMPOUND_STRING** or the **COMPOUND_STRING_COMPONENT** functions. String literals are simply quoted text, and are stored either as NULL-terminated strings or as compound strings, depending on the complexity of the specified string. The UIL concatenation operator (**&**) concatenates both string literals and compound strings.

UIL also supports wide-character strings, which are documented in Chapter 11.

9.5.1 Simple Text Strings

A simple text string, or string literal, consists of text, a character set, and a layout direction. The character set may be explicitly specified, or a default value will be used. UIL infers the layout direction from the character set.

Some string literals will actually be stored by UIL as compound strings. This happens when

- A string expression consists of two or more literals with different character sets or writing directions, or
- The literal or expression in question is used as a value that has a compound string data type (such as the value of a resource whose data type is **XmString**).

UIL recognizes a number of keywords specifying character sets. UIL associates parsing rules, including parsing direction and whether characters have 8 or 16 bits, for each character set it recognizes. It is also possible to define a character set by using the UIL **CHARACTER_SET** function.

The syntax of a string literal is one of the following:

```
'[character_string]'
[#char_set]"[character_string]"
```

For each syntax, the character set of the string is determined as follows:

- For a string declared as **’***string***’**, the character set is the code set component of the **LANG** environment variable if it is set in the UIL compilation environment, or the value of **XmFALLBACK_CHARSET** if **LANG** is not set or has no code set. By default, the value of **XmFALLBACK_CHARSET** is ISO8859-1, but vendors may supply different values.
- For a string declared as *#char_set***"***string***"**, the character set is *char_set*. If the *char_set* in a string specified in this form is not a built-in charset, and is not a user-defined charset, the charset of the string will be set to **XmFONTLIST_DEFAULT_TAG**, and an informational message will be issued to the user to note that this substitution has been made.
- For a string declared as **"***string***"**, the character set depends on whether or not the module has a **character_set** clause and on whether or not the UIL compiler’s **use_setlocale_flag** is set:

— If the module has a **character_set** clause, the character set is the one specified in that clause.

— If the module has no **character_set** clause but the **uil** command was invoked with the **–s** option, or the **Uil** function was invoked with the **use_setlocale_flag** set, UIL calls **setlocale** and parses the string in the current locale. The character set of the resulting string is **XmFONTLIST_DEFAULT_TAG**.

— If the module has no **character_set** clause and the **uil** command was invoked without the **–s** option, or the **Uil** function was invoked without the **use_setlocale_flag**, the character set is the code set component of the **LANG** environment variable if it is set in the UIL compilation environment; if **LANG** is not set or has no code set, the character set is the value of **XmFALLBACK_CHARSET**.

Note that certain predefined escape sequences, beginning with a backslash (), may appear in string literals, with these exceptions:

- A string in single quotes can span multiple lines, with each newline escaped by a backslash. A string in double quotes cannot span multiple lines.
- Escape sequences are processed literally inside a string that is parsed in the current locale (a localized string).

9.5.2 Specifying Compound Strings

A programmer may create a compound string with the **COMPOUND_STRING** or the **COMPOUND_STRING_COMPONENT** functions. The **COMPOUND_STRING** function takes as arguments a string expression and optional specifications of a character set, direction, and whether or not to append a separator to the string. If no character set or direction is specified, UIL derives it from the string expression, in the same manner as for string literals, described in the previous section.

In order to create a complex compound string in UIL, it is necessary to use the **COMPOUND_STRING_COMPONENT** function. Use it to create compound strings in UIL consisting of single components, in a manner analogous to **XmStringComponentCreate**. This function lets you create simple compound strings containing components such as **XmSTRING_COMPONENT_TAB** and **XmSTRING_COMPONENT_RENDITION_BEGIN** that are not produced by the **COMPOUND_STRING** function. These components can then be concatenated to other compound strings to build more complex compound strings. (Use the UIL **&** operator to concatenate compound strings.)

The first argument of the **COMPOUND_STRING_COMPONENT** function must be an **XmStringComponentType** enumerated constant. The type and interpretation of the second argument depends on the first argument. For example, if you specify any of the following enumerated constants for the first argument, then you should not specify a second argument:

- **XmSTRING_COMPONENT_SEPARATOR**
- **XmSTRING_COMPONENT_LAYOUT_POP**
- **XmSTRING_COMPONENT_TAB**
- **XmSTRING_COMPONENT_LOCALE**

However, if you specify an enumerated constant from the following group, then you must supply a *string* as the second argument:

- **XmSTRING_COMPONENT_TAG**
- **XmSTRING_COMPONENT_TEXT**
- **XmSTRING_COMPONENT_LOCALE_TEXT**
- **XmSTRING_COMPONENT_WIDECHAR_TEXT**
- **XmSTRING_COMPONENT_RENDITION_BEGIN**
- **XmSTRING_COMPONENT_RENDITION_END**

If **XmSTRING_COMPONENT_DIRECTION** is the first argument, the second argument must be an **XmStringDirection** enumerated constant. Finally, if you specify **XmSTRING_COMPONENT_LAYOUT_PUSH** as the first argument, then you must specify an **XmDirection** enumerated constant as the second argument.

For more information on UIL string and compound string syntax, see the **UIL(5X)** reference page in the *Motif 2.1—Programmer’s Reference*.

9.5.3 Render Tables and Tab Lists

Render tables, renditions, tab lists, and tab stops are implemented as a special class of objects, in a form analogous to the widget class. Though a render table is not itself a widget, its specification lists resemble one, and a widget that wants to use a render table must specify the render table in its *controls* list, not just with the **XmNrenderTable** resource.

In the same manner, a render table *controls* list contains the names of the renditions that make it up, a rendition *controls* list contains the names of the tab lists it uses, and the tab list *controls* list contains the names of the tab stops it uses. Although none of these objects are actual widgets, the format reads exactly as if the render table were a widget controlling a rendition widget, which in turn controlled a tab list widget, which in turn controlled a tab stop widget.

The following sample of UIL code shows a scrolled list widget that uses a render table.

```
object
  Scrolled_List:  XmScrolledList {
     arguments {
        XmNlistSpacing =  10;
        XmNlistMarginWidth =  10;
        XmNlistMarginHeight =  10;
        XmNitems =  string_table
                ("item1", "item2", "item3", "item4",
                "item5", "item6", "item7");
        XmNitemCount =  7;
        XmNvisibleItemCount =  4;
        XmNselectionPolicy = XmSINGLE_SELECT;
        XmNrenderTable = rt1;
     };
     callbacks {
        XmNsingleSelectionCallback = procedure Report_Callback
                        ('singleSelectionCallback');
     };
     controls {
        XmRenderTable rt1;
     };
   };
```

```
object
  rt1: XmRenderTable {
     controls {
        XmRendition rend;
     };
  };

object
  rend: XmRendition {
     arguments {
        XmNtag = XmFONTLIST_DEFAULT_TAG;
        XmNfontName = ’9x15’;
        XmNfontType = XmFONT_IS_FONT;
     };
     controls {
        XmTabList tabl;
     };
  };

object
  tabl: XmTabList {
     controls {
        XmTab tabstop;
     };
  };

object
  tabstop: XmTab {
     arguments {
        XmNtabValue = 1.0;
        XmNunitType = XmCENTIMETERS;
        XmNoffsetModel = XmABSOLUTE;
     };
  };
```

The render table (**rt1**) in this sample only contains one rendition, with the tag **XmFONTLIST_DEFAULT_TAG**. If the tag were not explicitly specified, its value would default to the name of the rendition object, in this case *rend*. The rendition in question uses resources to specify a font, and names a tab list (*tabl*), which in turn names its only member tab stop (*tabstop*). The tab stop again uses resources in its *arguments* list to specify its properties.

Chapter 10

Text

Motif has widgets for displaying two kinds of text: static text, as in labels and messages, and editable text. Static text usually appears in Label widgets or Label subclasses, including buttons, and in Lists. The application or user can specify initial text for Labels or Lists by using resource or UIL files, but the user cannot edit the text. The application can replace the text during the program by setting the appropriate resources. In Labels and Label subclasses and in Lists, Motif represents text as compound strings. These are opaque data types that contain the text itself and tags that the toolkit matches with tags in render tables in order to select the appropriate fonts or font sets to display the text.

For editing text, Motif provides Text and TextField widgets. The displayed text in these widgets may or may not be editable, depending on the value of the **XmNeditable** resource. When the Text is editable and the user enters a text character, that character is inserted into the text. Other translations and actions allow the user to navigate or to select, cut, copy, paste, or scroll the text. In Text and TextField widgets, Motif represents text as strings of either multibyte (*char*) or wide (**wchar_t**) characters. The Text widget uses a single font or font set from a render table to display the text.

This chapter discusses the Text and TextField widgets. Labels and their subclasses are discussed in Chapter 5; and compound strings, render tables, and localization of text are discussed in Chapter 11. It is also possible for an application to construct its own text-editing widget by using a DrawingArea. This is discussed in Chapter 15.

10.1 Text and TextField

The Text widget uses the type **String**. A TextField is essentially the same as a Text widget in single-line mode, except that its performance is optimized for single-line text operations. Although TextField has a complete set of convenience routines of its own, the widget argument to the Text convenience routines can be either a Text or a TextField widget.

The text in a Text widget can be multiline or constrained to be a single line, depending on the value of the **XmNeditMode** resource. In multiline Text, pressing osfUp moves the insertion cursor, the point at which new text is inserted, physically upward, and pressing osfDown moves the insertion cursor physically downward. Other actions move the insertion cursor forward and backward by paragraphs. Pressing **Space**, **Tab**, or **Enter** causes the corresponding character to be inserted into the text. For this reason, some action key bindings are different in Text from those in other widgets, as shown in Table 10-1.

Table 10–1. Text Action Key Bindings

Action	**Actual Key Events**
Activate	**Ctrl<Key>Return**
	KeyosfActivate
Extend	**Ctrl Shift<Key>space**
	ShiftKeyosfSelect
NextField	**Ctrl<Key>Tab**
Select	**Ctrl<Key>space**
	KeyosfSelect

In a single-line widget, pressing **Space** still inserts a space into the text. However, osfUp and osfDown now move keyboard focus to the previous or next traversable widget, and **Tab** traverses to the next tab group. **Enter** invokes the **XmNactivateCallback** callbacks. The actions for moving by paragraphs have no effect. In other words, a single-line Text widget acts more as a simple control than a field control.

Note: Asian languages are supported by vertical writing capabilities; for detailed information, refer to the reference pages for the following interfaces: **XmText**, **XmTextField**, **XmTextGetBaseline**, **XmTextGetCenterline**, **XmTextPosToXY**, **XmTextXYToPos**, and **XmTextScroll**.

10.2 Data Transfer in Text

Text and TextField allow the user to cut, copy, and paste text by using the clipboard, primary transfer, or secondary transfer. The user can also drag and drop text within a widget, between widgets, or from a Label or List widget to a Text or TextField widget. In all cases, the user first selects text in some widget and then inserts the selected text into a Text or TextField widget.

This section explains how selection works in Text and TextField. Understanding selection requires understanding of several concepts: **primary selection**, **secondary selection**, **clipboard selection**, the *destination* widget, the **insertion cursor**, the selection *anchor*, and **pending delete**.

Selections are the primary means of exchanging data between X clients. A selection is a piece of data. Each display may have several kinds of selections, but only one selection of each kind can exist at any time on the display. A client owns each selection, and the selection is attached to a window. Clients can acquire or give up ownership of a selection and can request that the owner convert the selection into some data type and place the results on a property of a particular window. This mechanism makes it possible to select and then cut, copy, or paste data from one client to another. Selections are discussed in detail in the X Window System *Inter-Client Communication Conventions Manual* (ICCCM).

Text and TextField support transfers by using the three kinds of selection common to all X clients:

Primary	The primary selection is the principal selection on the display. Unless they are qualified, the terms *selecting text* and *the selection* refer to the primary selection.
Secondary	The secondary selection is used to transfer data without disturbing the primary selection. Text and TextField use the secondary selection for *quick* transfer, in which the user selects and then moves or copies text by using a single series of mouse gestures.
Clipboard	The clipboard selection usually holds data cut or copied from one client and available to be pasted into another. Text and TextField provide actions for cutting and copying text to the clipboard and for pasting text from the clipboard.

The *destination* is the widget that, at any particular time, would receive the selection if the user were to invoke a move, copy, or paste operation. A Text or TextField widget must be both sensitive and editable to become the destination. When the **XmNkeyboardFocusPolicy** of the shell is **XmEXPLICIT**, an editable widget becomes the destination when it receives keyboard focus. When the **XmNkeyboardFocusPolicy** is **XmPOINTER**, an editable widget becomes the destination when it receives any mouse button or keyboard input. If the destination widget becomes insensitive or uneditable, there is no destination widget.

The **insertion cursor** is an I-beam cursor that shows where text, including a selection, would be inserted in a Text or TextField widget. The insertion cursor appears as a solid I-beam when the widget is in **normal mode** (explained below) and when it is either the widget with keyboard focus or the destination widget. Otherwise, the insertion cursor appears as a stippled I-beam.

The *anchor* is a position in the text of a widget that marks one boundary of a selection or a potential selection. For example, the user can select a range of text by pressing, dragging, and releasing Btn1. The anchor is set at the point of the button press, and the selection extends to the point of the button release. When the user takes an action to extend an existing selection, Motif first adjusts the anchor by using a balance-beam method: it moves the anchor to the end of the existing selection that is farthest from the point of the button or key press that initiates the extend action.

Text and TextField have an **XmNpendingDelete** resource. This resource controls whether or not text is deleted in certain situations. Consider the following conditions:

- **XmNpendingDelete** is True (as it is by default).

- A selection exists and the insertion cursor is not disjoint from it.
- The widget is in Add Mode.

Given these conditions, an operation that inserts text (including a transfer of the secondary or clipboard selection) will delete the primary selection before inserting the text. Similarly, an operation that deletes text will delete the primary selection instead of the text that would otherwise be removed. When **XmNpendingDelete** is False, the insertion and deletion operations do not delete the selection.

Users typically use the mouse to make selections and to initiate and/or complete data transfers. (See the *CDE 2.1/Motif 2.1—User's Guide* for complete details.) The **XmDisplay** resource **XmNenableBtn1Transfer** controls the purpose of Btn1 and Btn2 in a Text, TextField, Container, or List widget. For example, when **XmNenableBtn1Transfer** is set to **XmOFF**, users press Btn1 to select text and Btn2 to trigger certain kinds of data transfer. (See **XmDisplay**(3) in the *Motif 2.1—Programmer's Reference* for details.)

Users can optionally use the keyboard instead of the mouse to perform selection and transfer operations. All operations available from the mouse, except secondary selection, are available from the keyboard. Text has two keyboard selection modes, Normal Mode and Add Mode.

In Normal Mode, if text is selected, a navigation operation deselects the selected text and moves the anchor to the current position of the insertion cursor before navigating. In Add Mode, navigation operations have no effect other than navigation.

In Normal mode, when the widget contains the primary selection and the insertion cursor is disjoint from it, any operation that inserts or pastes text into the widget (except a transfer of the primary selection from the same widget) first deselects the primary selection. In Add Mode, such an operation does not deselect the primary selection.

10.2.1 Mouse Selection

The user makes a primary selection with Btn1. Pressing Btn1 deselects any existing selection and moves the insertion cursor and the anchor to the position in the text where the button is pressed. Dragging Btn1 selects all text between the anchor and the pointer position, deselecting any text outside that range. Releasing Btn1 moves the insertion cursor to the position where the button is released. Clicking Btn1 deselects any existing selection and moves the insertion cursor and the anchor to the position where Btn1 is released.

ShiftBtn1 extends a selection using the balance-beam method. When the user presses ShiftBtn1, the selection becomes anchored at the edge of the selection farthest from the pointer position. When the user releases ShiftBtn1, the selection extends from the anchor to the position where ShiftBtn1 is released, and any text outside that range is deselected. The insertion cursor moves to the position where ShiftBtn1 is released.

Clicking CtrlBtn1 moves the insertion cursor to the position where CtrlBtn1 is released without affecting the selection.

Clicking Btn2 moves the insertion cursor to the position where Btn2 is released. Then, unless the insertion cursor is in the midst of the selection, it copies the primary selection to the insertion cursor and moves the insertion cursor to the end of the copied text. The original selection remains selected. Clicking ShiftBtn2 has the same effect except that it moves the primary selection to the insertion cursor, deleting the original selection if possible.

Dragging AltBtn2 outside of the primary selection starts a secondary selection consisting of all text between the position of the pointer and the position where AltBtn2 was pressed. Releasing AltBtn2 copies the secondary selection to the insertion cursor in the destination widget. Before copying the secondary selection, if the destination contains the primary selection and the insertion cursor is not disjoint from it, releasing AltBtn2 deletes the primary selection. Dragging AltShiftBtn2 also makes a secondary selection, and releasing AltShiftBtn2 moves the secondary selection to the destination widget.

Dragging Btn2 with the insertion cursor positioned within a primary selection initiates a drag operation. The user may press a modifier key to indicate whether the drag is a copy, move, or link operation. Releasing Btn2 either in the same Text widget or a different widget moves the insertion cursor to the position where Btn2 is released, drops the selected text at that point, and moves the insertion cursor to the end of the dropped text.

Pressing osfCancel during the operation aborts the operation and no data exchange occurs. If the user presses osfHelp over a drop site, the user has the option to continue or to cancel the drop operation in response to the help information that the application provides.

10.2.2 Keyboard Selection

Selection operations available with the mouse, except secondary selection, are also available from the keyboard. Text has two keyboard selection modes, Normal Mode and Add Mode. In Normal Mode, if text is selected, a navigation operation deselects the selected text and moves the anchor to the current position of the insertion cursor before navigating. In Add Mode, navigation operations have no effect other than navigation. In both modes, pressing osfSelect has the same effect as pressing Btn1 at that position.

In Normal mode, when the widget contains the primary selection and the insertion cursor is disjoint from it, any operation that inserts or pastes text into the widget (except a transfer of the primary selection from the same widget) first deselects the primary selection. In Add Mode, such an operation does not deselect the primary selection.

Pressing osfExtend extends the current selection to the insertion cursor using the balance-beam method. The current selection becomes anchored at the edge of the selection farthest from the insertion cursor. The selection then extends from the anchor to the insertion cursor, and any text outside that range is deselected.

Shifted navigation operations also extend a selection. In Normal Mode, if no text is selected, a shifted navigation operation moves the anchor to the insertion cursor, navigates, selects the navigated text, and deselects any text outside that range. In the remaining cases—Normal Mode and Add Mode with any selection—a shifted navigation operation extends the selection using the balance-beam method. Before navigation, the current selection becomes anchored at the edge of the selection farthest from the insertion cursor. After navigation, the selection extends from the anchor to the insertion cursor, and any text outside that range is deselected.

osfPrimaryCopy copies the primary selection to the insertion cursor. osfPrimaryCut cuts the primary selection to the insertion cursor.

osfCopy copies the current selection in the Text widget to the clipboard; osfCut cuts the selection; and osfPaste inserts the contents of the clipboard at the insertion cursor.

10.3 Text Editing and Callbacks

Text has a number of callback lists for communication with the application. Text invokes callbacks whenever the widget gains or loses focus, when it gains or loses the primary selection, before the insertion cursor is moved or text is modified, and when the text string changes or the activate() action is invoked.

Text passes these callbacks a pointer to either an **XmAnyCallbackStruct** or an **XmTextVerifyCallbackStruct** (or **XmTextVerifyCallbackStructWcs**) structure. The two verification structures contain the current and new positions of the insertion cursor, the starting and ending positions of the text to be modified, a pointer to an **XmTextBlockRec** (or **XmTextBlockRecWcs**) structure with information about the text to be modified, and a Boolean in/out *doit* member that the callback procedure can set to tell the widget whether or not to go ahead with the modification.

Following is a summary of the callbacks:

XmNmotionVerifyCallback
: Text invokes this list, passing a pointer to an **XmTextVerifyCallbackStruct** as the widget data, before moving the insertion cursor. The application can prevent the action by setting the *doit* member of the callback struct to False.

XmNmodifyVerifyCallback or **XmNmodifyVerifyCallbackWcs**
: Text invokes this list, passing a pointer to an **XmTextVerifyCallbackStruct** structure or an **XmTextVerifyCallbackStructWcs** structure as the widget data, before deleting or inserting any text. The application can prevent the action by setting the *doit* member of the callback struct to False.

XmNvalueChangedCallback
: Text invokes this list, passing a pointer to an **XmAnyCallbackStruct** as the widget data, after text is inserted or deleted.

XmNfocusCallback
: Text invokes this list, passing a pointer to an **XmAnyCallbackStruct** as the widget data, when the widget gains input focus.

XmNlosingFocusCallback
: Text invokes this list, passing a pointer to an **XmTextVerifyCallbackStruct** as the widget data, before the

widget loses input focus. The application can prevent the action by setting the *doit* member of the callback struct to False.

XmNgainPrimaryCallback
Text invokes this list, passing a pointer to an **XmAnyCallbackStruct** as the widget data, when the widget gains ownership of the primary selection.

XmNlosePrimaryCallback
Text invokes this list, passing a pointer to an **XmAnyCallbackStruct** as the widget data, when the widget loses ownership of the primary selection.

XmNactivateCallback
Text invokes this list, passing a pointer to an **XmAnyCallbackStruct** as the widget data, when the activate() action is invoked. By default no translations are bound to this action, but in a single-line Text widget or a TextField widget, pressing osfActivate invokes the**XmNactivateCallback** callbacks.

These callbacks provide a great deal of flexibility for an application to alter the behavior of the Text widget. For example, an application can prevent text from being inserted, as when the user types a password, by using the **XmNmodifyVerifyCallback** or **XmNmodifyVerifyCallbackWcs** callbacks. The application can prevent any text from appearing by setting the *doit* member of the **XmTextVerifyCallbackStruct** (or **XmTextVerifyCallbackStructWcs**) to False. The application can also alter the text that will appear by creating a new text string and setting the *ptr* member of the **XmTextBlockRec** structure (or the *wcsptr* member of the **XmTextBlockRecWcs** structure) to the new string.

Following is an example of an **XmNmodifyVerifyCallback** that substitutes a string of characters for any text a user enters. Because the **XmNmodifyVerifyCallback** procedures are most commonly invoked after the user enters a character, this routine usually substitutes the replacement string for each character the user types. This example could be used with a single-line Text widget as part of a simple password-entry program. In this case, the **XmNmodifyVerifyCallback** procedure would need additional code to save the characters the user types, and the program would need an **XmNactivateCallback** procedure to check whether the saved characters match the password.

```
/* XmNmodifyVerifyCallback procedure that replaces text the user enters
 * with a replacement string passed in as application data. */
```

```
void ModifyVerifyCB(Widget w, XtPointer app_data,
                    XtPointer widget_data)
{
  char *replace_string = (char *) app_data;
  XmTextVerifyCallbackStruct *widget_info =
    (XmTextVerifyCallbackStruct *) widget_data;
  if (widget_info->text->length > 0) {
    widget_info->text->length = strlen(replace_string);
    widget_info->text->ptr = replace_string;
/* The widget will automatically free this ptr.  Therefore,
   your app should make a copy of it with strcpy. */
  }
}
```

Text and TextField differ from most other Motif widgets in that calling some convenience routines and setting some resources causes the widget to invoke callback procedures. In general

- Setting resources or calling convenience routines that change the contents of the text invokes the **XmNmodifyVerifyCallback** and **XmNmodifyVerifyCallbackWcs** callbacks. If these procedures allow the text to be modified, the **XmNvalueChangedCallback** callbacks are invoked.
- Setting resources or calling convenience routines that change the position of the insertion cursor invokes the **XmNmotionVerifyCallback** callbacks.
- Setting resources or calling convenience routines that cause the widget to gain the primary selection invokes the **XmNgainPrimaryCallback** callbacks.
- Setting resources or calling convenience routines that cause the widget to lose the primary selection invokes the **XmNlosePrimaryCallback** callbacks.

If the application needs to distinguish between callbacks invoked as a result of user action and callbacks invoked as a result of application action (such as setting a resource or calling a convenience routine), it needs to set a flag before taking the application action and clear the flag afterward.

10.4 Text Resources and Geometry

In addition to the resources discussed in the previous section, Text has many others, including the following:

- The text itself, **XmNvalue** or **XmNvalueWcs**. For Text and TextField, the text is represented to the application as an array of either *char* elements (for **XmNvalue**) or **wchar_t** elements (for **XmNvalueWcs**). The application can set or get either resource.
- The render table (**XmNrenderTable**) that the widget uses to select a font or font set and other attributes to display the text. Text and TextField use only the font-related rendition resources of the render table.
- Resources representing the insertion cursor position and blink rate, the position of text at the top of the window, and whether the insertion cursor is always visible. A text position (of type **XmTextPosition**) is an integer representing the number of characters from the beginning of the buffer.
- A resource (**XmNmaxLength**) representing the maximum length of the text string that the user can enter.
- A resource (**XmNwordWrap**) that specifies whether lines are broken at word boundaries when the text would be wider than the widget. Breaking a line at a word boundary does not insert a newline into the text.

In addition, Text and TextField have several resources that determine the geometry of the widget:

- Two resources, **XmNmarginHeight** and **XmNmarginWidth**, that determine the margins between the text and the shadow, if present. Text and TextField also use the Primitive resources that determine shadow and highlight appearance.
- Resources that specify the number of rows of text (**XmNrows**) and the number of horizontal character positions (**XmNcolumns**). Single-line Text and TextField always have one row.
- Resources that determine whether or not the widget grows vertically (**XmNresizeHeight**) or horizontally (**XmNresizeWidth**) to accommodate all its text. **XmNresizeHeight** does not apply to single-line Text or TextField.
- Resources that apply only when the widget is inside a ScrolledWindow whose **XmNvisualPolicy** is **XmVARIABLE**. **XmNscrollHorizontal** determines whether or not the widget should have a horizontal ScrollBar and should scroll horizontally

instead of growing when the text expands beyond the width allocated for it. **XmNscrollVertical** determines whether or not the widget should have a vertical ScrollBar and should scroll vertically instead of growing when the text expands beyond the height allocated for it. **XmNscrollLeftSide** and **XmNscrollTopSide** determine which side of the widget receives the corresponding ScrollBar. These resources do not apply to TextField, and **XmNscrollVertical** and **XmNscrollLeftSide** do not apply to single-line Text.

XmNresizeWidth is initialized to False when **XmNscrollHorizontal** is True or **XmNwordWrap** is True. **XmNresizeHeight** is initialized to False when **XmNscrollVertical** is True. Word wrap will not take effect if **XmNscrollHorizontal** is True.

If the user or application initializes or sets a specific height (**XmNheight**) or width (**XmNwidth**), that value is used as the corresponding dimension of the widget. In addition, if a height is specified, **XmNrows** is recalculated based on that height and, if a width is specified, **XmNcolumns** is recalculated based on that width.

If the user or application initializes or sets **XmNrows** but not **XmNheight**, the geometry calculation depends on the value of **XmNresizeHeight**. If **XmNresizeHeight** is False, as it is by default, the height of the widget is simply the height needed to display **XmNrows** of text. The same relations hold for **XmNcolumns**, **XmNwidth**, and **XmNresizeWidth**.

If the user or application does not initialize either **XmNrows** or **XmNheight**, the geometry calculation depends on the value of **XmNresizeHeight**. If **XmNresizeHeight** is True, the height of the widget is the height needed to display all the text. If **XmNresizeHeight** is False, the height of the widget is the height needed to display the default for **XmNrows**, which is one row of text. The same relations hold for **XmNcolumns**, **XmNwidth**, and **XmNresizeWidth**, except that the default number of columns is 20.

If the contents of the text (**XmNvalue** or **XmNvalueWcs**) change, as a result of user editing or an action by the application, the geometry calculation depends on the value of **XmNresizeHeight**. If **XmNresizeHeight** is True, the height of the widget is the height needed to display all the text. If **XmNresizeHeight** is False, the height of the widget does not change. The same relations hold for **XmNvalue**, **XmNvalueWcs**, **XmNresizeWidth**, and the width of the widget.

If the application sets another resource that affects the height needed by the widget, such as **XmNmarginHeight** or **XmNrenderTable**, the geometry calculation depends on the value of **XmNresizeHeight**. If **XmNresizeHeight** is True, the height of the widget is the height needed to display all the text with the new resource values. If **XmNresizeHeight** is False, the height of the widget is the height needed to display **XmNrows** of text by using the new resources. The same relations hold for these resources, **XmNresizeWidth**, **XmNcolumns**, and the width of the widget.

Note: Most of the resources described in this section have other meanings when **XmNlayoutDirection** is **XmTOP_TO_BOTTOM** (for vertical writing). For example, if word-wrapping is turned on, text is broken at the end of a column and is continued at the next column.

10.5 Convenience Routines

Text convenience routines permit the application to perform many functions, including these:

- Insert and replace text.
- Cut, copy, and paste by using the clipboard.
- Get and set the editable state, the insertion cursor position, the maximum length of text, the primary selection and its position, the source, the text string, and the position of the first character displayed. For Text and TextField, all routines that have parameters or return values that are strings have both **char *** and **wchar_t *** versions.
- Convert between a text position and x and y coordinates.
- Display text at a given position and scroll the text.

10.6 ScrolledText

ScrolledText is a Text widget inside a ScrolledWindow. The application can use **XmCreateScrolledText** to create a Text widget inside a ScrolledWindow. This routine creates Text and ScrolledWindow widgets, and forces the following initial values for ScrolledWindow resources:

- **XmNscrollingPolicy** is set to **XmAPPLICATION_DEFINED**.
- **XmNvisualPolicy** is set to **XmVARIABLE**.
- **XmNscrollBarDisplayPolicy** is set to **XmSTATIC**.
- **XmNshadowThickness** is set to 0.

Note that the **XmCreateScrolledText** function returns the widget ID of the created Text widget. An application can then use **XtParent** to get the widget ID of the ScrolledWindow.

10.7 Storing Text in a File

A common requirement of many text editors is the ability to read text from a file, allow the user to edit the text, and then store the text in a file. An application usually obtains pathnames from the user by means of a FileSelectionBox, often invoked as a dialog from a MenuBar File Menu. Following are very simple routines that use ANSI C input/output facilities to read text from a file into a Text widget and save text from a Text widget into a file:

```
void ReadTextFromFile(Widget w, char *filename)
{
  FILE           *file;
  char            buffer[MAXSIZE];
  char           *ptr, *end;
  XmTextPosition  last_pos;
  int             val;
  if (file = fopen(filename, "r")) {
    XmTextSetString(w, "");
    ptr = buffer;
    end = buffer + MAXSIZE - 1;
    while((val = getc(file)) != EOF) {
      if (ptr < end) {
        *ptr++ = (char) val;
      } else {
        *ptr = '\0';
        last_pos = XmTextGetLastPosition(w);
        XmTextReplace(w, last_pos, last_pos, buffer);
        ptr = buffer;
      }
    }
    if (ptr > buffer) {
      *ptr = ’\0’;
      last_pos = XmTextGetLastPosition(w);
      XmTextReplace(w, last_pos, last_pos, buffer);
    }
    (void) fclose(file);
  }
}

void SaveTextToFile(Widget w, char *filename)
{
  FILE     *file;
  char     *text;
  if (file = fopen(filename, "w")) {
    text = XmTextGetString(w);
    (void) fputs(text, file);
    (void) fclose(file);
    XtFree(text);
  }
}
```

10.8 Sharing Text Sources

Each Text widget has a data structure of type **XmTextSource** that functions as the source and sink of text for the widget. The source is the value of the **XmNsource** resource.

Two or more Text widgets can share the same source. In this case, editing of Text in one widget changes the text of the source and therefore the text of all widgets that share that source. For example, an application might use a PanedWindow with multiple text widgets, each functioning as a "window" onto a single text source. Editing changes in one pane are reflected in all Text panes that share the same source.

An application creates a Text source by creating a Text widget. The program uses **XmTextGetSource** for Text or **XtGetValues** for the **XmNsource** resource to obtain that widget’s source. The application then creates another Text widget, supplying the source obtained from the first widget by using **XmTextSetSource**, the initialization argument list, or **XtSetValues** of the **XmNsource** resource.

Setting a Text source destroys the existing source of the widget if no other widgets are sharing that source. To replace a Text source but keep it for later use, the application can create an unmanaged Text widget and set its source to the Text source the program wants to keep.

If the application does not supply a source, Text creates a default string source.

Chapter 11
Internationalization

An internationalized application is one that may be run in many different languages without having to be rewritten or recompiled. This chapter describes how to design applications to use Motif's internationalization capability. It is not a general discussion of internationalization.

11.1 Issues in Internationalized Applications

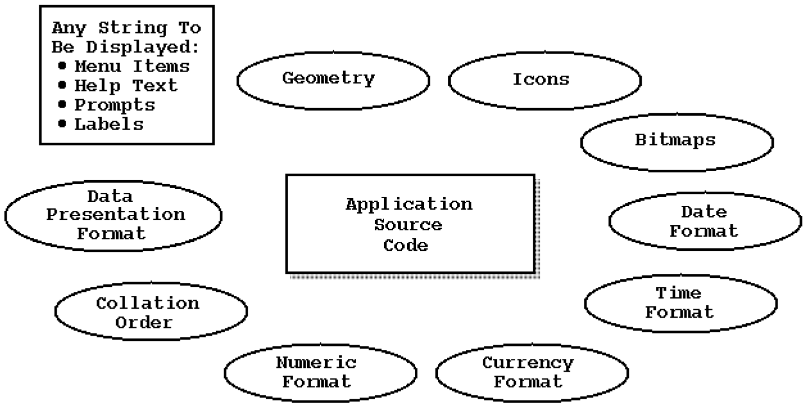
There are several important issues to keep in mind when designing an application so that it takes advantage of Motif's internationalization capabilities.

11.1.1 Internationalization and Localization

An internationalized application contains no code that is dependent on the user's language, the characters needed to represent that language, or any formats (such as date and currency) that the user expects to see and interact with. Motif accomplishes this by storing language and custom dependent information outside the application.

The following figure shows the kinds of information that should be external to an application to simplify internationalization.

Figure 11–1. Information External to the Application



Because the language and culture dependent information is separate from the application source code, the application does not need to be rewritten or recompiled to be marketed in a different countries. Instead, the only requirement is for the external information to be *localized* to accommodate local language and custom.

Localizing the application includes the process of translating certain parts of the external information into the appropriate language and storing the translated information in files that are then accessed by the application. In addition, the application may be told the format to use to display time, date, and the other language or culture dependent formats shown in the previous figure.

Every language consists of a set of characters that, either individually or in combination, represents meaningful words or concepts in the language. The set of characters is called a **character set**. The set of binary values needed to represent all the characters in a language is called a coded character set or, more simply, a **code set**.

Several attempts were started long ago to standardize code sets and continue to this day. The most commonly used code set for English is the American National Standard Code for Information Interchange (ASCII). It originally used a 7-bit encoding scheme plus an eighth bit for error control. Using 7 bits for character representation allows 128 unique binary values. Later versions use the eighth bit as a code bit allowing 255 characters. Both are fine for English and some other alphabetic languages, but neither is suitable for ideographic languages such as Chinese, Japanese, and Korean. Ideographic languages represent a concept or an idea as a single character; consequently, there are thousands of characters in these languages, and two or more bytes are needed to represent the characters.

Other standard code sets have been developed to accommodate other languages. The ISO8859 standard is perhaps the most commonly used of these. Different versions of the ISO8859 standard exist for various areas of the world. The following table shows a typical language and code set relationship for various areas. The code sets shown generally cover many more areas than are indicated, and the Table 11-1 is merely meant as a guide. (As an example, the ISO8859-3 code set covers, in addition to the languages indicated in the table, Afrikaans, Esperanto, German, Italian, Maltese, and Turkish. You can also use it for English.)

Table 11–1. Areas and Typical Code Sets

Area or Language	**Code Set**
English	ASCII, ISO8859-1
Western Europe	ISO8859-1
Eastern Europe	ISO8859-2
Dutch, Catalan, Spanish	ISO8859-3
Northern Europe	ISO8859-4
Russian, Ukrainian, Serbian	ISO8859-5
Hebrew	ISO8859-6
Greek	ISO8859-7, 8, 9
Japan	Shift JIS
Japan	UJIS

See the specifications for the American National Standards Institute (ANSI) C programming language and the *X/Open Portability Guide, Issue 3* (XPG3) for more information on standards involved in internationalization.

11.1.2 Obtaining Input

Special considerations must be made for the user of an application to input characters in the local written language. Virtually all applications require some action on the part of the user, often asking for input in one form or another. For example, an application can ask the user to input information in text form, such as name, home address, and so on. The user must then enter this information by typing it on the keyboard in the normal manner. This is done with relative ease in an English-based application but can become more complex when text in another language is desired.

Motif uses Xlib functions to provide the basic support for obtaining input in the Text widget.

11.1.2.1 The Problems

Many languages are expressed by means of an alphabet made up of characters or letters. The letters are arranged in groups to form meaningful words. A keyboard suitable for the language contains all the letters of the alphabet, plus the standard numerals and punctuation marks. The first problem arises when, as in English, standard spelling and usage requires *two* characters for each letter of the alphabet, while the standard keyboard contains only *one* key for each letter. The solution to this problem is a **Shift** key, which, when pushed in combination with another key, changes the character that key produces.

A somewhat more serious problem arises when the keyboard does *not* have all the alphabet characters. This can happen when a German user is using an English-based keyboard and needs a German character such as "β."

A far more involved example is the case of defining a keyboard to use for the ideographic languages. Because thousands of characters are needed to represent an ideographic language, no reasonable keyboard can be constructed with a single key for each character.

11.1.2.2 The Solution

X and Motif solve these input problems by using an *input method*, which, in its simplest form, is a layer of mapping between the keyboard keys (or combinations of keys) that the user types and the text data that is passed to the application. For example, the Danish user with an English keyboard who needs the letter "" must enter a *combination* of keystrokes (this varies among vendors but could be **Extend char** **O** / as an example) rather than just one keystroke. This is very similar to the act of using the **Shift** key to access uppercase letters.

An ideographic language's input method is often based on the language's phonetics, but there are also input methods based on a common graphics property of certain characters. The graphics method involves defining a key to map to a common graphic symbol that is the basis for multiple characters. The phonetic method is more commonly used. It requires a phonetic (alphabet-based) writing system. The number of phonetic signs or characters is few enough that a unique key is assigned to each phoneme. Characters are entered by pressing the appropriate phonetic keys.

Note that the full definition of an input method actually includes the manner in which text is typed as well as the simple keyboard mapping. In one form of input method, text is simply typed at the spot where it is to appear. In another method, often used in languages where every character requires more than one keystroke, preliminary text appears in some secondary window on the screen until enough has been typed to uniquely specify a new character, which is then passed to the application. In several popular input methods, the user types a phonetic representation of a spoken word and the input method determines which characters are pronounced that way. If only one character meets this criterion, it is displayed. If more than one character meets the criterion, a list of all characters found is displayed and the user chooses the desired one. It is then passed to the application. See Section 11.4.1 for more information on input methods.

11.1.3 Displaying Output

Displaying the output produced by an application intended for international use also requires some consideration. In order to display text, it must have the appropriate content, encoding and fonts. For example, many languages, especially ideographic ones, require more than one font. Bitmaps and pixmaps must be localized as well.

An icon that is an appropriate or meaningful symbol in one country may be totally inappropriate or meaningless in another.

11.1.4 Locales and Localization

A locale is the language environment determined by the application at run time. The *X Portability Guide* defines locale as a means of specifying three characteristics of a language environment that may be needed for localization: language, territory, and code set. Motif supports only one locale per application; that is, an application can set the locale only once, at start-up time.

Motif uses the locale to help find:

- Resource files
- UID files
- Bitmap files
- Fonts used to display text and labels
- Text input method
- Character size

The ANSI C method of setting the locale in an application is to use the function *setlocale*. How *setlocale* obtains a language when the language is not explicitly referenced in the call to *setlocale* is system dependent. For example, on POSIX systems, the environment variable *LANG* is used. The locale name is also used to establish a path to the localized files of information. How this is actually accomplished is explained in Section 11.2.

11.2 Localizing Applications

An internationalized application can be tailored to operate in many areas of the world, each with its own requirements for the language and customs to be used. This section explains some methods for localizing an application.

The following section describes how the user and the application developer (and perhaps the system administrator) establish the language environment of an application. It then discusses two general approaches to localizing applications. Succeeding sections focus on four aspects of localizing information in Motif programs:

- Resource files
- UID files
- Message catalogs
- X bitmap files

Many aspects of localization depend on the particular operating system, Motif implementation, and user environment in which the application runs. The following must all cooperate for correct localization to occur:

- The operating system's locale mechanism, if any
- The Motif implementation
- The application itself
- The user's system administrator
- The user's language environment

11.2.1 Techniques for Localization

Although there are different methods for localizing an application, there are some common considerations:

- The application should not explicitly code any language-dependent information in the application. This includes strings, fonts, and language-dependent pixmaps.
- The application should isolate text, fonts, and pixmaps, and translate them into the languages needed. Usually this information is stored in separate directories by language.

11.2.1.1 Establishing the Language Environment

The term *language environment* refers to the set of localized data that the application needs in order to run correctly in the user-specified locale. A language environment supplies the rules associated with a specific language. In addition, the language environment consists of any externally stored data, such as localized strings or text used by the application. For example, the menu items displayed by an application might be stored in separate files for each language supported by the application. This type of data can be stored in resource files, UID files, or, on XPG3-compliant systems, message catalogs.

A single language environment is established when an application executes. The actual language environment in which an application operates is specified by the application user, often either by setting an environment variable (*LANG* on POSIX-based systems) or by setting the **xnlLanguage** resource. The application then sets the language environment based on the user’s specification. The application can do this either by using *setlocale* in a language procedure established by **XtSetLanguageProc**, or by using a method that does not call *setlocale*. In either case, Xt caches a per-display language string that is used by **XtResolvePathname** to find resource, bitmap, and UIL files.

An application that supplies a language procedure may either provide its own or use an Xt default procedure. In either case, the application establishes the language procedure by calling **XtSetLanguageProc** before calling **XtAppInitialize**. When a language procedure is installed, Xt calls it in the process of constructing the initial resource database. Xt uses the value returned by the language procedure as its per-display language string.

The default language procedure performs the following tasks:

- Sets the locale. On ANSI C-based systems, this is done by using the following code:

  ```
  setlocale(LC_ALL, language);
  ```

  where *language* is the value of **xnlLanguage** or the empty string ("") if **xnlLanguage** is not set. When **xnlLanguage** is not set, the locale is generally derived from an environment variable (*LANG* on POSIX-based systems).

- Calls **XSupportsLocale** to verify that the locale just set is supported. If not, a warning message is issued and the locale is set to "C."

- Calls **XSetLocaleModifiers** specifying the empty string.
- Returns the value of the current locale. On ANSI C-based systems, this is the result of calling the following:

```
setlocale(LC_ALL, NULL);
```

The application can use the default language procedure by making the call to **XtSetLanguageProc** in this manner:

```
XtSetLanguageProc(NULL, NULL, NULL);
.
.
toplevel = XtAppinitialize(...);
```

By default, Xt does not install any language procedure. If the application does not call **XtSetLanguageProc**, Xt uses as its per-display language string the value of the **xnlLanguage** resource if it is set. If **xnlLanguage** is not set, Xt derives the language string from the environment. On POSIX-based systems, this is the value of the *LANG* environment variable.

It is important to note that the per-display language string that results from this process is implementation dependent and that Xt provides no public means of examining the language string once it is established. The following vary by operating system and by Motif implementation:

- The mechanism, if any, used to set the locale
- On ANSI C-based systems, the value returned by *setlocale*
- The possible values of any environment variables used to establish the language environment
- Whether or not **xnlLanguage** is used and, if so, its possible values

Furthermore, by supplying its own language procedure, an application may use any procedure it wants for setting the language string.

11.2.1.2 Using Locales

The locale provides local information to an application based on the user's language, territory, and code set. Both language and territory are needed because some languages are spoken in more than one country and more than one language may be spoken in some countries. (French is an example of the first, and Belgium, Canada, and Switzerland are examples of the second.)

Information in resource, UID, and image files can be localized and stored in separate directories by language. The Xt function **XtResolvePathname** uses the run-time locale to determine the proper directory to use.

On XPG3-compliant systems, an application can use message catalogs to localize text and messages. A message catalog file exists for each language, and each is usually stored in a separate directory by language.

The locale method of localizing compound strings and font lists consists of the following steps:

1. Establish a language procedure before calling **XtAppInitialize**. The language procedure calls *setlocale*.
2. Localize the compound strings and render tables using resource files, message catalogs, or UID files. Normally, do not specify any charset tags other than **XmFONTLIST_DEFAULT_TAG**.
3. Use font sets in resource or UID file font lists.
4. Use **XmStringGenerate** to create compound strings in the program, but only use the rendition tag **_MOTIF_DEFAULT_LOCALE**.

The run-time locale determines which fonts are used to display text. This is accomplished in the following manner:

- Motif calls **XtResolvePathname** to load resource or UID files that specify the names of fonts for font sets. **XtResolvePathname** uses a file search path that may vary depending on the display's language string.
- *XCreateFontSet* uses the locale to determine the fonts to be used from the base font name and the locale charset.

In this method, the application usually does not specify charset tags other than **XmFONTLIST_DEFAULT_TAG**. It is possible to supply explicit rendition tags with locale-dependent text. For example, text might be displayed using large and small fonts or bold and italic fonts. The application can do this with special tags in both the compound string and the render table associated with it. In the render table, match the tag with a font set specification that supplies the desired attribute (point size, for example). When the application creates the font set, the charset comes from the locale. For example, a resource file might specify a render table in the following manner to obtain fonts with a different point size:

```
*fontList:  -*-*-*-R-Normal--*-120-100-100-*-*:,\
            -*-*-*-R-Normal--*-180-100-100-*-*:BIG,\
            -*-*-*-R-Normal--*-80-100-100-*-*:SMALL
```

See Chapter 9 for more information about fonts and controlling font selection.

11.2.1.3 Localization Without Locales

In this method, the locale is not set in the program, and a language procedure is not needed. Instead, the user specifies the language environment by using either **xnlLanguage** or an environment variable such as *LANG*. Resource, UID, and image files are localized and stored in separate directories by language, as they are when the application uses locales. **XtResolvePathname** uses the display’s language string in the same way to determine the proper locations of these files.

Message catalogs are not used in this method. Also, in this case Text and TextField cannot accommodate 16-bit data.

The nonlocale method of localizing compound strings and render tables consists of these steps:

1. Localize compound strings by using UIL files. Localized render tables and font lists can appear in resource files.
2. Specify explicit rendition tags *other* than **_MOTIF_DEFAULT_LOCALE** in both compound strings and render tables.
3. Use font names with explicit charset components in resource or UIL files. Do not use font sets.
4. To create compound strings in the program, use **XmStringGenerate** with the rendition tag set to something other than **_MOTIF_DEFAULT_LOCALE**.

11.2.2 Resources and Localization

The resources used in an application that are subject to internationalization ought to be stored in files external to the application. These resources include

- All labels, particularly those that identify controls. Such labels are defined as type **XmString**, meaning they are compound strings.
- Text strings; that is, strings of text that are not compound strings.
- Render tables.
- Font lists.

11.2.2.1 Initial Resource Database

The information in the external resource files is used when Xt builds the initial resource database. The **XtDisplayInitialize** function loads the resource database by merging in resources from the following sources, in order of precedence (that is, each component takes precedence over the following components):

- The application command line
- A per-host user environment resource file on the local host
- Screen-specific resources for the default screen of the display
- A resource property on the server or user preference resource file on the local host
- An application-specific user resource file on the local host
- An application-specific class resource file on the local host

Localization applies to two components of the initial resource database—the application-specific user and class resources. Localized resources that are controlled by the programmer are in the application class resource file, and localized resources that are controlled by the user are in the user resource file. Note that the user resources take precedence over the application class resources.

11.2.2.2 Resource File Locations

XtDisplayInitialize calls **XtResolvePathname** to load both the user and the class resources.

To load the user's application resource file, **XtDisplayInitialize** uses the value of the **XUSERFILESEARCHPATH** environment variable as the search path. If that variable is not set or if the search path fails to find the file, and if the environment variable **XAPPLRESDIR** is defined, **XtDisplayInitialize** next tries an implementation-dependent search path with a number of entries that include **XAPPLRESDIR** and the user's home directory. If **XAPPLRESDIR** is not set or if that search path fails, **XtDisplayInitialize** tries another implementation-dependent search path with a number of entries that include the user's home directory.

To load the application-specific class resource file, **XtDisplayInitialize** uses the value of the **XFILESEARCHPATH** environment variable as the search path. If that variable is not set or if the search path fails to find the file, **XtDisplayInitialize** tries an implementation-dependent search path.

The search paths for both resource files may contain any substitutions recognized by **XtResolvePathname**. That routine substitutes the display's language string for **%L**. In an implementation-dependent manner, it substitutes the language, territory, and code set components of the language string for **%l**, **%t**, and **%c**, respectively. This mechanism allows Xt to load different resource files for different languages, as specified by the display's language string.

The display's language string is determined by the application's language procedure, if present, or else by the value of the **xnlLanguage** resource or by the environment. The language string associated with any particular language and the search paths used to find the resource files depend on the system vendor, the Motif vendor, the application, and the user's system administrator. Determining the actual directories in which localized resource files reside requires coordination among all these sources.

In general, an application developer prepares a set of localized application class resource files, one for each language the application supports. The developer may also need to supply a language procedure appropriate for one or more of the systems on which the application will run. The application vendor must arrange for the resource files to be installed in the correct directories, depending on the operating system and the Motif implementation on which the application will run.

11.2.2.3 An Example

Following is an example of an application class defaults file for a simple program that creates a MainWindow with a Text widget. Because the render table specification includes a single rendition with a default tag, this resource file would be appropriate for an application that uses locales.

```
*renderTable.fontName:                -*-*-*-R-Normal--*-180-100-100-*-*
*renderTable.fontType:                 XmFONT_IS_FONTSET
*Text1.value:\
Hier ist etwas Text fur das Text Widget.\n\
Gemischter 8-und 16-bit Text.
*version_box.messageString:     Dies ist i18n Demo Version
*version_box.okLabelString:     Schliessen
*version_box.dialogTitle:       I18n Demo Version
*pgm_ver_btn.labelString:       I18n Demo Version
*events_btn.labelString:        Aktionen
*help_btn_menu.labelString:     Hilfe
*help_btn_cascade.labelString:  Hilfe
*help_box.messageString:        Leider ist keine Hilfe hier.
*help_box.okLabelString:        Schliessen
*help_box.dialogTitle:          i18n Demo Hilfe
*stop_btn.labelString:          Enden
```

11.2.3 UIL and Localization

The general models for localizing applications that use UIL are the same as those for applications that do not use UIL. An application developer creates separate UIL files, each containing string and resource values for a particular language. UIL files can also be used in conjuction with localized resource and pixmap files. As with localization of resource files, there are two basic approaches to localizing UIL files: one that uses locales and one that does not.

11.2.3.1 Preparing Localized UID Files

When using locales with UIL, an application developer should follow these rules:

- Do not use a **character_set** declaration for the module.
- When creating compound strings in a UIL file, use double quotes and no character set specification for the text.
- When creating render tables or font lists in a UIL file, use font sets, not fonts. Do not specify character sets for the font sets.
- Before compiling a UIL file via the **uil** command, set up any environment variables (such as *LANG*) or other mechanisms the system vendor recommends to establish the locale that is appropriate for the UIL file to be compiled. Invoke the **uil** command with the **–s** option. This enables the UIL compiler to set the locale and parse double-quoted strings without explicit character sets in the locale's encoding. It also ensures that localized compound strings and font list entries are created with font list element tags of **XmFONTLIST_DEFAULT_TAG**.
- Before using the **Uil** function to compile a UIL file, set the locale that is appropriate for the UIL file to be compiled. In the **Uil_command_type** structure that is the first argument to the **Uil** function, set the **use_setlocale_flag** member to 1. This has the same effect as invoking the **uil** command with the **–s** option.

When localizing UIL files without using locales, an application developer should follow these rules:

- When using single quotes for the text of compound strings, supply a **character_set** declaration for the module.
- When using double quotes for the text of compound strings, supply an explicit character set for each text component.
- When creating font lists in a UIL file, use fonts, not font sets. Specify an explicit character set for each font.
- When compiling a UIL file via the **uil** command, do not invoke the command with the **–s** option. The UIL compiler does not set the locale, and it parses each string by using rules derived from the explicitly specified character set for that string.
- When compiling a UIL file via the **Uil** function, set the **use_setlocale_flag** member of the **Uil_command_type** structure to 0. This has the same effect as invoking the **uil** command without the **–s** option.

The UIL compiler processes a single source file for each invocation of the **uil** command or the **Uil** function. However, UIL has an **include file** directive that is similar to the

C preprocessor's **#include** directive. If the file argument for this directive is not an absolute pathname, the compiler searches for the file in a series of directories. These include the directory of the main UIL source file and any directories specified via the **–I** option to the **uil** command or the **include_dir** member of the **Uil_command_type** structure for the **Uil** function.

One strategy for maintaining localized UIL source files is to place only language-independent information in the main UIL source file and to put all language-dependent information in included files that are in separate directories for each language. Then a developer can compile the UIL files for different languages without editing any UIL files. When using locales, a developer first sets up the environment for the intended locale. Whether using locales or not, the developer then invokes the UIL compiler with the proper include directory for the intended language.

In general, a developer can mix localized UIL files with localized resource files. For example, the developer might specify compound strings in UIL files and render tables in resource files. Note one exception: it is not practical to use resource files to localize compound strings without using locales. This is because no resource file syntax exists for supplying an explicit charset/locale tag for a compound string.

For resource values that the user may override, the developer must use resource files or fallback resources, or must in some way ensure that the user's resource settings can override the developer's settings from the UIL file.

11.2.3.2 MRM and Localized UID Files

Once the developer has generated localized UID files, the vendor and the user's system administrator must arrange for these files to be installed in the appropriate directories for the system where the program is to run. As with resource files, these directories depend on configurations established by the operating system vendor, the Motif vendor, and the system administrator.

MrmOpenHierarchyPerDisplay takes as an argument a list of names of UID files. It calls **XtResolvePathname** to find each file the list. If a filename is an absolute pathname, that pathname is the search path for **XtResolvePathname**. Otherwise, **MrmOpenHierarchyPerDisplay** constructs a search path in the following way:

- If the environment variable **UIDPATH** is set, the value of that variable is the search path.

- If **UIDPATH** is not set, but **XAPPLRESDIR** is set, **MrmOpenHierarchyPerDisplay** uses a default search path with entries that include *$XAPPLRESDIR*, the user's home directory, and vendor-dependent system directories.
- If neither **UIDPATH** nor **XAPPLRESDIR** is set, **MrmOpenHierarchyPerDisplay** uses a default search path with entries that include the user's home directory and vendor-dependent system directories.

These paths may include the substitution field **%U**. In each call to **XtResolvePathname**, **MrmOpenHierarchyPerDisplay** substitutes the current filename from the list of UID files for **%U**. The paths may also include other substitution fields accepted by **XtResolvePathname**. In particular, **XtResolvePathname** substitutes the display's language string for **%L**, and it substitutes the components of the display's language string (in a vendor-dependent way) for **%l**, **%t**, and **%c**. If necessary **MrmOpenHierarchyPerDisplay** searches the path twice, first with **%S** mapped to **.uid** and then with **%S** mapped to NULL. The substitution field **%T** is always mapped to *uid*.

The usual mechanism for employing localized UID files is to use a search path that contains one of the substitutions derived from the display's language string. As with resource files, the vendor and system administrator must ensure that the directories where the localized UID files reside match the display's language string (or the appropriate component of the language string).

11.2.4 Message Catalogs and Localization

On an XPG3-compliant system, an application can use message catalogs to localize text. The format of message catalogs is implementation dependent, and the application must take steps to coordinate the locations of the message catalogs with the locations of resource, UID, and image files. Use of message catalogs requires the following steps:

1. Using an implementation-dependent method, prepare a separate message catalog containing text to be localized for each language.
2. Arrange to have the message catalogs installed in the appropriate directories on the systems on which the application will run.

3. Arrange for the user’s environment to be set up correctly so that the application can read the message catalog appropriate to the language.
4. In the program, use the *catopen* function to open a message catalog and the *catclose* function to close it.
5. Use the *catgets* function to read text from an open message catalog.
6. If necessary, convert the text to the target format (such as a compound string) and, for resources, supply the text in the appropriate widget creation argument list or call to **XtSetValues**.

The *catopen* function takes as an argument the name of the message catalog file. If this is an absolute pathname, *catopen* opens that file. Otherwise, *catopen* uses the value of the **NLSPATH** environment variable as a search path. This path can contain a number of substitution fields. The filename passed to *catopen* is substituted for **%N**. The value of the *LANG* environment variable is substituted for **%L**, and its language, territory, and code set components are substituted for **%l**, **%t**, and **%c**, respectively.

Note that these values may not be the same as the display’s language string or its components. An application and software vendor that use message catalogs must coordinate the locations of message catalogs with those of localized resource, UID, and image files, which usually depend on the display’s language string. One possible strategy is to call *catopen* with an absolute pathname constructed by calling **XtResolvePathname** with the value of **NLSPATH** as the search path argument. **XtResolvePathname** substitutes the display’s language string and its components for **%L**, **%l**, **%t**, and **%c** in *$NLSPATH*. In this way, the application can use a single mechanism, the display’s language string, to distinguish file locations by language. The software vendor must still arrange for the user’s system administrator to install the message catalogs in the correct locations and to ensure that **NLSPATH** is appropriately set in the user’s environment.

11.2.5 Images, Pixmaps, and Localization

A pixmap is a screen image that is stored in memory so that it can be recalled and displayed when needed. Motif has a number of pixmap resources that allow the application to supply pixmaps for backgrounds, borders, shadows, label and button faces, drag icons, and other uses. As with text, some pixmaps may be specific to particular language environments; these pixmaps need to be localized.

Motif maintains caches of pixmaps and images. The function **XmGetPixmapByDepth** searches these caches for a requested pixmap. If the requested pixmap is not in the pixmap cache and a corresponding image is not in the image cache, **XmGetPixmapByDepth** searches for an X bitmap file whose name matches the requested image name. **XmGetPixmapByDepth** calls **XtResolvePathname** to search for the file. If the requested image name is an absolute pathname, that pathname is the search path for **XtResolvePathname**. Otherwise, **XmGetPixmapByDepth** constructs a search path in the following way:

1. If the environment variable **XBMLANGPATH** is set, the value of that variable is the search path.
2. If **XBMLANGPATH** is not set but **XAPPLRESDIR** is set, **XmGetPixmapByDepth** uses a default search path with entries that include *$XAPPLRESDIR*, the user's home directory, and vendor-dependent system directories.
3. If neither **XBMLANGPATH** nor **XAPPLRESDIR** is set, **XmGetPixmapByDepth** uses a default search path with entries that include the user's home directory and vendor-dependent system directories.

These paths may include the substitution field **%B**. In each call to **XtResolvePathname**, **XmGetPixmapByDepth** substitutes the requested image name for **%B**. The paths may also include other substitution fields accepted by **XtResolvePathname**. In particular, **XtResolvePathname** substitutes the display's language string for **%L**, and it substitutes the components of the display's language string (in a vendor-dependent way) for **%l**, **%t**, and **%c**. The substitution field **%T** is always mapped to *bitmaps*, and **%S** is always mapped to NULL.

As with resource and UID files, the usual mechanism for employing localized X bitmap files is to use a search path that contains one of the substitutions derived from the display's language string. As with resource and UID files, the vendor and system administrator must ensure that the directories where the localized X bitmap files reside match the display's language string (or the appropriate component of the language string).

See Chapter 12 for more information on images and pixmaps.

11.2.6 Comparing Approaches to Localization

The locale approach allows an application to use existing internationalization routines. On the other hand, the application is limited in portability to systems that support the same internationalization standards (XPG3, POSIX, or ANSI). This approach is also only applicable to applications using a single language.

The nonlocale approach only addresses the aspect of isolating information from the application and ensuring that it uses the proper localized version of this information. The disadvantage is that there is more work for the programmer and there may be nonstandard functionality. The advantages are that there is guaranteed portability across all platforms that support Motif, and that it allows handling of multiple character sets for specialized applications that require this functionality.

11.3 Layout Direction

Layout direction refers to the direction that is used to display visual elements such as widget children, widget components, and text. In general, this direction matches the direction that people use when reading or writing in a particular language. Languages such as English, French, German, and Swedish are read and written from left to right. Therefore, when users working in those languages enter characters from a computer keyboard, each new character is displayed to the right of the preceding one. These same users would also expect the layout of other visual elements to be displayed from left to right. For example, in a menu bar, the cascade buttons would be laid out from left to right so that a simple menu bar would position the "File" cascade button in the upper left corner, and the "Help" cascade button would appear in the upper right corner of the menu bar.

Languages such as Arabic and Hebrew are read and written from right to left. To display text correctly in these languages on the screen, each successive character that a user enters must appear to the left of the preceding character. Using the example above for layout of other visual elements, these users would expect a menu bar to lay out cascade buttons from right to left. The result would typically position the "File" cascade button in the upper right corner and the "Help" cascade button in the upper left corner of the menu bar.

There are several reasons why it is helpful for programmers to be able to specify the layout direction in applications:

- Application programmers want to use the same application in a variety of locales, including those with right-to-left oriented languages. They need to be able to specify that, when using a locale whose language is either Hebrew or Arabic, menus, labels, and messages, for example, should be displayed from right to left and be right justified.
- When applications require entering numeric values, even if the application is restricted to an audience with a right-to-left locale, users need to be able to enter numbers in certain text widgets so that they display from left to right while still entering text that displays from right to left in other text widgets.

You can use the **XmNlayoutDirection** resource to set the default layout direction for your entire application. This resource specifies the default layout direction for all widgets that are affected by it. In turn, the **XmNlayoutDirection** resource sets a default rendering direction for any compound string (**XmString**) that does not have a component specifying the direction for that string. A widget that needs to render such a string should use this resource value to substitute for the missing direction indicator.

The following two examples clarify the use of the **XmNlayoutDirection** resource.

Suppose your application contains only unidirectional compound strings; that is, every **XmString** in the application is either left-to-right or right-to-left. To set the layout direction, all you need to do is set the appropriate value for the **XmNlayoutDirection** resource. You do not need to create compound strings with specific direction components. When the application renders an **XmString**, it should look to see if the string was created with an explicit direction (**XmStringDirection**). If there is no direction component, the application should check the value of the **XmNlayoutDirection** resource for the current widget and use that value as the default rendering direction for the **XmString**.

Another more complex example involves an application that runs in a locale with right-to-left languages but includes text widgets for entering numbers, which need to be displayed from left to right. In this example, the application needs to set the **XmNlayoutDirection** resource to right-to-left for the entire application and then explicitly reset the **XmNlayoutDirection** resource to left-to-right only for those widgets that display numerical values. You still do not need to set any direction components for the compound strings themselves.

In Motif applications, you can set the layout direction by using the **XmNlayoutDirection** resource from the **VendorShell** or **MenuShell**. Manager and Primitive widgets (as well as Gadgets) also have an **XmNlayoutDirection** resource. The default value is inherited from the closest ancestor that has the same resource.

The layout direction resource affects some or all of the subwidgets of the following three widget classes:

- **XmGadget**
- **XmPrimitive**
- **XmManager**

The specific effects of the **XmNlayoutDirection** resource vary with the widget in question. The following three sections outline these effects for the widgets in the three listed classes.

11.3.1 XmLabelGadget and Related Widgets

The following list describes display situations with elements of the **XmLabelGadget** class that are dependent on the layout direction.

- **XmCascadeButtonGadget**
  - — Positioning of cascade graphics
  - — Positioning of menu popup
- **XmLabelGadget**
  - — Meaning of **XmNalignment** resource values
  - — Default **XmNstringDirection**
  - — Positioning of accelerator text
- **XmPushButtonGadget**, *XmTabButtonGadget*
  - — Positioning of accelerator text
  - — Meaning of **XmNalignment** resource values
- **XmToggleButtonGadget**
  - — Positioning of accelerator text

— Meaning of **XmNalignment** resource values

— Positioning of toggle graphic

11.3.2 XmPrimitive and Related Widgets

The widgets **XmCascadeButton**, **XmLabel**, **XmPushButton**, *XmTabButton*, and **XmToggleButton** use the **XmNlayoutDirection** resource in the same manner as their corresponding gadget outlined in the previous section.

Note that the arrow keys osfRight and osfLeft refer to absolute directions within a row, and do not change their meaning when a widget's layout direction changes. Similarly, osfUp and osfDown refer to absolute directions within a column, and do not change their meaning when a widget's layout direction changes. However, a widget's layout direction does affect the interpretation of the arrow keys when the pointer is at the end of a row or column. In other words, a widget's layout direction will affect the way in which the widget "wraps" when it reaches the end of a row or column.

The following list describes display situations with elements of the **XmPrimitive** class that are dependent on the layout direction.

- **XmDrawnButton**

  — Resizing edge default

- **XmList**

  — **XmNstringDirection** default value

  — Meaning of alignment resources

  — MARQUEE behavior during selection

- **XmScrollBar**

  — Default value of **XmNprocessingDirection**

- **XmText**

  — Text writing direction only for *XmTOP_TO_BOTTOM* is supported.

11.3.3 XmManager and Related Widgets

The following list describes display situations with elements of the **XmManager** class that are dependent on the layout direction.

- **XmBulletinBoard**
  - The **XmNdialogTitle** direction is set from **XmNstringDirection**
- **XmComboBox**
  - Layout of arrows with regard to text
  - Direction in which List can be displayed
  - Resize direction for List
- **XmCommand**
  - Positioning of prompt string
- **XmContainer**
  - Default layout of contained objects
  - Positioning of label with regard to pixmap for icons
  - MARQUEE selection behavior
  - Position of header
- **XmDrawingArea**
  - Resizing edge default
- **XmFileSelectionBox**
  - Placement of scrollbars
- **XmForm**
  - Meaning of left and right in resource values
  - Default side for attachments
- **XmFrame**
  - Layout of children
  - Meaning of **XmNchild*Alignment** resources
- **XmMainWindow**

  - Layout of children
- **MenuBar**
  - Layout of children
- **XmMessageBox**
  - Layout of buttons
  - Positioning of pixmap
  - Default alignment of labels
- **XmNoteBook**
  - Default layout of children and book visuals
  - Meaning of "left" and "right" for arrows
- **OptionMenu**
  - Alignment of label
  - Positioning of bar graphic
  - Positioning of pulldown menu
- **XmPanedWindow**
  - Positioning of sash
  - Meaning of **XmNsashIndent**
- **PopupMenu**
  - Location of hotspot
  - Positioning of menu
- **PulldownMenu**
  - Alignment of edge of menu with regard to parent cascade button
- **XmRowColumn**
  - Layout of children, including menu bar cascades
  - Meaning of **XmNentryAlignment**
  - Meaning of **XmNentryVerticalAlignment**
  - Meaning of **XmNisAligned**

- **XmScale**
  - Positioning of text string
  - Positioning of value if shown
  - Default value of **XmNprocessingDirection**
  - Positioning of tick marks.
- **XmScrolledWindow**
  - Default positioning of scrollbar
- **XmSelectionBox**
  - Layout of buttons
  - Alignment of labels
  - Interpretation of **XmNchildPlacement** resource
- **XmSpinBox**
  - Default layout of text with regard to arrows
  - Meaning of left and right arrows

11.4 Internationalization and Text Input

An application subject to internationalization presents some unique problems when it deals with text input. The application must be able to correctly interpret and process text input in any language. This section explains how an application accomplishes this.

11.4.1 Input Method

Although there are many different keyboards in use, sometimes certain characters in an alphabetic language are not directly available on any keyboard. In this case, the user must type a combination of keys to input the desired character. In English, for example, the capital letters are produced by pressing the <Shift> key in combination with a letter key. Other alphabetic languages with larger alphabets than English may use slightly more complex combinations of keystrokes to describe their entire alphabet.

This problem, however, is compounded many times in the case of ideographic languages, which may require thousands of different characters for basic text. This far exceeds the capability of any keyboard and makes it impossible to have a keyboard with all of the language's symbols. An *input method* can be used to overcome this difficulty.

An input method is simply the mechanism that is used to map between the keys pressed by a user and the resulting characters that are input to an application. A common feature of many input methods is that the application user may press combinations of keys to create a single character. Creating characters from keystrokes is called **pre-editing**.

Input methods may require several areas to display the actual keystrokes.

- The **status area** is an output-only window that identifies the style of input (phonetic, numeric, stroke and radical, and so on) and the current status of an input method interaction.
- The **pre-edit area** displays the intermediate text for languages that are composed before the application acts on the data. There are several possible locations for the pre-edit area:

Over-the-spot	Displays the data in an input method window that is placed over the point of insertion.
Off-the-spot	Displays the pre-edit window inside the application window (usually at the bottom) but not at the point of insertion.
On-the-Spot	Displays the pre-edit string in the text widget window.
Root-window	Uses a pre-edit window that is a child of the root window.

  A VendorShell resource, **XmNpreeditType** determines which style is used for a Text or TextField input method. The syntax, possible values, and default value of this resource are implementation dependent.
- The **auxiliary area** is used for popup menus and customizing dialogs that some input methods use.

Input methods are supplied by vendors and are implementation dependent. The VendorShell resource **XmNinputMethod** is an implementation-dependent string that specifies the input method portion of the locale modifiers. If a value is supplied for this resource, Motif uses it to set the locale modifiers before opening an input method for Text or TextField.

Figure 11-2 shows one possible program window with a Text widget using over-the-spot interaction for Japanese text input. The status area indicates that phonetic input is in use and insert mode is enabled. The pre-edit area shows that the letter "H" has been entered. Since there is no Hiragana phonetic equivalent, the "H" appears in the pre-edit window.

Figure 11–2. Text Widget Pre-Edit and Status Areas Using Over-the-Spot

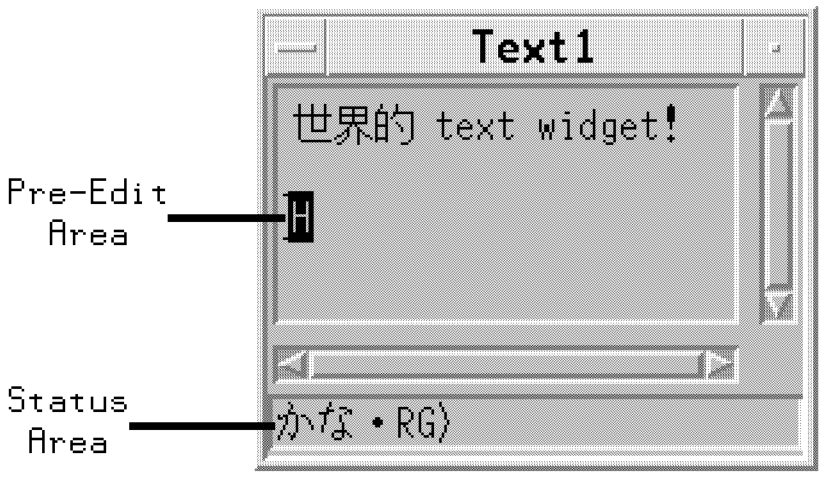
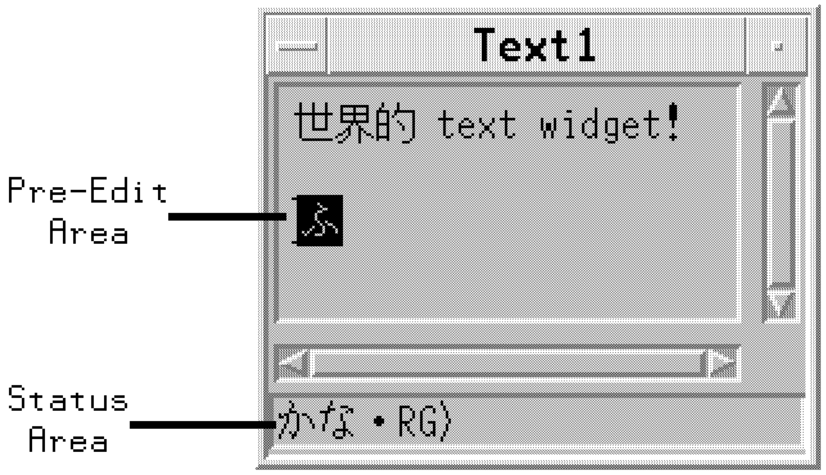


Figure 11-3 shows the same window after a "u" has been entered following the "H" shown in Figure 11-2.

Figure 11–3. Text Widget Pre-Edit Area After Next Character Entry



Here the pre-edit area is displaying the phonetic equivalent of the English letters "hu" in Hiragana.

When on-the-spot input style is used, a pre-edit string is displayed in the text widget window. This preedit string is considered part of the text widget value, and its integrity is ensured by the verify callbacks of the text widget. If the verify callbacks of the text widget do not accept any part of the preedit buffer, the preedit string is committed. The following actions also cause the text widget to commit the preedit string before performing the specified action:

Table 11–2. Highlight Modes

Commit Actions
cut
paste
selection
cursor movement
commit key

Note: Note: Cursor movement may be interpreted by the input method as cursor movement within the preedit buffer. If this is the case, the preedit buffer may not be committed. This behavior is completely dependent upon the implementation of the input method.

In the case of a shared XIC, the widgets that share the XIC shall retain the preedit buffer, if any, and preedit state when focus is switched between the widgets that share the XIC.

When the preedit buffer is active, it may be highlighted. This highlight value can be set by the input method server. The following mapping table relates Text widget highlighting modes to the input method highlighting feedbacks:

Table 11–3. Highlight Modes

XIMFeedback	*XmHighlightMode*
XIMReverse	XmHIGHLIGHT_SELECTED
XIMUnderline	XmHIGHLIGHT_SECONDARY_SELECTED
XIMHighlight	XmHIGHLIGHT_NORMAL
XIMPrimary	XmHIGHLIGHT_SELECTED
XIMSecondary	XmHIGHLIGHT_SECONDARY_SELECTED
XIMTertiary	XmHIGHLIGHT_SELECTED

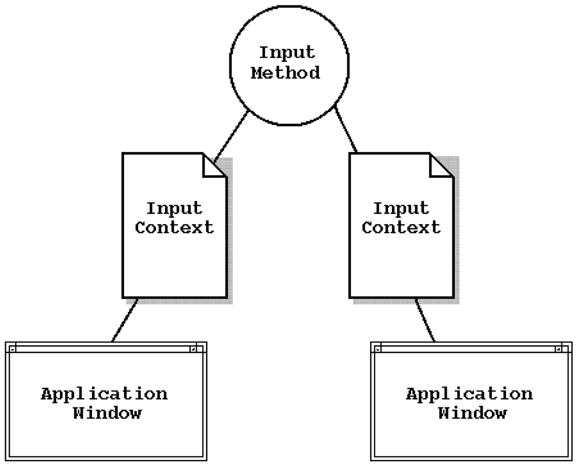
In normal insertion mode, original data in the text widget is shifted out to give way for the preedit string at the insertion point.

In overstrike mode, the preedit string will replace the same number of characters, if available before the end of the text widget value, at the insertion point during the preedit process.

11.4.2 Input Context

An **input context** is the mechanism used to provide the state information needed to manage the information flow between the application and the input method. It is a combination of an input method, a locale specifying the encoding of character strings to be returned, an application window, and internal state information. The relationship between the input method and its input contexts is roughly comparable to that between the display and its windows. The following figure shows the relationships involved. The input method is determined by the **XmNinputMethod** resource of the nearest ancestor **VendorShell**, or by the locale specified by the application user.

Figure 11–4. Input Method and Input Contexts



11.4.3 Input Method Functions

The widgets in the Motif widget set are equipped to choose the appropriate input method based on the specified locale. This process is transparent to the user, as well as to the programmer of most applications. Most programmers will not need any more than to know what to expect under different values of the **XmNpreeditType** resource. Occasionally, however, a programmer will require direct access to the input method. For example, in order to write a widget that accepts typed input, the input method must be explicitly nominated. Motif makes available a high-level interface to the basic Xlib input method functions.

Note: The following XmIm functions are made available only for the convenience of the authors of new widgets. Except for the **XmDrawingArea** widget, these routines should *not* be used with existing widgets.

- **XmImRegister**
- **XmImUnregister**
- **XmImGetXIM**
- **XmImCloseXIM**

A widget must be registered with an input method and context before an application can use **XmImMbLookupString** to retrieve multibyte text from the input method. The **XmImRegister** function is the usual method for selecting and allocating an input method for a widget. It starts from the given widget, and searches up the widget hierarchy until it finds a shell that has an **XmNinputMethod** resource. This resource is only defined for the **VendorShell** widget. If there is an open input method for this display, the current widget will be attached to it. If not, a new input method, specified by the **XmNinputMethod** resource, will be opened and attached to the **XmDisplay**. If the **XmNinputMethod** is NULL, or unspecified, the input method corresponding to the current locale will be chosen and opened. Motif will only support a single input method per application.

XmImRegister and **XmImSetValues** (or **XmImSetFocusValues**) do not return the *XIM* and *XIC* data structures to the calling application, but maintain the connections between the widget and its input method and input context in an internal registry until **XmImUnregister** is called. This will then permit the application to use **XmImMbLookupString** and *XmImSetFocus* functions, which take the widget as an argument.

The two **XmImRegister** and **XmImUnregister** functions, as well as and **XmImSetValues** (or **XmImSetFocusValues**), are the only input manager functions needed to establish input methods and contexts for all but the most unusual applications. These are the only calls needed to implement the Motif Text and TextField widgets.

Like the **XmImRegister** function, the **XmImGetXIM** function searches from the given widget, up the widget hierarchy, until it finds a shell that has an **XmNinputMethod** resource. If there is an open input method for the display, it will simply be returned. Otherwise, a new input method is opened, as specified by the **XmNinputMethod** resource found, and attached to the **XmDisplay**. If the **XmNinputMethod** is NULL, or unspecified, the input method corresponding to the current locale will be chosen and opened. Motif will only support a single input method per application. The **XmImRegister**, **XmImUnregister**, and **XmImSetValues** (or **XmImSetFocusValues**) functions will suffice for most applications. Use **XmImGetXIM** and **XmImCloseXIM** when the widget needs a different input policy than its parent, or needs access to the input method data structure.

Use **XmImCloseXIM** to release the input method associated with the input widget's **XmDisplay**. The function also frees all the input contexts associated with the input method, and their associated memory, and unregisters all widgets associated with the freed input contexts. To close only the input context for one widget, use **XmImUnregister** or **XmImFreeXIC**.

The following functions handle the creation and deletion of an input context.

- **XmImGetXIC**
- **XmImSetXIC**
- **XmImFreeXIC**

Use the **XmImGetXIC** function to create and register a new input context for a widget. A new input context is not required if the current input policy is **XmPER_SHELL** and an open input policy already exists. In this case, **XmImGetXIC** registers the input widget with the existing input context, and returns the shared *XIC*. The **XmImSetValues** function is equivalent to calling the **XmImGetXIC** function with a NULL argument list and an input policy of **XmINHERIT_POLICY**. Unlike **XmImRegister**, **XmImGetXIC** returns the created *XIC* data structure. Use **XmImGetXIC** to override the default input policy, or to specify new values for input context parameters.

The **XmImSetXIC** function allows the application to provide other input contexts to use with the widget and to access the current registered input context data structure. The input *XIC* is registered as the current *XIC*, and any *XIC* previously registered with the input widget is removed. The function returns the newly registered *XIC*. If the input *XIC* argument is NULL, **XmImSetXIC** simply returns the currently registered *XIC*.

The **XmImFreeXIC** function unregisters all widgets currently registered with a particular input context. It then removes the *XIC* itself, and frees the memory allocated to it.

These functions may be used to modify an existing input context:

- **XmImSetValues**

- **XmImSetFocusValues**
- **XmImUnsetFocus**

Use **XmImSetValues** to create an input context, or to modify an existing one. If the current state of the input context does not allow modification, the *XIC* will be unregistered and deleted, and a new one will be created and registered with the input widget. Also, if there is not yet an *XIC* for the widget, one will be created using the given parameters.

The **XmImSetFocusValues** function is nearly identical to the **XmImSetValues** function, except that, after the input context values have been reset, the input focus window for the *XIC* is set to the window of the input widget.

Use the **XmImUnsetFocus** function to unset the focus window for any *XIC* registered to the input widget. If the focus window is not set, this function has no effect.

The following function is used to receive data from the input context:

- **XmImMbLookupString**

The **XmImMbLookupString** function is the heart of the input method. On input, it accepts a widget and a KeyEvent. Using the input context registered with the given widget, the function returns a buffer of multibyte text, a keysym computed for the event, and the length of the string in the output buffer.

Note that, if the key event did not complete some necessary pre-edit sequence, the length of the returned string may be zero. For example, in Figure 11-2, the key event that produced the "H" shown, since it does not specify a unique Hiragana character, would produce a zero-length return from **XmImMbLookupString**. The subsequent key event, shown in Figure 11-3, produces a "u." **XmImMbLookupString** can now map the English letters "Hu" to a unique Hiragana character, so that character is returned to the application, presumably to be drawn in some appropriate place.

11.4.4 Input and the Motif Text Widgets

The Motif Text and TextField widgets, when editable, provide a transparent connection to the locale-specific input method for text input. The application programmer specifies an appropriate font set in the text widget’s **XmNfontList** resource and creates the widget as a descendant of **VendorShell**. **VendorShell** provides geometry management of the status and pre-edit areas. It also supplies a visual separator between the status area window and the application's top level window.

Setting the **VendorShell** resource **XmNpreeditType** dictates the location of the input method window. With an off-the-spot input method, the pre-edit and status area windows appear at the bottom of the application window.

11.4.5 Text Input Using a DrawingArea

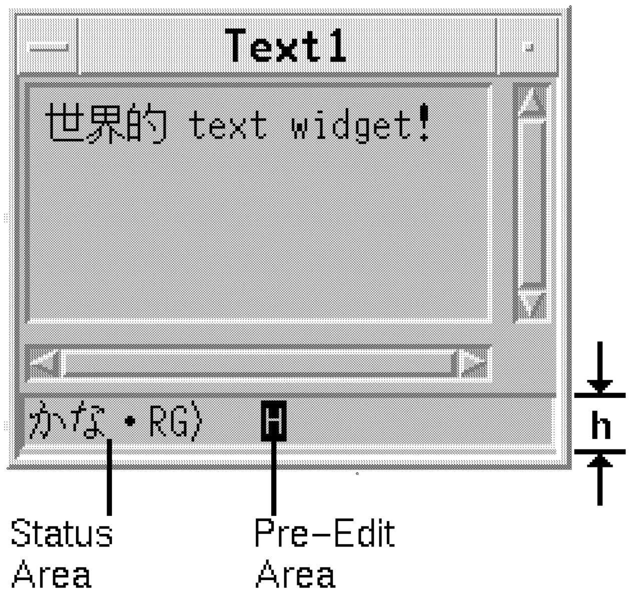
An application that needs special text processing may use a DrawingArea for text input and output. For internationalized text input with any widget other than the various Motif text widgets, the application must use the *XmIm* input method facilities. These allow the application to open an input method and input context and to obtain input from the input method. When using these facilities, an application may also need to handle input method geometry management, focus management, event filtering, and other issues.

11.4.6 Geometry Management of Pre-Edit and Status Areas

When an off-the-spot input method is used with the Text or TextField widget, the pre-edit and status areas are below the client's main window but inside the **VendorShell**. **VendorShell** accomplishes the necessary geometry management. If the application uses either **XtGetValues** or **XtSetValues** to get or set the height (**XmNheight**) of **VendorShell**, the height includes the height of the input method area.

The following figure shows a Text widget using an off-the-spot input method. The distance "h" is the additional height that the input manager needs to display the status and pre-edit areas. Note that, in off-the-spot, the pre-edit area is at the bottom of the interaction.

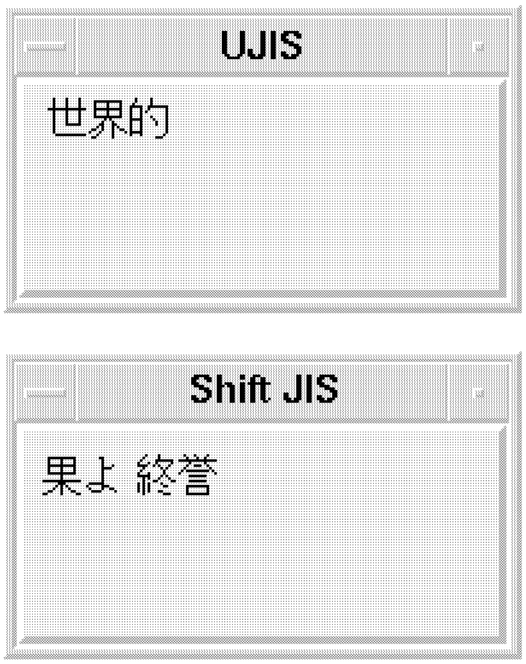
Figure 11–5. Text Widget Pre-Edit and Status Areas Using Off-the-Spot



11.5 Compound Strings and Compound Text

Compound text is the standard format for exchanging textual data between X window system applications. This is necessary when the user moves text displayed in one code set to another window with text in a different code set. For example, the following figure shows two windows: one titled "UJIS" and the other titled "Shift JIS."

Figure 11–6.   Reason for Compound Text



Both windows represent a Motif Text widget, one with some Japanese UJIS characters displayed, and the other with some Shift JIS characters. If the user wants to cut text from one window and paste it in the other window, compound text is used to pass data between the two. The Motif Text widget does this automatically.

If one of the widgets in the previous figure is a Label widget instead of a Text widget, a different situation exists. This is because the Label widget has its text data in compound string format, while the Text widget data is a simple character string. In order to pass text data between a Text or TextField widget and any other widget, the application needs to convert the compound string to compound text.

Motif has two functions, **XmCvtXmStringToCT** and **XmCvtCTToXmString**, for converting between compound strings and compound text.

XmCvtXmStringToCT converts a compound string to compound text. The converter uses the font list tag associated with a given compound string text component to select a compound text format for that component. A registry defines a mapping between font list tags and compound text encoding formats. The converter uses the following algorithm for each compound string text component:

1. If the associated tag is mapped to **XmFONTLIST_DEFAULT_TAG** in the registry, the converter passes the text of the compound string component to **XmbTextListToTextProperty** with an encoding style of **XCompoundTextStyle** and uses the resulting compound text for that component.
2. If the associated tag is mapped to an MIT registered charset in the registry, the converter creates the compound text for that component by using the charset (from the registry) and the text of the compound string component as defined in the X Consortium Standard *Compound Text Encoding*.
3. If the associated tag is mapped to a charset in the registry that is neither **XmFONTLIST_DEFAULT_TAG** nor an MIT registered charset, the converter creates the compound text for that component by using the charset (from the registry) and the text of the compound string component as an "extended segment" with a variable number of octets per character.
4. If the associated tag is not mapped in the registry, the result is implementation dependent.

An application can use **XmRegisterSegmentEncoding** to map a font list element tag to a compound text encoding format. For example, the application may be using a font list element tag of "BOLD" to identify a compound text component consisting of localized text to be displayed in a bold font. To ensure that the component is treated as localized text when converted to compound text, the tag "BOLD" should be mapped to **XmFONTLIST_DEFAULT_TAG** as follows:

```
char *old_encoding = XmRegisterSegmentEncoding("BOLD",
                             XmFONTLIST_DEFAULT_TAG);
XtFree(old_encoding);
```

The following functions may be used both to convert text in a Motif compound string into the compound text format, and to create Motif compound strings from compound text.

- **XmCvtCTToXmString**
- **XmCvtXmStringToCT**

XmCvtCTToXmString converts compound text to a compound string. This function is implementation dependent. There is also the reverse function, called **XmCvtXmStringToCT**.

The following example uses the compound text format to change a window title to a string in the language of the locale. Of course, the language environment must be properly set, and the strings used must also translate properly into the target language using the locale’s code set. This example uses the EUC coding of Japanese, and produces a **XmNtitle** reading "Information" and a **XmNdialogTitle** reading "Unsaved Changes." The following example consists of two pieces: a fragment from an X resource file and a piece of C code. First the X resource file fragment:

```
mwm*renderTable.fontName:       -*-fixed-medium-r-normal--*-150-*
mwm*renderTable.fontType:       XmFONT_IS_FONTSET
```

and now the C code:

```
Widget     toplevel;
Widget     dialog;
Arg        ArgList[10];
int        n;
Atom       atom;
XmString   compound_string1, compound_string2;
char      *compound_text;

    /* Set compound_string1 to "Information" (EUC coding) */
    compound_string1 = XmStringCreateLocalized("%$%s%U%)%a!<%7%g%s");
    /* Set compound_string2 to "Unsaved Changes" (EUC coding) */
    compound_string2 = XmStringCreateLocalized("JQ99$rJ]B8$7$F$$$^$;$s!#");

    atom = XmInternAtom(XtDisplay(toplevel), "COMPOUND_TEXT", False);

    compound_text = XmCvtXmStringToCT(compound_string1);

    n = 0;
    XtSetArg(ArgList[n], XmNtitle, compound_text); n++;
    XtSetArg(ArgList[n], XmNtitleEncoding, atom); n++;
    XtSetArg(ArgList[n], XmNiconName, compound_text); n++;
    XtSetArg(ArgList[n], XmNiconNameEncoding, atom); n++;
    XtSetValues(toplevel, ArgList, n);
```

```
n = 0;
XtSetArg(ArgList[n], XmNdialogTitle, compound_string2); n++;
XtSetValues(dialog, ArgList, n);
```

See Chapter 16 for more information on transferring data between applications. The compound text format is described in the X Consortium Standard *Compound Text Encoding*.

Chapter 12

Color and Pixmaps

Motif uses colors and pixmaps for several general purposes:

- To fill window backgrounds and borders
- To draw text and graphics in window foregrounds
- To generate shadows with a three-dimensional appearance
- To highlight the widget that has keyboard focus
- To indicate that a button is armed or selected

Motif uses other pixmaps for specific purposes:

- As the application's icon for use by the window manager
- For drag icons and drop site animation
- As a CascadeButton symbol indicating that a menu is attached to the CascadeButton
- As a MessageBox symbol indicating the type of message displayed
- As the face of a button when the button is insensitive

All of these colors and pixmaps are represented as resources. The user or application can set the resource values using resource files, and the application can set them using **XtSetValues**.

Motif also uses a number of pixmaps that are not represented as resources. The user and application cannot change these. Among these fixed pixmaps are the pixmaps used to draw arrows in ScrollBars.

12.1 Default Colors and Pixmaps

The following resources determine the colors or pixmaps generally used in Motif:

Borders	Core resources **XmNborderColor** and **XmNborderPixmap**
Backgrounds	Core resources **XmNbackground** and **XmNbackgroundPixmap**
Foregrounds	Gadget, Primitive, and Manager resource **XmNforeground**; Label and LabelGadget resources **XmNlabelPixmap** and **XmNlabelInsensitivePixmap**
Shadows	Gadget, Primitive, and Manager resources **XmNbottomShadowColor**, **XmNbottomShadowPixmap**, **XmNtopShadowColor**, and **XmNtop-ShadowPixmap**
Focus highlights	Gadget, Primitive, and Manager resources **XmNhighlightColor** and **XmNhighlightPixmap**
Arming and selection	PushButton and PushButtonGadget resources **XmNarmColor** and **XmNarmPixmap**; ToggleButton and ToggleButtonGadget resources **XmNarmColor**, **XmNindeterminateInsensitivePixmap**, **XmNindeterminatePixmap**, **XmNselectInsensitivePixmap**, **XmNselectPixmap**, and **XmNunselectColor**; ScrollBar resource **XmNtroughColor**; Display resource **XmNenableToggleColor**

The following sections describe these groups of resources and their defaults.

12.1.1 Borders

The border color or border pixmap is used to fill the border of a widget if **XmNborderWidth** is greater than 0. Note that the border is outside the widget; that is, it is not within the area determined by the widget's **XmNheight** and **XmNwidth**. If the user or application supplies a value for **XmNborderPixmap**, that pixmap is used to fill the border; otherwise, **XmNborderColor** is used.

If the application resource **reverseVideo** is False or unspecified, the default for **XmNborderColor** is the black pixel of the widget's screen. If **reverseVideo** is True, the default for **XmNborderColor** is the white pixel of the widget's screen.

12.1.2 Backgrounds

The background color or background pixmap is used to fill a widget before anything else is displayed in it. If the user or application supplies a value for **XmNbackgroundPixmap**, that pixmap is used to fill the background; otherwise, the **XmNbackground** color is used. A gadget inherits the background color and background pixmap of its parent, if those resources are not set.

The default for **XmNbackground** is implementation dependent.

12.1.3 Foregrounds

The foreground color is used to display text and most graphics in a widget. Most widgets use the **XmNforeground** color for this purpose. Label, LabelGadget, and their subclasses (except for ToggleButton and ToggleButtonGadget which are special cases described later in this chapter) have pixmap resources that are used for the face of the label or button when **XmNlabelType** is set to **XmPIXMAP**. In this case, **XmNlabelPixmap** is used for the face when the widget is sensitive, and **XmNlabelInsensitivePixmap** is used when the widget is insensitive. A gadget inherits the foreground color of its parent (if none is set).

The default for **XmNforeground** is a color that contrasts with the background color, based on the XmScreen resource **XmNforegroundThreshold**. The value of this resource is an integer between 0 and 100, inclusive, that specifies a level of perceived brightness for a color. If the perceived brightness of the background color is equal to or below this level, Motif treats the background as "dark" when computing the default foreground color. If the perceived brightness of the background color is above this level, Motif treats the background as "light" when computing the default foreground color. When the background is "dark," the default foreground is white; when the background is "light," the default foreground is black.

12.1.4 Shadows

The top shadow color or top shadow pixmap is used to draw the top and left sides of the three-dimensional shadow at the edge of some widgets. If the user or application supplies a value for **XmNtopShadowPixmap**, that pixmap is used for the top and left sides; otherwise, **XmNtopShadowColor** is used.

The bottom shadow color or bottom shadow pixmap is used to draw the bottom and right sides of the three-dimensional shadow. If the user or application supplies a value for **XmNbottomShadowPixmap**, that pixmap is used for the bottom and right sides; otherwise, **XmNbottomShadowColor** is used.

A gadget inherits the top and bottom shadow colors and pixmaps of its parent, if those resources are not set.

In computing the defaults for **XmNtopShadowColor** and **XmNbottomShadowColor**, Motif uses the XmScreen resources **XmNdarkThreshold** and **XmNlightThreshold**. The value of each resource is an integer between 0 and 100, inclusive, that specifies a level of perceived brightness for a color. If the perceived brightness of the background color is equal to or below the **XmNdarkThreshold**, Motif treats the background as "dark" when computing the default shadow colors. If the perceived brightness of the background color is above the **XmNlightThreshold**, Motif treats the background as "light" when computing the default shadow colors. Otherwise, Motif treats the background as "medium" when computing the defaults.

Motif computes the defaults in the following way, depending on the perceived brightness of the background:

Dark background	The top and bottom shadow colors are interpolated toward white from the background, with the top shadow color shifted more toward white than the bottom shadow color.

Light background	The top and bottom shadow colors are interpolated toward black from the background, with the bottom shadow color shifted more toward black than the top shadow color.
Medium background	The top shadow color is interpolated toward white from the background, and the bottom shadow color is interpolated toward black from the background.

12.1.5 Focus Highlights

The highlight color or highlight pixmap is used to draw the highlighting rectangle around widgets that have keyboard focus. If the user or application supplies a value for **XmNhighlightPixmap**, that pixmap is used for the highlight; otherwise, **XmNhighlightColor** is used. The highlight color is also used to draw the location cursor around List items that have keyboard focus. A gadget inherits the highlight color and highlight pixmap of its parent if those resources are not set.

The default highlight color is the same as the default foreground color.

12.1.6 Arming and Selection

In PushButtons and PushButtonGadgets outside menus, **XmNarmColor** is used as the button background when the **XmNfillOnArm** resource is True and the user arms the button. In PushButtons and PushButtonGadgets outside menus, **XmNarmPixmap** is used as the button face (the label area) when **XmNlabelType** is **XmPIXMAP** and the user arms the button. These also work inside menus if the **XmNenableEtchedInMenu** resource of **XmDisplay** is **True**.

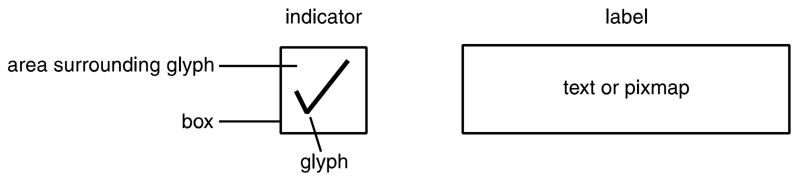
Selection rules in ToggleButton and ToggleButtonGadget are rather complicated. Many different resources control the colors and pixmaps displayed by ToggleButtons. Before getting into the selection rules, we should define a few terms. ToggleButton and ToggleButtonGadgets display the following visual areas:

- A label, which consists of either text or a pixmap.
- An optional indicator, which consists of an optional glyph, an enclosing border, and the background area between the enclosing border and the glyph. **XmNindicatorOn** specifies whether or not the ToggleButton or

ToggleButtonGadget contains an indicator area. If it does, **XmNindicatorOn** also specifies the kind of indicator. The optional glyph is either a checkmark or a cross. If **XmNindicatorType** is **XmN_OF_MANY**, then the glyph is included as part of the indicator area. If **XmNindicatorType** is something other than **XmN_OF_MANY**, then the indicator area does not contain a glyph.

Figure 12-1 illustrates a ToggleButton displaying these two visual areas:

Figure 12–1. Visual Areas of a ToggleButton



XmNset specifies whether a user or application has set or unset a ToggleButton or ToggleButtonGadget. A third possibility is that the ToggleButton or ToggleButtonGadget is neither set nor unset, but is in an indeterminate state. ToggleButton and ToggleButtonGadget must visually convey the value of **XmNset** through colors and pixmaps.

Table 12-1 summarizes the color used to paint various "background areas" in ToggleButtons or ToggleButtonGadgets. For example, the background area of an indicator is the area surrounding the glyph. Table 12-1 assumes that the ToggleButton or ToggleButtonGadget is outside of a menu and that **XmNindicatorOn** is something other than **XmINDICATOR_NONE**.

Table 12–1. Color of Various Background Areas in ToggleButton

XmNfillOnSelect	**XmNset**	**Resulting Color of "Background Area"**
True	**XmSET**	**XmNselectColor**
True	**XmUNSET**	**XmNunselectColor**
True	**XmINDETERMINATE**	Stipple of **XmNselectColor** and **XmNunselectColor**
False	any value	**XmNbackground**

If **XmNindicatorOn** is **XmINDICATOR_NONE**, then the ToggleButton or ToggleButtonGadget displays no indicator area. In this case, the colors in Table 12-1 apply to the background area of the label.

If the **XmNindicatorType** is **XmN_OF_MANY**, then the indicator area contains a glyph. Table 12-2 summarizes the color of the glyph.

Table 12–2. Color of Glyph in ToggleButton

XmNset	**Resulting Glyph Color**
XmSET	**XmNforeground**
XmUNSET	No glyph is drawn
XmINDETERMINATE	Stipple of **XmNforeground** and **XmNbackground**

If a user or application does not specify a value for **XmNselectColor**, then the **XmNenableToggleColor** resource of **XmDisplay** determines the background color.

If the label area is a pixmap, then the ToggleButton's sensitivity and the value of its **XmNset** resource determines which pixmap is rendered. Table 12-3 summarizes these combinations.

Table 12–3. Pixmap Used in Label Area of ToggleButton

XmNset	**Sensitive?**	**Resulting Pixmap**
XmSET	Yes	**XmNselectPixmap**
XmUNSET	Yes	**XmNlabelPixmap**
XmINDETERMINATE	Yes	**XmNindeterminatePixmap**
XmSET	No	**XmNselectInsensitivePixmap**
XmUNSET	No	**XmNlabelInsensitivePixmap**
XmINDETERMINATE	No	**XmNindeterminateInsensitivePixmap**

In ScrollBars, **XmNtroughColor** is used to fill the part of the slider area that is not taken up by the slider.

Motif computes a single default, known as the select color, for **XmNarmColor**, **XmNselectColor**, and **XmNtroughColor**. Motif uses the XmScreen resources

XmNdarkThreshold and **XmNlightThreshold** to determine whether the background is "dark," "light," or "medium" in the same way as for shadow colors. Motif then computes the default in the following way:

Dark background	The select color is interpolated toward white from the background.
Light background	The select color is interpolated toward black from the background.
Medium background	The select color is interpolated toward black from the background.

12.2 Application-Defined Color Generation

Motif generates default colors whenever the application creates a widget that has no specified value for one or more of the color resources. Motif does not regenerate default colors when an application changes the value of **XmNbackground** using **XtSetValues**.

An application can use **XmChangeColor** to recalculate default colors for a widget based on a new background and set the appropriate color resources in the widget. For primitives and managers, **XmChangeColor** sets **XmNbackground**, **XmNforeground**, **XmNtopShadowColor**, and **XmNbottomShadowColor**. For widgets and gadgets with select colors, **XmChangeColor** also sets the appropriate resources for those colors.

An application can use **XmGetColors** to produce default colors for a given background color without setting any resources. **XmGetColors** takes as arguments a screen pointer, a colormap, and a background pixel representing a color allocated in the colormap. **XmGetColors** also has return arguments that are pointers to pixel values for the foreground, top shadow, bottom shadow, and select colors. The function generates default colors for the given background. For each of the return arguments that is not NULL, **XmGetColors** allocates a color in the colormap and returns the pixel value at the address specified by the argument.

By default, **XmChangeColor** and **XmGetColors** calculate colors as described in the previous discussion of default colors. An application can use **XmSetColorCalculation** to change the procedure that these routines use and that Motif uses to calculate default colors when the application creates a widget. **XmSetColorCalculation** takes as its only argument a procedure of type **XmColorProc**. It sets Motif's color-calculation procedure to the new **XmColorProc** and returns the color-calculation procedure used previously. **XmSetColorCalculation** does not change the procedure used by **XmChangeColor**, **XmGetColors**, and Motif to calculate default colors for a monochrome screen.

Motif calls the **XmColorProc** when it needs to compute default colors or when the application calls **XmChangeColor** or **XmGetColors**. The **XmColorProc** takes five arguments, all pointers to *XColor* structures. The *red*, *green*, *blue*, and *pixel* members of the first structure are filled in with the background color. The procedure calculates *red*, *green*, and *blue* values for the foreground, select, top shadow, and bottom shadow colors and fills in the other four *XColor* structures with these values.

The procedure should not allocate color cells for any of these colors. Motif caches the returned *XColor* structures and allocates a color when it needs a pixel value. Usually Motif allocates a color when it computes the default value for a resource, when the application calls **XmChangeColor**, or when the application calls **XmGetColors** with a non-NULL value for one of the return pixel values. When allocating colors as a result of widget creation or a call to **XmChangeColor**, Motif uses the colormap of the widget. When allocating colors as a result of a call to **XmGetColors**, Motif uses the colormap passed as an argument to the function.

XmGetColorCalculation returns the color-calculation procedure being used at the time of the call to that routine. Calling **XmSetColorCalculation** with an argument of NULL restores the Motif default color-calculation procedure.

12.3 Colormaps

The colormap used by a widget is the value of the Core resource **XmNcolormap**. An application that does not supply its own colormap does not need to set this resource. The default for a top-level shell is the default colormap of the screen. For other widgets, the default is copied from the parent.

An application that uses its own colormap should not use **XtAppInitialize** to create the top-level shell, because the shell would then use the screen's default colormap. Instead, the application should open the display, create the colormap, and then call **XtAppCreateShell** with the colormap as the **XmNcolormap** argument.

If an application uses different colormaps for some windows in its hierarchy, it must tell the window manager about those colormaps by setting a WM_COLORMAP_WINDOWS property on the top-level window. See Chapter 18 for more information.

For more information about colormaps, see *Xlib—C Language X Interface*.

12.4 Using Pixmaps

Motif uses pixmaps supplied by the application or the user for widget borders, backgrounds, labels, shadows, focus highlights, and button arming or selection indicators. Motif also uses other pixmaps that the application or user can supply for more specific purposes: as application icons, drag icons, CascadeButton menu indicators, MessageBox symbols, and labels for insensitive buttons.

Motif provides facilities for an application to install and cache images and pixmaps. Motif also has string-to-pixmap resource converters that retrieve pixmaps from the cache or install them from files in X bitmap format or in X pixmap format. Because of these converters, both applications and users can specify pixmaps as resource values from resource files or the command line.

An application can use **XmGetPixmapByDepth** to retrieve or create a pixmap with a specified name, screen, foreground, background, and depth. **XmGetPixmapByDepth** finds or creates a pixmap in the following way:

- It searches the pixmap cache for a pixmap matching the specified name, screen, foreground, background, and depth. If it finds a matching pixmap, it returns the pixmap.
- If it does not find a matching pixmap in the cache, it searches the image cache for an image matching the specified name. If it finds a matching image, it creates and caches a pixmap of the specified depth on the specified screen, transfers the image to the pixmap, and returns the pixmap.
- If it does not find a matching image in the cache, it uses **XtResolvePathname** to search for a file of the specified name. The search path comes from the environment variable **XBMLANGPATH** or, if **XBMLANGPATH** is not set, from a default search path. Performance is usually better when **XBMLANGPATH** is set to a short list of directories than when the system uses the default search path.

If it finds such a file, it assumes that the file is in X bitmap format, reads the file, and creates and caches an image in **XYBitmap** format. It then creates and caches a pixmap of the specified depth on the specified screen, transfers the image to the pixmap, and returns the pixmap. (Since images are cached after they are found, the performance improvement from setting **XBMLANGPATH** only affects the initial search for an image.)

- If it does not find a matching X bitmap file, it returns **XmUNSPECIFIED_PIXMAP**.

Motif preinstalls a number of images in the image cache. The names and characteristics of these images are documented in the **XmInstallImage**(3) reference page. Motif offers two ways for an application to provide its own image as the source for a pixmap to be created by **XmGetPixmapByDepth**:

- The application can create its own image, usually by using **XCreateImage** or **XGetImage**. The image can be of any depth. The application can then call **XmInstallImage** to install the image in the image cache by name.
- The application or user can create a file in X bitmap format and install the file under an appropriate name in a directory that is in the search path used by **XmGetPixmapByDepth**. For a description of the X bitmap format, see *Xlib—C Language X Interface*.

Both of these mechanisms have advantages and disadvantages. An application using **XmInstallImage** can create an image of any depth. However, if it intends to use the image name in a resource specification, it must be sure to call **XmInstallImage** before creating any widgets that use the image.

An application using an X bitmap file is limited to creating an image of depth 1. However, the image is always available for use by a resource converter, and the application can use the search path mechanism of **XtResolvePathname** for such purposes as supplying different images for different locales.

XmInstallImage does not make a copy of the image when it caches it. The application must not destroy the image until it removes the image from the cache, using **XmUninstallImage**. An application can use **XmDestroyPixmap** to free a pixmap cached by **XmGetPixmapByDepth**. **XmDestroyPixmap** does not actually destroy the pixmap until all references to it are freed.

Chapter 13

Input, Focus, and Keyboard Navigation

The X server communicates with clients by means of various classes of *events*. Among these are events denoting input from the keyboard and mouse (and, in some X extensions, input from other devices). Each event is associated with a window, and the X server sends the event to any client that has expressed interest in events of that type on that window.

In the simplest case, when a keyboard or pointer event occurs, the X server sends the event to the client that has expressed interest in events of that type on the window that contains the pointer. If no such client exists, the server searches up the window's hierarchy until it finds a client that has expressed interest in events of that type on an ancestor window. In many cases, however, event processing is more complex:

- A client can *grab* a pointer button or key, the pointer or keyboard, or the entire server. The grabbing client then receives the relevant events for the duration of the grab.
- A client can set the **keyboard focus** to some window. Keyboard events that would normally be reported to this window or one of its inferiors are reported as usual,

but other events are reported with respect to the focus window. Window managers typically use this technique to implement a "click-to-type" interaction style, in which the user clicks the pointer on some window, and that window retains the keyboard focus regardless of the position of the pointer. Other clients, often in cooperation with the window manager, can set the focus to a particular window within the application hierarchy.

To insulate applications from the complexities of X event handling, Xt and Motif have developed higher-level facilities based on widgets:

- Motif supplies a VendorShell resource, **XmNkeyboardFocusPolicy**, to allow a user or application to control the model of keyboard focus in the VendorShell and its descendants. Keyboard focus can be with the widget the contains the pointer or with the widget in which the user presses Btn1.
- In the click-to-type model, the user can also use keys to navigate from widget to widget. Motif provides a model of **tab groups**, which are widgets or sets of widgets to which the user moves by using osfNextField and osfPrevField. Within a tab group, the user traverses between widgets by using osfUp, osfDown, osfLeft, and osfRight. Motif supplies resources to control whether or not a widget constitutes a tab group and whether or not the user can traverse to it using the keyboard. Motif also has a general routine, **XmProcessTraversal**, for use by the application in moving keyboard focus to a widget or tab group. The Motif menu system has a specialized traversal mechanism.
- Xt provides the basic event-dispatching loop used by most applications. Xt takes events out of the application's queue and dispatches them to the appropriate widget, usually the widget that has input focus. Xt usually invokes an *action* associated with the particular event through a table of *translations* from event specifications to action routines. The action, in turn, often invokes a callback list. An application primarily responds to events by means of its callback routines. At a lower level, it can also provide its own **event handler**, a routine invoked by the Xt dispatching loop when the widget receives events of the specified type.
- Motif and Xt provide *mnemonics* and *accelerators*, which are shortcuts for taking actions associated with a widget when the widget does not have input focus. A *mnemonic* is a keysym for a key that activates a visible button in a menu. An *accelerator* is a description for an event that invokes an action routine through a translation.

13.1 Focus Models

Motif provides two models for determining which widget within an application receives keyboard events. The focus model is determined for all descendants of a VendorShell by the value of the VendorShell resource **XmNkeyboardFocusPolicy**:

- When the value is **XmEXPLICIT**, the widget under the pointer does not necessarily receive keyboard events. The user must take an action other than moving the pointer to transfer keyboard focus to a widget. The user can usually transfer focus to a widget by pressing Btn1 on that widget or by using a keyboard navigation action to traverse to the widget.

  When the value is **XmEXPLICIT**, a widget must be *traversable* to receive keyboard events. In general, a widget is traversable when its **XmNsensitive**, **XmNancestorSensitive**, and **XmNtraversalOn** resources are True and when the widget and its ancestors are managed, realized, mapped, and viewable. See Section 13.2 for more information.

- When the value is **XmPOINTER**, the widget under the pointer receives keyboard events, unless that widget is insensitive. Keyboard navigation operations are not available. However, the user can still use the keyboard to traverse a menu system. Pressing osfMenuBar moves focus to the MenuBar, and osfMenu posts a PopupMenu if available. When the user posts a menu by using osfMenu or Btn1Up, osfActivate, or osfSelect in a CascadeButton, keyboard navigation operations are available in the menu until the menu is unposted. When the user exits the menu system, keyboard focus returns to the widget under the pointer.

MWM provides two parallel focus models for determining which top-level window receives keyboard events. The focus model is determined by the value of the **mwm** resource **keyboardFocusPolicy**:

- When the value is "explicit", the window under the pointer does not necessarily receive keyboard events. The user must take an action other than moving the pointer to transfer keyboard focus to a window. The user can usually transfer focus to a window by pressing osfSelect on that window or by using AltF6, AltTab, AltShiftTab, AltShiftF6, AltShiftEsc, or AltEsc to traverse to the window.

- When the value is "pointer", the widget under the pointer receives keyboard events. Keyboard window navigation operations are not available.

When the focus policy is "explicit", four Boolean **mwm** resources can be set to True to allow a window to receive keyboard focus automatically at specified times:

autoKeyFocus
When the window with focus is iconified or unmapped (gives focus to the window that last had it)

deiconifyKeyFocus
When the window is iconified

raiseKeyFocus
When the window is raised to the top of the stack

startupKeyFocus
When the window is mapped

13.2 Controlling Keyboard Navigation

In order to receive keyboard focus when the shell's **XmNkeyboardFocusPolicy** is **XmEXPLICIT**, a widget or gadget must meet the following conditions:

- The widget and its ancestors must not be in the process of being destroyed.
- The widget and its ancestors must be *sensitive*. A widget is sensitive when its **XmNsensitive** and **XmNancestorSensitive** resources are both True.
- The **XmNtraversalOn** resource for the widget and its ancestors must be True.
- The widget must be viewable. This means that the widget and its ancestors must be managed, realized, and (except for gadgets) mapped. Furthermore, in general, some part of the widget's rectangular area must be unobscured by the widget's ancestors.

  In a ScrolledWindow with an **XmNscrollingPolicy** of **XmAUTOMATIC**, a widget that is obscured because it is not within the clip window may be traversable if some part of the widget is within the work area and if an **XmNtraverseObscuredCallback** routine can make the widget unobscured by scrolling the window.

Most managers cannot receive focus even if they meet all these conditions. In general only primitives and gadgets are eligible to receive focus. A DrawingArea can receive focus if it meets the conditions above and if, in addition, it has no child whose **XmNtraversalOn** resource is True.

XmGetFocusWidget takes a widget argument that identifies a widget hierarchy, up to the nearest shell ancestor. It returns the widget in that hierarchy that has keyboard focus or that last had focus when the user navigated away from that hierarchy.

An application can use **XmIsTraversable** and **XmGetVisibility** to determine whether a widget is eligible to receive focus. **XmIsTraversable** returns True if the widget argument meets all the conditions described in this section. Otherwise, it returns False. This routine generally returns False if the widget argument is a composite, even if it has traversable children.

Note: When all widgets in a shell hierarchy have been made untraversable, they are considered to have lost focus. When a widget in this hierarchy is made traversable again, it regains focus.

XmGetVisibility returns a value indicating the visibility of the widget argument:

XmVISIBILITY_FULLY_OBSCURED
The widget is completely obscured by its ancestors or is not visible for some other reason (such as being unmapped or unrealized).

XmVISIBILITY_PARTIALLY_OBSCURED
Some part of the widget's rectangular area is obscured by its ancestors.

XmVISIBILITY_UNOBSCURED
None of the widget's rectangular area is obscured by its ancestors.

Note that a fully obscured widget may be traversable if it is inside the work area of an automatic ScrolledWindow with an **XmNtraverseObscuredCallback** list. See Section 13.2.5 for more information.

13.2.1 Sensitivity

Unless a widget is sensitive, Xt does not dispatch keyboard or pointer events to the widget. An insensitive widget, therefore, cannot receive keyboard focus.

A widget can be sensitive only when all its ancestors are sensitive. Two Boolean resources determine sensitivity: **XmNsensitive** and **XmNancestorSensitive**. **XmNsensitive** indicates whether the widget itself is sensitive, and **XmNancestorSensitive** indicates whether all ancestors are sensitive.

An application uses the function **XtIsSensitive** to find out whether a widget is sensitive. This function returns True when **XmNsensitive** and **XmNancestorSensitive** are both True; otherwise, it returns False.

The function **XtSetSensitive** changes the sensitivity of a widget. With an argument of False, this function sets **XmNsensitive** to False and sets each child's **XmNancestorSensitive** to False. With an argument of True, this function sets **XmNsensitive** to True and, if the widget's **XmNancestorSensitive** is also True, it sets each child's **XmNancestorSensitive** to True. The function then recursively descends the widget tree. For each descendant whose **XmNsensitive** and **XmNancestorSensitive** are both True, it sets **XmNancestorSensitive** to True for that widget's children. Otherwise, it sets **XmNancestorSensitive** to False for the descendant widget's children.

In this way, **XtSetSensitive** ensures that each widget's **XmNancestorSensitive** is True only when the parent's **XmNsensitive** and **XmNancestorSensitive** are both True. In other words, the widget is sensitive only when it and all its ancestors are sensitive. To maintain this relation, an application should always use **XtSetSensitive** to change a widget's sensitivity instead of calling **XtSetValues** on the widget's resources.

Note that **XtSetSensitive** does not modify any resources for pop-up children. If the parent widget is insensitive when a pop-up child is created, the child's **XmNancestorSensitive** will be False. **XtSetSensitive** on the parent widget will not change this value, and the child will remain insensitive. To avoid this problem, an application that creates a DialogShell or a MenuShell should either ensure that the parent is sensitive when the child is created, or specify a value of True for the child's **XmNancestorSensitive**. One way to do this is in a resource file:

```
*XmMenuShell.ancestorSensitive:   True
*XmDialogShell.ancestorSensitive: True
```

When a widget or gadget is insensitive, Motif indicates the insensitivity to the user by stippling or graying the widget.

13.2.2 XmNtraversalOn

XmNtraversalOn determines whether or not a widget is eligible to receive keyboard focus when **XmNkeyboardFocusPolicy** is **XmEXPLICIT**. When **XmNtraversalOn** is False and **XmNkeyboardFocusPolicy** is **XmEXPLICIT**, it is not possible for the user to give keyboard focus to the widget, even if the widget is sensitive and viewable. **XmNtraversalOn** has no effect when **XmNkeyboardFocusPolicy** is **XmPOINTER**.

The default value for **XmNtraversalOn** is True for most Motif widgets. Following are the exceptions:

- Separator and SeparatorGadget, where **XmNtraversalOn** is forced to False
- ScrollBar, where **XmNtraversalOn** defaults to True when it is the child of a ScrolledWindow whose **XmNscrollingPolicy** is **XmAUTOMATIC** and to False otherwise
- Label and LabelGadget, where **XmNtraversalOn** is forced to False inside menus and defaults to False otherwise
- RowColumn, where **XmNtraversalOn** defaults to True in a WorkArea and is not applicable otherwise

13.2.3 Tab Groups

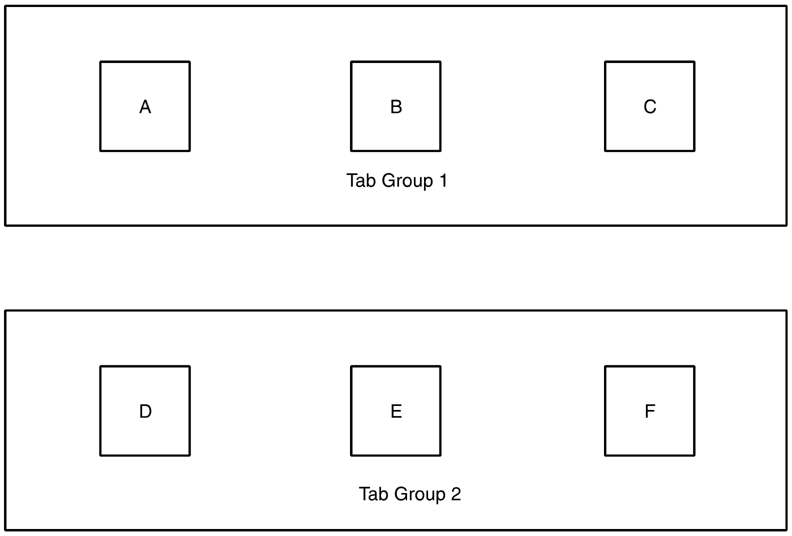
A tab group is a collection of traversable widgets or a single widget that contains a collection of traversable elements. When the shell's **XmNkeyboardFocusPolicy** is **XmEXPLICIT** and the Display's **XmNenableButtonTab** is False, the user traverses to a tab group by using osfNextField or osfPrevField. Within a tab group, when the focus is on a non-tab-group widget or an element, the user traverses to another non-tab-group widget or another element by using osfUp, osfDown, osfLeft, or osfRight.

A tab group is always represented by a widget or gadget. When the group is a collection of widgets, the tab group is typically the manager that is the parent of the widgets. When the group is a single widget like List or Text, the tab group is that widget itself.

The arrow keys do not traverse to tab groups or to non-tab-group widgets or elements outside the current tab group.

The Display resource **XmNenableButtonTab** influences the meaning of osfNextField and osfPrevField. In brief, setting **XmNenableButtonTab** to True lets a user navigate through an entire application using only one key. To examine **XmNenableButtonTab** more closely, consider an application containing the two tab groups shown in Figure 13-1. As the figure shows, each tab group contains three widgets. You can assume that the **XmNlayoutDirection** resource is set to **XmLEFT_TO_RIGHT_TOP_TO_BOTTOM**.

Figure 13–1. An Application With Two Tab Groups



Suppose that **XmNenableButtonTab** is False. Further suppose Widget B of Tab Group 1 has traversal focus. If **XmNenableButtonTab** is False, then an osfNextField event will send traversal focus to the first widget in Tab Group 1 (probably Widget D). That is, the application responds to the osfNextField event by traversing to the next tab group. If the user provides a second osfNextField event, Motif traverses back to the first widget in Tab Group 1 (probably Widget A).

Now suppose that **XmNenableButtonTab** is True and that traversal focus is at Widget B. Whenever the user provides an osfNextField event, Motif traverses to the next traversable widget in the application, whether or not that field is in the current tab group. Therefore, pressing osfNextField six times would send traversal focus to Widget C, then Widget D, then Widget E, then Widget F, then Widget A, and then back to Widget B.

To be eligible for traversal, a tab group must meet all the conditions discussed in Section 13.2, except that a manager that is a tab group and meets the other conditions is eligible for traversal as long as it contains a descendant that can receive focus. If the tab group does not meet these conditions, the osfNextField and osfPrevField actions ignore the tab group.

Within a tab group, non-tab-group widgets must also meet all the conditions discussed in Section 13.2 to be eligible for traversal. If they do not meet these conditions, the arrow key actions ignore the widgets.

Whether or not a widget is a tab group is determined by the value of the **XmNnavigationType** resource. The two primary values for this resource are **XmTAB_GROUP**, which indicates that the widget is a tab group, and **XmNONE**, which indicates that it is not.

When the user traverses to the next or previous tab group, the direction of the traversal is usually determined by the relative locations of the current and target groups. In a left-to-right language environment, traversal to each subsequent tab group proceeds from left to right and top to bottom. At the bottom right, traversal wraps to the tab group at the top left. Traversal to previous tab groups proceeds in the opposite direction.

The application can control the order of traversal by specifying an **XmNnavigationType** of **XmEXCLUSIVE_TAB_GROUP** for a widget in the hierarchy. When any widget in a hierarchy has an **XmNnavigationType** of **XmEXCLUSIVE_TAB_GROUP**, osfNextField and osfPrevField do not move to any widgets in that hierarchy that have been designated tab groups by means of an **XmNnavigationType** of **XmTAB_GROUP**. But osfNextField and osfPrevField do move to widgets whose **XmNnavigationType** is **XmSTICKY_TAB_GROUP**, even if some widgets are exclusive tab groups. Thus, an application that uses **XmEXCLUSIVE_TAB_GROUP** to control traversal must be sure that all tab groups have an **XmNnavigationType** of either **XmEXCLUSIVE_TAB_GROUP** or **XmSTICKY_TAB_GROUP**.

When any widget in a hierarchy has an **XmNnavigationType** of **XmEXCLUSIVE_TAB_GROUP**, traversal to subsequent tab groups does not depend on the relative locations of the groups. Instead, it proceeds to widgets in the order in which their **XmNnavigationType** resources were specified as **XmEXCLUSIVE_TAB_GROUP** or **XmSTICKY_TAB_GROUP**, either by creating the widgets with that value or by calling **XtSetValues**. That is, traversal proceeds to the widget whose **XmNnavigationType** was next specified to be **XmEXCLUSIVE_TAB_GROUP** or **XmSTICKY_TAB_GROUP**. Traversal to previous tab groups proceeds in the opposite direction.

Within a tab group whose **XmNnavigationType** is **XmEXCLUSIVE_TAB_GROUP**, the arrow keys do not behave the same way as they would if the **XmNnavigationType** were either **XmTAB_GROUP** or **XmSTICKY_TAB_GROUP**. With **XmTAB_GROUP** or **XmSTICKY_TAB_GROUP**, the direction of traversal using the arrow keys depends on the relative locations of the tab group's children. Pressing osfRight moves to the next traversable child to the right of the child with the focus; osfDown moves to the next traversable child below the child with the focus; and so on.

With **XmEXCLUSIVE_TAB_GROUP**, traversal using the arrow keys depends on the order of the tab group's list of children, not on the relative locations of the children. Pressing osfRight has the same effect as osfDown: both move to the next traversable child in the tab group's list of children. Pressing osfLeft has the same effect as osfUp: both move to the previous traversable child in the tab group's list of children.

There are three principal differences between **XmEXCLUSIVE_TAB_GROUP** and **XmSTICKY_TAB_GROUP**:

- **XmEXCLUSIVE_TAB_GROUP** has the effect of disabling traversal to tab groups that have an **XmNnavigationType** of **XmTAB_GROUP**. **XmSTICKY_TAB_GROUP** does not; it simply ensures that traversal to that tab group is possible, even when some widget in the hierarchy has an **XmNnavigationType** of **XmEXCLUSIVE_TAB_GROUP**.
- **XmEXCLUSIVE_TAB_GROUP** changes the order of traversal of tab groups within the widget hierarchy. **XmSTICKY_TAB_GROUP** does not.
- **XmEXCLUSIVE_TAB_GROUP** changes the order of traversal of widgets inside the tab group. **XmSTICKY_TAB_GROUP** does not.

The function **XmAddTabGroup** has the same effect as calling **XtSetValues** with an **XmNnavigationType** of **XmEXCLUSIVE_TAB_GROUP**. The function **XmRemoveTabGroup** has the same effect as calling **XtSetValues** with an **XmNnavigationType** of **XmNONE**. **XmAddTabGroup** and **XmRemoveTabGroup** are obsolete and exist for compatibility with earlier releases of Motif.

All Motif managers except RowColumn have a default **XmNnavigationType** of **XmTAB_GROUP**. In RowColumn, **XmNnavigationType** is not applicable for MenuBars, PulldownMenus, and PopupMenus. For a WorkArea the default is **XmTAB_GROUP**, and for an OptionMenu the default is **XmNONE**.

All Motif primitives except List, ScrollBar, Text, and TextField have a default **XmNnavigationType** of **XmNONE**. The default for List, Text, and TextField is **XmTAB_GROUP**, and the default for ScrollBar is **XmSTICKY_TAB_GROUP**. These are all controls that have their own internal navigation.

Motif sets the navigation type of widgets in some situations. In particular:

- The child of a shell always behaves as a tab group, no matter what the value of its **XmNnavigationType**.
- Panes and sashes inside PanedWindows have a default **XmNnavigationType** of **XmTAB_GROUP**. If the **XmNnavigationType** of a pane is **XmNONE** when the pane is created, Motif sets the value of that resource to **XmTAB_GROUP**.
- SelectionBox and its subclasses set the **XmNnavigationType** of their automatically created List and Text children to **XmSTICKY_TAB_GROUP**.

The function **XmGetTabGroup** returns the tab group that contains a widget. If the widget itself is a tab group or a shell, it returns that widget. If neither the widget nor any ancestor up to the nearest shell is a tab group, it returns the nearest ancestor that is a shell. Otherwise, it returns the nearest ancestor that is a tab group.

13.2.3.1 Controlling Tab Group Traversal Order

By default, osfNextField and osfPrevField traverse to successive tab groups in order of layout, from left to right and top to bottom, within a parent tab group, before proceeding in layout order to the next tab group that is a sibling of the parent. Traversal order changes when any widget in a shell hierarchy has an **XmNnavigationType** of **XmEXCLUSIVE_TAB_GROUP**. In this case, osfNextField and osfPrevField traverse only to widgets in the hierarchy whose **XmNnavigationType** is either **XmEXCLUSIVE_TAB_GROUP** or **XmSTICKY_TAB_GROUP**. The traversal order is the order in which the widgets' **XmNnavigationType** was specified to be either **XmEXCLUSIVE_TAB_GROUP** or **XmSTICKY_TAB_GROUP**.

This mechanism gives an application the means to control tab group traversal order. An application must do the following:

- Ensure that at least one widget in the shell hierarchy has an **XmNnavigationType** of **XmEXCLUSIVE_TAB_GROUP**

- Ensure that all widgets that the application wants to be tab groups have an **XmNnavigationType** of either **XmEXCLUSIVE_TAB_GROUP** or **XmSTICKY_TAB_GROUP**
- Specify values for the tab groups' **XmNnavigationType**, using either creation argument lists or **XtSetValues**, in the order in which the tab groups are to be traversed

Note that, when a tab group has an **XmNnavigationType** of **XmEXCLUSIVE_TAB_GROUP**, traversal to non-tab-group widgets inside that tab group proceeds in the order in which the children appear in their parents' **XmNchildren** lists. If the application wants to specify the order of tab group traversal but still wants traversal of non-tab-group widgets to proceed according to layout, it should select one widget in the hierarchy to have an **XmNnavigationType** of **XmEXCLUSIVE_TAB_GROUP**. This tab group should contain no non-tab-group widgets. For example, it could be the MainWindow if the MainWindow contains only tab groups, or it could be a primitive tab group, such as List or Text. The application should then specify an **XmNnavigationType** of **XmSTICKY_TAB_GROUP** for all other tab groups in the hierarchy.

13.2.4 Initial Focus

A tab group may contain any combination of tab group and non-tab-group widgets. A tab group that contains other widgets cannot receive focus itself. When the user traverses to a composite tab group, Motif gives focus to some widget within the tab group.

Motif uses the Manager resource **XmNinitialFocus** in determining which widget receives focus. The value of **XmNinitialFocus** is a widget that meets the following conditions:

- The widget must be either a tab group or a non-tab-group widget that can receive keyboard focus. In general, a widget can receive keyboard focus when it is a primitive, a gadget, or a manager (such as a DrawingArea with no traversable children) that acts as a primitive.
- The widget must not be a descendant of a tab group that is itself a descendant of the manager. That is, the widget cannot be contained within a tab group that is nested inside the manager.

- The widget and its ancestors must have a value of True for their **XmNtraversalOn** resources.

If the widget does not meet these conditions, **XmNinitialFocus** is treated as if the value were NULL.

Motif uses **XmNinitialFocus** to determine which widget receives focus in these situations:

- When the manager is the child of a shell and the shell hierarchy receives focus for the first time
- When focus is inside the shell hierarchy, the manager is a composite tab group, and the user traverses to the manager using the keyboard

Motif then determines focus as follows:

- If **XmNinitialFocus** is a traversable non-tab-group widget, that widget receives focus.
- If **XmNinitialFocus** is a traversable tab group, that tab group receives focus. If that tab group is a composite with descendant tab groups or traversable non-tab-group widgets, these procedures are used recursively to assign focus to a descendant of that tab group.
- If **XmNinitialFocus** is NULL, the first traversable non-tab-group widget that is not contained within a nested tab group receives focus.
- If **XmNinitialFocus** is NULL and no traversable non-tab-group widget exists, the first traversable tab group that is not contained within a nested tab group receives focus. If that tab group is a composite with descendant tab groups or traversable non-tab-group widgets, these procedures are used recursively to assign focus to a descendant of that tab group.

If a shell hierarchy regains focus after losing it, focus returns to the widget that had the focus at the time it left the hierarchy.

The use of **XmNinitialFocus** is undefined if the manager is a MenuBar, PulldownMenu, PopupMenu, or OptionMenu.

13.2.5 Traversing to Obscured Widgets

In general, a widget is not eligible to receive focus unless some part of its rectangular area is unobscured by its ancestors. However, it may be possible to traverse to a widget that is a descendant of a ScrolledWindow whose **XmNscrollingPolicy** is **XmAUTOMATIC**, even if that widget is not within the ScrolledWindow’s clip window. Traversal to such a widget is possible under the following conditions:

- Some part of the widget’s rectangular area is within the bounds of the ScrolledWindow’s work window.
- The ScrolledWindow’s clip window is completely unobscured by its ancestors. If the ScrolledWindow is a descendant of another ScrolledWindow, it must be unobscured by the ancestor’s work window but may be outside the ancestor’s clip window.
- The ScrolledWindow has a procedure on its **XmNtraverseObscuredCallback** list that can bring some part of the widget’s rectangular area into the clip window.
- The widget meets the other conditions for receiving focus described in Section 13.2.

Whenever the user attempts to traverse to such a widget and the widget is partially or fully obscured by the clip window, Motif calls the ScrolledWindow’s **XmNtraverseObscuredCallback** procedures. If the ScrolledWindow has one or more ancestor ScrolledWindows, Motif calls the **XmNtraverseObscuredCallback** list for each ScrolledWindow whose clip window obscures the traversal target, from the lowest level of the hierarchy to the highest. The **XmNtraverseObscuredCallback** procedure can try to bring the widget into the clip window if necessary, usually by calling **XmScrollVisible**. If the target widget is traversable after the **XmNtraverseObscuredCallback** procedures are invoked, that widget receives focus.

A procedure can determine the visibility of a widget by calling **XmGetVisibility**.

13.2.6 XmProcessTraversal

The principal routine for traversing to a widget is **XmProcessTraversal**. Motif uses this routine to effect traversal when the user presses an arrow key, osfNextField, or osfPrevField. An application can use **XmProcessTraversal** to implement its own traversal actions.

XmProcessTraversal takes two arguments, a widget and a constant specifying a traversal action. The routine uses the widget argument to identify the hierarchy that contains the widget and that has its root at the nearest shell. If that shell does not currently have the focus, any changes to the element with focus within that shell will not occur until the next time the shell receives focus.

The traversal action argument identifies one of three kinds of action to take. The following descriptions of these actions refer to traversable non-tab-group widgets and traversable tab groups. A traversable non-tab-group widget is a widget that is not a tab group and that meets all the conditions for receiving focus discussed in Section 13.2. A traversable tab group is a tab group widget that meets the same conditions, except that a manager that is a tab group and meets the other conditions is also traversable as long as it contains a descendant that can receive focus.

The routine begins the traversal action from the widget in the hierarchy that currently has keyboard focus or that last had focus when the user traversed away from the shell hierarchy.

Note that **XmProcessTraversal** cannot be called recursively. In particular, an application cannot call this routine from an **XmNfocusCallback** or **XmNlosingFocusCallback** procedure.

The descriptions in the following three subsections all assume that **XmNlayoutDirection** is set to **XmLEFT_TO_RIGHT_TOP_TO_BOTTOM**. Later in this section, we will explore the influence of **XmNlayoutDirection** in greater detail.

13.2.6.1 Traversal to a Non-Tab-Group Widget

This kind of traversal is possible only when the widget that currently has focus is not a tab group. Also, these actions do not move focus from one tab group to another. The actions first determine the containing tab group. This is the tab group containing the widget that currently has focus. The actions traverse only to a non-tab-group widget within the containing tab group.

XmTRAVERSE_RIGHT

If the **XmNnavigationType** of the containing tab group is not **XmEXCLUSIVE_TAB_GROUP**, focus moves to the next traversable non-tab-group widget to the right of the widget that currently has focus. At the right side of the tab group, this action wraps to the non-tab-group widget at the left side and next toward the bottom. At the lower right corner of the tab group, this action wraps to the non-tab-group widget at the upper left.

If the **XmNnavigationType** of the containing tab group is **XmEXCLUSIVE_TAB_GROUP**, focus moves to the next traversable non-tab-group widget in the tab group, proceeding in the order in which the widgets appear in their parents’ **XmNchildren** lists. After the last widget in the tab group, this action wraps to the first non-tab-group widget.

XmTRAVERSE_LEFT

If the **XmNnavigationType** of the containing tab group is not **XmEXCLUSIVE_TAB_GROUP**, focus moves to the next traversable non-tab-group widget to the left of the widget that currently has focus. At the left side of the tab group, this action wraps to the non-tab-group widget at the right side and next toward the top. At the upper left corner of the tab group, this action wraps to the non-tab-group widget at the lower right.

If the **XmNnavigationType** of the containing tab group is **XmEXCLUSIVE_TAB_GROUP**, focus moves to the previous traversable non-tab-group widget in the tab group, proceeding in the reverse order in which the widgets appear in their parents’ **XmNchildren** lists. After the first widget in the tab group, this action wraps to the last non-tab-group widget.

XmTRAVERSE_DOWN

If the **XmNnavigationType** of the containing tab group is not **XmEXCLUSIVE_TAB_GROUP**, focus moves to the next traversable non-tab-group widget below the widget that currently has focus. At the bottom of the tab group, this action wraps to the non-tab-group widget at the top and next toward the right. At the lower right corner of the tab group, this action wraps to the non-tab-group widget at the upper left.

If the **XmNnavigationType** of the containing tab group is **XmEXCLUSIVE_TAB_GROUP**, focus moves to the next traversable non-tab-group widget in the tab group, proceeding in the order in which the widgets appear in their parents’ **XmNchildren** lists. After the last widget in the tab group, this action wraps to the first non-tab-group widget.

XmTRAVERSE_UP

If the **XmNnavigationType** of the containing tab group is not **XmEXCLUSIVE_TAB_GROUP**, focus moves to the next traversable non-tab-group widget above the widget that currently has focus. At the top of the tab group, this action wraps to the non-tab-group widget at the bottom and next toward the left. At the upper left corner of the tab group, this action wraps to the non-tab-group widget at the lower right.

If the **XmNnavigationType** of the containing tab group is **XmEXCLUSIVE_TAB_GROUP**, focus moves to the previous traversable non-tab-group widget in the tab group, proceeding in the reverse order in which the widgets appear in their parents' **XmNchildren** lists. After the first widget in the tab group, this action wraps to the last non-tab-group widget.

XmTRAVERSE_NEXT

Focus moves to the next traversable non-tab-group widget in the tab group, proceeding in the order in which the widgets appear in their parents' **XmNchildren** lists. After the last widget in the tab group, this action wraps to the first non-tab-group widget.

XmTRAVERSE_PREV

Focus moves to the previous traversable non-tab-group widget in the tab group, proceeding in the reverse order in which the widgets appear in their parents' **XmNchildren** lists. After the first widget in the tab group, this action wraps to the last non-tab-group widget.

XmTRAVERSE_HOME

If the **XmNnavigationType** of the containing tab group is not **XmEXCLUSIVE_TAB_GROUP**, focus moves to the first traversable non-tab-group widget at the top left corner of the tab group.

If the **XmNnavigationType** of the containing tab group is **XmEXCLUSIVE_TAB_GROUP**, focus moves to the first traversable non-tab-group widget in the tab group, according to the order in which the widgets appear in their parents' **XmNchildren** lists.

13.2.6.2 Traversal to a Tab Group

The following actions begin by determining the current widget hierarchy and the containing tab group. The current widget hierarchy is the widget hierarchy whose root is the nearest shell ancestor of the widget that currently has focus. The containing tab group is is the tab group containing the widget that currently has focus.

XmTRAVERSE_NEXT_TAB_GROUP

If no tab group in the current widget hierarchy has a value of **XmEXCLUSIVE_TAB_GROUP** for **XmNnavigationType**, focus goes to the next traversable tab group that is to the right of the widget with current focus and is within the containing tab group. At the right side of the containing tab group, this action wraps to the tab group at the left side and next toward the bottom. At the lower right corner of the containing tab group, this action recursively moves up one level in the hierarchy. Focus then goes to the next traversable tab group that is to the right of the original containing tab group and is within the tab group that contains that one. At the lower right corner of the topmost tab group in the hierarchy, this action wraps to the first traversable tab group at the upper left corner of the topmost tab group.

If any tab group in the current widget hierarchy has a value of **XmEXCLUSIVE_TAB_GROUP** for **XmNnavigationType**, focus goes to the next traversable tab group in the hierarchy, in the order in which the **XmNnavigationType** resources of the tab groups were set to **XmEXCLUSIVE_TAB_GROUP** or **XmSTICKY_TAB_GROUP**. After the last tab group in the hierarchy, this action wraps to the first tab group.

XmTRAVERSE_PREV_TAB_GROUP

If no tab group in the current widget hierarchy has a value of **XmEXCLUSIVE_TAB_GROUP** for **XmNnavigationType**, focus goes to the next traversable tab group that is to the left of the widget with current focus and is within the containing tab group. At the left side of the containing tab group, this action wraps to the tab group at the right side and next toward the top. At the upper left corner of the containing tab group, this action recursively moves up one level in the hierarchy. Focus then goes to the next traversable tab group that is to the left of the original containing tab group and is within the tab group that contains that one. At the upper left corner of the topmost tab group in the hierarchy, this action wraps to the first traversable tab group at the lower right corner of the topmost tab group.

If any tab group in the current widget hierarchy has a value of **XmEXCLUSIVE_TAB_GROUP** for **XmNnavigationType**, focus goes to the previous traversable tab group in the hierarchy, in the reverse order in which the **XmNnavigationType** resources of the tab groups were set to either **XmEXCLUSIVE_TAB_GROUP** or **XmSTICKY_TAB_GROUP**. After the first tab group in the hierarchy, this action wraps to the last tab group.

13.2.6.3 Traversal to Any Widget

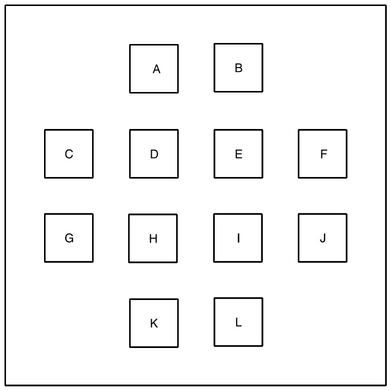
In the following case, the widget argument is the widget to which **XmProcessTraversal** tries to give focus.

XmTRAVERSE_CURRENT
Focus goes to the widget argument if that widget is a traversable non-tab-group widget or tab group.

13.2.6.4 Effect of XmNlayoutDirection on XmProcessTraversal

The descriptions in the preceding subsections all assumed that **XmNlayoutDirection** was set to **XmLEFT_TO_RIGHT_TOP_TO_BOTTOM**. When **XmNlayoutDirection** is set to some other direction, the navigation rules are somewhat different. To help explain these differences, consider the application shown in Figure 13-2. This application consists of one tab group. This one tab group consists of 12 widgets.

Figure 13–2. An Application With 12 Widgets in One Tab Group



Given the application shown in Figure 13-2, Table 13-1 summarizes what happens when the application calls **XmProcessTraversal** with a direction argument of **XmTRAVERSE_HOME**.

Table 13–1. Effect of XmNlayoutDirection on XmTRAVERSE_HOME

XmNlayoutDirection	**Home Widget**
XmLEFT_TO_RIGHT_TOP_TO_BOTTOM	A
XmRIGHT_TO_LEFT_TOP_TO_BOTTOM	B
XmLEFT_TO_RIGHT_BOTTOM_TO_TOP	K
XmRIGHT_TO_LEFT_BOTTOM_TO_TOP	L
XmTOP_TO_BOTTOM_LEFT_TO_RIGHT	C
XmTOP_TO_BOTTOM_RIGHT_TO_LEFT	F
XmBOTTOM_TO_TOP_LEFT_TO_RIGHT	G
XmBOTTOM_TO_TOP_RIGHT_TO_LEFT	J

Table 13-2 summarizes the influence of **XmNlayoutDirection** on **XmProcessTraversal** in the application shown in Figure 13-2. For example, suppose that the second argument to **XmProcessTraversal** is **XmTRAVERSE_RIGHT** and that the **XmNlayoutDirection** resource is set to **XmRIGHT_TO_LEFT_TOP_TO_BOTTOM**. If **XmNinitialFocus** for the tab group manager is NULL, Table 13-2 says that initial focus will default to widget B. When **XmProcessTraversal** is called again with **XmTRAVERSE_RIGHT** as its direction argument, the application will navigate to widget K, then to widget L, and so on. After navigating through all 12 widgets, the application will navigate around to the starting widget, which in this case is widget B.

One way to understand the navigation rules is to ask yourself whether the application is trying to traverse forwards or traverse backwards. To understand forwards and backwards, suppose that **XmNlayoutDirection** is set to **XmLEFT_TO_RIGHT_TOP_TO_BOTTOM**. In this case

- **XmTRAVERSE_RIGHT** and **XmTRAVERSE_DOWN** are forward traversals (with the flow)
- **XmTRAVERSE_LEFT** and **XmTRAVERSE_UP** are backwards traversals (against the flow)

Understanding forwards and backwards helps you understand what Motif does when the application wants to traverse past the last widget in a particular direction. For example, what happens when the application wants to traverse right but there are no widgets to the right of the current widget? If the traversal is forwards, then upon reaching the last widget in given direction, Motif navigates towards the direction specified by **XmNlayoutDirection**. If the traversal is backwards, Motif navigates away from the direction specified by **XmNlayoutDirection**.

Table 13–2. Effect of XmNlayoutDirection on XmProcessTraversal

XmProcessTraversal	**XmNlayoutDirection**	**Sequence**
XmTRAVERSE_RIGHT	**XmLEFT_TO_RIGHT_TOP_TO_BOTTOM**	A,B,C,D,E,F,G,H,I,J,K,L
	XmRIGHT_TO_LEFT_TOP_TO_BOTTOM	B,K,L,G,H,I,J,C,D,E,F,A
	XmLEFT_TO_RIGHT_BOTTOM_TO_TOP	K,L,G,H,I,J,C,D,E,F,A,B
	XmRIGHT_TO_LEFT_BOTTOM_TO_TOP	L,A,B,C,D,E,F,G,H,I,J,K
	XmTOP_TO_BOTTOM_LEFT_TO_RIGHT	C,D,E,F,G,H,I,J,K,L,A,B
	XmTOP_TO_BOTTOM_RIGHT_TO_LEFT	F,A,B,K,L,G,H,I,J,C,D,E
	XmBOTTOM_TO_TOP_LEFT_TO_RIGHT	G,H,I,J,C,D,E,F,A,B,K,L
	XmBOTTOM_TO_TOP_RIGHT_TO_LEFT	J,K,L,A,B,C,D,E,F,G,H,I
XmTRAVERSE_LEFT	**XmLEFT_TO_RIGHT_TOP_TO_BOTTOM**	A,L,K,J,I,H,G,F,E,D,C,B
	XmRIGHT_TO_LEFT_TOP_TO_BOTTOM	B,A,F,E,D,C,J,I,H,G,L,K
	XmLEFT_TO_RIGHT_BOTTOM_TO_TOP	K,B,A,F,E,D,C,J,I,H,G,L
	XmRIGHT_TO_LEFT_BOTTOM_TO_TOP	L,K,J,I,H,G,F,E,D,C,B,A
	XmTOP_TO_BOTTOM_LEFT_TO_RIGHT	C,B,A,L,K,J,I,H,G,F,E,D
	XmTOP_TO_BOTTOM_RIGHT_TO_LEFT	F,E,D,C,J,I,H,G,L,K,B,A
	XmBOTTOM_TO_TOP_LEFT_TO_RIGHT	G,L,K,B,A,F,E,D,C,J,I,H
	XmBOTTOM_TO_TOP_RIGHT_TO_LEFT	J,I,H,G,F,E,D,C,B,A,L,K
XmTRAVERSE_DOWN	**XmLEFT_TO_RIGHT_TOP_TO_BOTTOM**	A,D,H,K,B,E,I,L,F,J,C,G
	XmRIGHT_TO_LEFT_TOP_TO_BOTTOM	B,E,I,L,A,D,H,K,C,G,F,J
	XmLEFT_TO_RIGHT_BOTTOM_TO_TOP	K,C,G,F,J,B,E,I,L,A,D,H
	XmRIGHT_TO_LEFT_BOTTOM_TO_TOP	L,F,J,C,G,A,D,H,K,B,E,I
	XmTOP_TO_BOTTOM_LEFT_TO_RIGHT	C,G,A,D,H,K,B,E,I,L,F,J
	XmTOP_TO_BOTTOM_RIGHT_TO_LEFT	F,J,B,E,I,L,A,D,H,K,C,G

	XmBOTTOM_TO_TOP_LEFT_TO_RIGHT	G,F,J,B,E,I,L,A,D,H,K,C
	XmBOTTOM_TO_TOP_RIGHT_TO_LEFT	J,C,G,A,D,H,K,B,E,I,L,F
XmTRAVERSE_UP	**XmLEFT_TO_RIGHT_TOP_TO_BOTTOM**	A,G,C,J,F,L,I,E,B,K,H,D
	XmRIGHT_TO_LEFT_TOP_TO_BOTTOM	B,J,F,G,C,K,H,D,A,L,I,E
	XmLEFT_TO_RIGHT_BOTTOM_TO_TOP	K,H,D,A,L,I,E,B,J,F,G,C
	XmRIGHT_TO_LEFT_BOTTOM_TO_TOP	L,I,E,B,K,H,D,A,G,C,J,F
	XmTOP_TO_BOTTOM_LEFT_TO_RIGHT	C,J,F,L,I,E,B,K,H,D,A,G
	XmTOP_TO_BOTTOM_RIGHT_TO_LEFT	F,G,C,K,H,D,A,L,I,E,B,J
	XmBOTTOM_TO_TOP_LEFT_TO_RIGHT	G,C,K,H,D,A,L,I,E,B,J,F
	XmBOTTOM_TO_TOP_RIGHT_TO_LEFT	J,F,L,I,E,B,K,H,D,A,G,C

13.2.7 Focus Callbacks

BulletinBoard, Text, and TextField have **XmNfocusCallback** callback lists. Motif invokes the procedures on these lists when these widgets receive keyboard focus. A callback procedure may change the widget's state to reflect the new focus, but it should not try to change the focus and, in particular, must not call **XmProcessTraversal**.

Text and TextField also have **XmNlosingFocusCallback** callback lists. The Text and TextField traversal actions invoke these procedures before traversing to another widget. The third argument to each procedure is a pointer to an **XmTextVerifyCallbackStruct** structure whose *reason* member is **XmCR_LOSING_FOCUS**. If a callback procedure sets the *doit* member of this structure to False, the traversal action does not carry out the traversal. In this way the application can prevent a user from traversing out of the widget by means of these actions.

Motif also invokes the **XmNlosingFocusCallback** procedures when the widget loses focus by some other means. For example, the user might click Btn1 in another traversable widget, or when the shell's **XmNkeyboardFocusPolicy** is **XmPOINTER**, the user might move the pointer into another widget. In such cases, setting the *doit* member of the callback structure has no effect.

13.3 Translations and Actions

In Xt, the primary means of associating an input event with a widget-specific procedure is the combination of translations and actions. Each widget (but not gadget) instance contains a table of translations that maps event descriptions to procedure names. Each widget instance also has a table of actions that maps these procedure names to actual procedures. When a widget receives an input event, the Xt event-dispatching facility looks up the event in the translation table, looks up the associated procedure in the action table, and invokes the action procedure itself. This procedure usually takes some action to change the widget state and often invokes callback procedures.

13.3.1 Translation Table Format

An application or user specifies a translation table as a string whose format is defined in *X Toolkit Intrinsics—C Language Interface*. In general, the table consists of individual translations separated by "\n". Each translation consists of an event description sequence, a colon, and one or more associated procedure names. Each procedure name also has a list of parameters within parentheses to be passed to the procedure when it is invoked as a result of that translation.

An event description in general consists of an optional list of modifiers, an event type within angle brackets (< and >), an optional repeat count within parentheses, and an optional event detail. Modifiers apply only to key, button, motion, enter, and leave events. If an exclamation point (!) precedes the modifiers, then the modifiers in the list and no others must be asserted for the action to be invoked. Otherwise, the modifiers in the list must be asserted, but others may be as well. A tilde (~) before any modifier means that that modifier must not be asserted. If the modifier list is empty, any modifiers may be asserted.

The detail field varies depending on the event type. The most common use is to identify the keysym for a **KeyPress** or **KeyRelease** event.

Event descriptions in a sequence are separated by commas. Mouse motion is discarded if it occurs between events in a sequence that does not include explicit motion events. This allows the following sort of translation to invoke an action even if the mouse moves between button press and release:

```
<Btn1Down>,<Btn1Up>: action()
```

Following are some important considerations in using translations:

- More specific events should always precede less specific events in the table:

  ```
  Ctrl<Key>space: action_1()
  <Key>space: action_2()
  ```

- Translations with event sequences that are noninitial subsequences of other translations are not invoked when the events occur as part of the longer sequence. For instance, **up_action()** in the following example would not be invoked on a button release that followed a button press:

  ```
  <Btn1Down>,<Btn1Up>: click_action()
  <Btn1Up>: up_action()
  ```

- Event descriptions that use a repeat count expand into longer sequences. For example, the following descriptions are more or less equivalent:

  ```
  <Btn1Up>(2): double_click()
  <Btn1Up>,<Btn1Down>,<Btn1Up>: double_click()
  ```

  This result, combined with the implicit insertion of motion events between any two other events, means that motion translations cannot exist in a table with multiclick translations.

See *X Toolkit Intrinsics—C Language Interface* for more information on the format of translation tables.

13.3.2 Using Translations

One translation table frequently needs to be merged with another. For example, a user may want to add one or more translations to a widget’s default translations. A translation table may begin with one of three directives that specifies how the table is to be merged with an existing table:

#replace The new translation table should completely replace any existing table. This is the default if no directive is specified.

#augment The new translation table should be added to any existing table. If the two tables contain duplicate event descriptions, the translations in the existing table are used.

#override The new translation table should be added to any existing table. If the two tables contain duplicate event descriptions, the translations in the new table are used.

A widget's translation table is the value of the Core **XmNtranslations** resource. The initial value is determined in the following way:

- If a non-NULL value is specified for **XmNtranslations** in the widget creation argument list, the widget class translations are merged with that value, in order, and the resulting table is used.
- Otherwise, the following tables are merged, in order, and the resulting table is used:
  - — The widget class translations
  - — The value of the **baseTranslations** resource from the resource database
  - — The value of the **XmNtranslations** resource from the resource database or, if no value was specified, the default value for the widget's **XmNtranslations**

To take advantage of this initialization ordering, an application should usually provide any translations of its own by specifying a value for **baseTranslations** rather than **XmNtranslations** in an application class defaults file or a fallback resource list. This essentially reserves **XmNtranslations** to the user. The application can change the widget class translations by specifying **baseTranslations**, and the user can change the application's translations by specifying **XmNtranslations**.

As the value of a widget's **XmNtranslations**, a translation table must be in a parsed format rather than a string. The string-to-translation-table converter parses a resource string into a translation table. An application can also use **XtParseTranslationTable** to compile a translation table string into the parsed format. The application can then merge the parsed table with a widget's **XmNtranslations** in three ways:

- **XtAugmentTranslations** merges the parsed table in **#augment** mode
- **XtOverrideTranslations** merges the parsed table in **#override** mode
- **XtSetValues** of **XmNtranslations** replaces the existing value with the parsed table

Some Motif widgets merge additional translations in their *initialize* and **set_values** methods. This process may make it impossible for an application or user to override some translations by means of resource files. For example, for some widgets it may not be possible to change traversal translations in this way.

13.3.3 Actions

Each widget instance has a table that maps action procedure names, as they appear in translation tables, to actual action procedures. When an action is invoked through a translation, Xt looks up the action procedure name in this table and calls the associated procedure.

Each widget class may have its own action table. In addition, an application can use **XtAppAddActions** to add entries to an action table associated with the application context. Only one such table exists per application context. If a call to **XtAppAddActions** contains an action name that is already in the table, the action name becomes associated with the action procedure supplied in the call to **XtAppAddActions**, overriding the existing action.

Xt creates a widget's action table when the widget is realized. It uses actions from the following action tables, those listed first having highest precedence:

- The action tables for the widget's class and its superclasses, in subclass-to-superclass order
- The action tables for the parent's class and its superclasses, in subclass-to-superclass order, and so on up the widget hierarchy
- The application context action table (created by calls to **XtAppAddActions**)

This ordering means that an application cannot use **XtAppAddActions** to provide a new action procedure for an action name that is already registered by a widget class. To do that, the application must supply a translation that maps the event to an action name that is not registered by the class. The application must then call **XtAppAddActions** to supply a procedure for the action name.

An action procedure is a function of type *XtActionProc*. This function receives four arguments:

- The widget
- The event, or the last event of a sequence, that caused the procedure to be invoked
- A list of strings representing the parameters specified for this action in the translation table
- An integer representing the number of parameters in the parameter list

An application can use the parameter list to perform a number of related actions in a single action routine. For example, a widget might have the following translations:

```
c <Key> osfLeft: move-object(left) \n\
c <Key> osfRight: move-object(right) \n\
c <Key> osfUp: move-object(up) \n\
c <Key> osfDown: move-object(down)
```

The routine implementing the move-object() action is passed one of the strings "left", "right", "up", and "down" as the only item in the parameter list, depending on which key event invoked the action. The routine performs the action appropriate for this parameter.

13.3.4 Bindings for osf Keysyms

Motif maintains a client-side mechanism for mapping one set of keysyms to another set. This mapping allows Motif widgets and applications to use a single set of keysyms in translation tables and also allows applications and users to customize the keysyms used in the translations for the particular keyboard used with the display.

The names of keysyms eligible for use in translations in this way begin with the prefix "osf" and are referred to as *osf keysyms*. Motif maintains a mapping between these "virtual" keysyms and the "actual" keysyms that correspond to keys on a particular keyboard. When Xt receives a keyboard event, the function **XmTranslateKey** translates the keycode of the event to the appropriate osf keysym if a mapping exists for that keysym. Xt then dispatches the event to the appropriate action routine if a translation exists for that osf keysym.

The mapping between osf and actual keysyms is determined at application startup based on information obtained from one of the following sources, listed in order of precedence:

- A **defaultVirtualBindings** application resource in the resource database.
- A property on the root window, which can be set by **mwm** on startup, by the **xmbind** client, or on prior startup of a Motif application.
- A **.motifbind** file in the user's home directory.
- A default binding based on the vendor string and optionally the vendor release of the X server. Motif searches the file **xmbind.alias** in the user's home directory, the

directory specified by the environment variable **XMBINDDIR**, or the directory **/usr/lib/Xm/bindings**.

The file **xmbind.alias** maps combinations of vendor strings and vendor release numbers to pathnames. Each pathname represents a file that contains keysym bindings for a particular vendor string and optional vendor release number. If Motif fails to find a bindings file for the current display, it uses a set of hard-coded fallback bindings.

The format of the **defaultVirtualBindings** resource is similar to that of a string specifying translations. Each binding consists of an osf keysym, a colon, a list of key event descriptions (with optional modifiers) for actual keysyms, and "\n". Use a comma to separate multiple key event descriptions. The format of a **.motifbind** file or a file containing vendor bindings is the same, except that each binding is on a separate line.

Following is an example of a specification for the **defaultVirtualBindings** resource in a resource file:

```
*defaultVirtualBindings: \
        osfBackSpace:       <Key>BackSpace       \n\
        osfInsert:       <Key>InsertChar      \n\
...
        osfDelete:       <Key>DeleteChar
```

The example specification above appears as follows in a **.motifbind** or vendor bindings file:

```
osfBackSpace:       <Key>BackSpace
osfInsert:       <Key>InsertChar
...
osfDelete:       <Key>DeleteChar
```

For more information, see the **VirtualBindings**(3) and **xmbind**(1) reference pages in the *Motif 2.1—Programmer's Reference*.

13.4 Mnemonics and Accelerators

Sometimes it is desirable for an event received by one widget to activate an action in another. For example, the application may establish a shortcut for activating a button in a menu; the user can activate the menu item even when focus is not in the menu. Motif has two facilities, mnemonics and accelerators, for allowing events in one widget to invoke actions in another.

A mnemonic is a keysym that identifies a key the user can press to activate a menu item when the menu is posted. A button in a MenuBar, PulldownMenu, or PopupMenu can have a mnemonic. When the button is in a PulldownMenu or PopupMenu that is the most recently posted menu, the user activates the button by pressing the key associated with the mnemonic. When the button is in a MenuBar, the MenuBar must have focus for the mnemonic to activate the button. However, the user can activate the button from within the hierarchy that contains the MenuBar, even if the MenuBar does not have focus, by pressing the key while holding the Alt modifier.

An application or user supplies a mnemonic for a button by specifying a value for the Label or LabelGadget resource **XmNmnemonic**. When the button is displayed, Motif underlines the first character in the label string that exactly matches the mnemonic in the character set specified by **XmNmnemonicCharSet**. Although the mnemonic must match a character in the label string exactly in order to be underlined, the user can activate the mnemonic by pressing either the shifted or the unshifted key.

An accelerator allows the user to activate a menu item when focus is anywhere in the hierarchy containing the menu, even if the menu is not posted. Accelerators are supported only for PushButtons and ToggleButtons (or their gadget equivalents) in PulldownMenus and PopupMenus.

An application or user supplies an accelerator for a button by specifying a value for the Label or LabelGadget resource **XmNaccelerator**. The value is a string in the same format as an event description in a translation table, except that only **KeyPress** events are allowed. Thus, an accelerator can have a modifier like Ctrl or Alt. **XmNacceleratorText** is a compound string that describes the accelerator event, for example, "Ctrl+A". Motif displays the accelerator text to the side of the button’s label string or pixmap.

The following example creates a button with a mnemonic and an accelerator:

```
n = 0;
XtSetArg(args[n], XmNmnemonic, XStringToKeysym("A"); n++;
XtSetArg(args[n], XmNaccelerator, "Ctrl<Key>A"); n++;
XtSetArg(args[n], XmNacceleratorText,
        XmStringCreateLocalized("Ctrl+A"); n++;
```

```
button1 = XmCreatePushButton(file_pane, "Answer", args, n);
```

Motif’s button accelerators and mnemonics are supported only for buttons in certain menus. Xt has a more general facility, also called accelerators, for allowing events in one widget to invoke actions in another.

Xt accelerators are mappings of event descriptions to actions, in the same format as a translation table. An application or user supplies accelerators for a widget as the value of the Core resource **XmNaccelerators**. The accelerators map events to actions of this widget, called the source widget. The application must then install the accelerators on a destination widget, using **XtInstallAccelerators**. This routine takes two arguments: the source widget, whose **XmNaccelerators** resource contains the accelerator table; and the destination widget, where the accelerators are to be installed. When the user produces an event in the destination widget that maps to an accelerator in the table, the event invokes the corresponding action *in the source widget*.

XtInstallAccelerators merges the accelerators with the destination widget’s existing translations (the value of **XmNtranslations**). Accelerators can be merged in either **#augment** mode, the default, or **#override** mode. An accelerator table may begin with an **#augment** directive or a **#override** directive. The **#replace** directive is ignored.

As with translations, accelerators must be in an internal format when they are the value of **XmNaccelerators**. A string-to-accelerator-table converter parses an accelerator table string from a resource file. An application can use **XtParseAcceleratorTable** to compile an accelerator table string explicitly.

Accelerators are often defined for a parent source widget and installed on one or more child destination widgets. The SelectionBox and FileSelectionBox widgets install accelerators, the value of **XmNtextAccelerators**, on their text children. The default accelerators bind osfUp, osfDown, osfBeginLine, osfEndLine, and osfRestore events in the Text widget to SelectionBox or FileSelectionBox actions that select an item in the List and replace the Text widget value with that List item.

13.5 Event Handlers

Many applications can implement their entire input processing by adding procedures to widget callback lists and by adding mnemonics and accelerators for menu buttons. Some applications change translations, accelerators, or actions. More rarely, an application needs finer control over event processing. Such an application can register an event handler with the Xt event dispatcher.

An event handler is a procedure that the Xt event dispatcher calls when the application receives events of one or more types. An event handler procedure is of type *XtEventHandler*. It receives four arguments: the widget for which the event arrived; any client data registered with the event handler; a pointer to the event; and a Boolean return argument telling the Xt dispatch facility whether or not to call the remaining event handlers registered for this event. This argument is initialized to True and should rarely be changed.

An application usually registers an event handler by using the function **XtAddEventHandler**. The arguments are the widget, an event mask, an indication whether or not the hander should be called for nonmaskable events, the procedure itself, and any client data to be passed to the event handler when it is called. The order in which event handlers are called is undefined when more than one handler exists for a given widget and event type. However, if the application registers the event handler by using **XtInsertEventHandler**, it can specify that the procedure is to be called either before or after all currently registered event handlers.

Motif requires an application to provide an event handler if it wants to post a PopupMenu on a button press. The call to **XtAddEventHandler** should specify **ButtonPressMask** as the event mask and the popup RowColumn as the client data. The event handler should use **XmMenuPosition** to position the menu at the x and y location of the button press event. It should then manage the RowColumn. If the button press matches the event specified by the RowColumn's **XmNmenuPost** resource, Motif posts the PopupMenu. See Chapter 6 for more information.

13.6 Guidelines for Grabs

Following is a summary of the restrictions on Motif programs regarding the posting and unposting of shell widgets and the use of grabs:

- The functions **XtAddGrab** and **XtRemoveGrab** cannot be used by a Motif application.
- The Intrinsics convenience popup callback functions **XtCallbackNonexclusive** and **XtCallbackExclusive** cannot be used by Motif applications.

- The functions **XtPopup**, **XtPopupSpringLoaded**, and **XtPopdown** can only be used on shell widgets created with the functions **XtCreatePopupShell** or **XtVaCreatePopupShell**. **XtMapWidget** and **XtUnmapWidget** should never be used on shells created in this manner.
- For Motif applications, the value of the second argument of **XtPopup** must always be **XtGrabNone**.
- Never use **XtManageChild** and **XtUnmanageChild** on shell widgets.
- Use **XtMapWidget** and **XtUnmapWidget** only on widgets with the **XmNmappedWhenManaged** resource set to **False**.
- By default, top level shells (those shells created with **XtAppCreateShell** and **XtVaAppCreateShell**) are mapped at the time that they are realized. Generally, only shells of classes *applicationShellWidgetClass* and *topLevelShellWidgetClass* are created in this manner. Except for iconification, these shells are generally expected to remain mapped until they are destroyed or until the application is terminated. (Transient shells should use the "popup" interfaces).
- For unconventional uses of these top level shells, control of posting and unposting can be achieved by creating these shells with the **XmNmappedWhenManaged** resource set to **False**, and then using **XtMapWidget** and **XtUnmapWidget** to post and unpost them. Note that neither the **XtSetValues** nor the **XtSetMappedWhenManaged** interfaces are effective for this resource on shell widgets; these widgets must be created with **XmNmappedWhenManaged** set to **False** in order to use **XtMapWidget** and **XtUnmapWidget**.

Chapter 14
Managing Geometry

The geometry of a widget consists of its size, location, and stacking order. Widgets often have preferred sizes and perhaps locations. For example, a Label widget may prefer to be just large enough to display the text of the label. But composite widgets usually have preferences or constraints in laying out their children, and these may conflict with the preferences of the child widgets. Furthermore, the user or the application can change a widget's geometry at any time; for example, by resizing the top-level window. Geometry management is the process by which the user, parent widgets, and child widgets negotiate the actual sizes and locations of the widgets in the application.

Following are some common occasions for geometry changes:

- The application manages or unmanages a child widget.
- The application sets a geometry resource.
- The application sets a resource that causes one of the geometry resources to change. For example, setting a new label for a Label widget may cause a geometry change.
- The user resizes a top-level window using the window manager.

- The user resizes a pane of a PanedWindow.

Following are the basic Core and RectObj resources that determine widget geometry:

XmNx	Specifies the x coordinate of the upper left outside corner (outside the border) of the widget's window. The value is relative to the upper left inside corner (inside the border) of the parent window.
XmNy	Specifies the y coordinate of the upper left outside corner (outside the border) of the widget's window. The value is relative to the upper left inside corner (inside the border) of the parent window.
XmNwidth	Specifies the inside width (excluding the border) of the widget's window.
XmNheight	Specifies the inside height (excluding the border) of the widget's window.
XmNborderWidth	Specifies the width of the border that surrounds the widget's window on all four sides. Note, however, that you should use resources like **XmNshadowThickness** and **XmNhighlightThickness** instead of **XmNborderWidth** to specify border widths.

14.1 Shells and Their Children

Shell widgets encapsulate application widgets, principally to communicate with the window manager. Motif has three shell classes based on Intrinsics shell classes:

VendorShell	Subclass of **WMShell** and superclass for other shell classes that contain both persistent top-level widgets and dialogs
XmDialogShell	Subclass of **TransientShell** (which is a subclass of **VendorShell**) used to contain dialog widgets, commonly subclasses of **XmBulletinBoard**
XmMenuShell	Subclass of **OverrideShell** used to contain RowColumn PulldownMenu and PopupMenu widgets

A shell has only one managed child. Except when a shell contains an off-the-spot input method, the shell's window is coincident with the child's window. The **geometry_manager** procedures of the shell classes treat geometry requests from the child as geometry requests for the shell, and the *resize* procedures of the shell classes make the child the same size as the shell. Applications should usually change the geometry of the child, not of the shell.

In particular, setting **XmNheight**, **XmNwidth**, or **XmNborderWidth** for either a shell or its child sets that resource to the same value in both the parent and the child. For a child of a shell, setting **XmNx** or **XmNy** sets the corresponding resource of the parent but does not change the child's position relative to the parent. **XtGetValues** for the child's **XmNx** or **XmNy** yields the value of the corresponding resource in the parent. The x and y coordinates of the child's upper left outside corner relative to the parent's upper left inside corner are both zero minus the value of **XmNborderWidth**.

The exception is a VendorShell or DialogShell that contains an off-the-spot input method. In this case, the input method appears inside the shell and below the application widget. The conventions for geometry parameters are the same as for other shells, except that the values of **XmNheight** for the child and the shell are not identical. The height of the shell is the sum of the height and border width of the application window and the height of the area occupied by the input method.

When the Shell resource **XmNallowShellResize** is False, a shell's **geometry_manager** procedure returns **XtGeometryNo** for all geometry requests from a realized child.

14.2 Manager Widgets and Their Children

Each Primitive widget has resources that determine its layout or contents. For example, the size of a Text widget depends on the values of the **XmNrows**, **XmNcolumns**, **XmNmarginHeight**, and **XmNmarginWidth** Text resources; the **XmNhighlightThickness** and **XmNshadowThickness** Primitive resources; and the basic Core geometry resources. In addition, when the Text **XmNresizeHeight** or **XmNresizeWidth** resource is True, the size of the widget can depend on the size of the text (the **XmNvalue** resource). Setting any of these resources can cause Text to generate a geometry request.

Manager widgets have their own layout policies, which they use in responding to geometry requests from their children or to resizing by their parents. These policies are determined by the Manager's own resources and, for some Managers, by its constraint resources.

Constraints are resources defined by the Manager but associated with each child. An application or user initializes, sets, or gets constraint resources for the child as if they were resources defined by the child's class. Initialization, **XtSetValues**, and **XtGetValues** for the child operate on the parent's constraint resources associated with that child. The Manager has constraint *initialize* and **set_values** procedures that allow it to set other constraints and recompute its layout.

Motif uses constraints in determining the layout of Form, PanedWindow, and Frame widgets. Motif also uses constraints to adjust the positions of child widgets in PanedWindow and RowColumn. The Form widget is discussed in Section 14.5. PanedWindow and Frame are discussed in Chapter 8.

14.3 Managing Geometry Using RowColumn

In addition to its role as the menu widget, RowColumn provides general-purpose layout and geometry management for child widgets arranged in rows, columns, or grids. The default RowColumn type, **XmWORK_AREA**, provides the layout features but not the menu semantics.

RowColumn's layout is controlled by two sets of resources. One set determines the position of children within the parent. The other set specifies whether RowColumn adjusts the internal layout characteristics of the children, such as margins and text alignment.

The two primary resources that control child positioning are **XmNorientation** and **XmNpacking**. **XmNorientation** determines whether RowColumn lays out its children in rows or columns. When **XmNorientation** is **XmVERTICAL**—the default for a WorkArea—the layout is column-major. When **XmNorientation** is **XmHORIZONTAL** the layout is row-major.

XmNpacking controls the general style of the layout. The resource has three possible values:

XmPACK_TIGHT

RowColumn places children one after the other along the major dimension (for example, in a column when **XmNorientation** is **XmVERTICAL**). It proceeds until no more children fit along that dimension and then begins a new row or column. When

XmNorientation is **XmVERTICAL** and the vertical distance remaining in the current column is too small to accommodate the child being placed, RowColumn begins a new column if **XmNresizeHeight** is False or the RowColumn cannot become larger. When placing children in a column, RowColumn does not alter their heights, but it makes the width of each child in the column equal to the width of the widest child in that column. Analogous rules apply to row-major layouts. **XmPACK_TIGHT** is the default value for **XmNpacking** in a WorkArea.

XmPACK_COLUMN
RowColumn makes the width and height of each child identical. The width is the maximum width of all children, and the height is the maximum height. RowColumn uses the value of **XmNnumColumns** to determine the maximum number of columns (in **XmVERTICAL** orientation) or rows (in **XmHORIZONTAL** orientation) to produce. RowColumn tries to create **XmNnumColumns** columns (or rows) with an equal number of children in each column (or row).

XmPACK_NONE
RowColumn does not change the position of any child. Unless **XmNresizeWidth** is False, it tries to grow large enough to enclose the greatest x extent of any child. Unless **XmNresizeHeight** is False, it tries to grow large enough to enclose the greatest y extent of any child.

Several other resources influence the position and size of children:

XmNadjustLast
This resource applies only when **XmNpacking** is **XmPACK_TIGHT** or **XmPACK_COLUMN**. When this resource is True and the orientation is vertical, RowColumn increases the widths of children in the last column when necessary so that all children extend to the right edge of the RowColumn. When this resource is True and the orientation is horizontal, RowColumn increases the heights of children in the last row when necessary so that all children extend to the bottom edge of the RowColumn.

XmNentryBorder
When this resource is nonzero, it specifies the border width for all children of the RowColumn. When this resource is zero, RowColumn does not alter the border width of its children.

XmNmarginHeight
: This resource specifies the amount of space between the top edge of the RowColumn and the first item in each column, and between the bottom edge of the RowColumn and the last item in each column.

XmNmarginWidth
: This resource specifies the amount of space between the left edge of the RowColumn and the first item in each row, and between the right edge of the RowColumn and the last item in each row.

XmNresizeHeight
: When this resource is True, RowColumn adjusts its own height when possible to accommodate its children. When this resource is False, RowColumn does not request a new height during layout.

XmNresizeWidth
: When this resource is True, RowColumn adjusts its own width when possible to accommodate its children. When this resource is False, RowColumn does not request a new width during layout.

XmNspacing
: This resource applies only when **XmNpacking** is **XmPACK_TIGHT** or **XmPACK_COLUMN**. It specifies the amount of vertical space between each child in a vertical orientation and the amount of horizontal space between each child in a horizontal orientation.

RowColumn also has several resources that can cause the RowColumn to change the internal layout of some classes of children:

XmNadjustMargin
: This resource applies only to children that are subclasses of **XmLabel** and **XmLabelGadget**. When this resource is True and the orientation is vertical, RowColumn sets the **XmNmarginLeft** and **XmNmarginRight** for all children to the maximum values for those resources among all children. When this resource is True and the orientation is horizontal, RowColumn sets the **XmNmarginTop** and **XmNmarginBottom** for all children to the maximum values for those resources among all children. In PopupMenus and PulldownMenus this resource adjusts the margins only for button children, not for labels.

XmNentryAlignment
: This resource applies only to children that are subclasses of **XmLabel** and **XmLabelGadget**. When **XmNisAligned** is True, RowColumn

sets the **XmNalignment** of all children to the value specified by **XmNentryAlignment**. Following are the possible values:

XmALIGNMENT_BEGINNING
The child's text or pixmap is aligned with the left edge of the child's window.

XmALIGNMENT_CENTER
The child's text or pixmap is aligned with the center of the child's window.

XmALIGNMENT_END
The child's text or pixmap is aligned with the right edge of the child's window.

In menus, RowColumn sets the alignment only for button children, not for labels.

XmNentryVerticalAlignment
This resource applies only to children that are subclasses of **XmLabel**, **XmLabelGadget**, **XmText**, and **XmTextField**. It also applies only when **XmNpacking** is **XmPACK_COLUMN** (in either orientation) or when **XmNpacking** is **XmPACK_TIGHT** and the orientation is horizontal. The value specifies a reference point for aligning the children in any row:

XmALIGNMENT_BASELINE_BOTTOM
Causes the last baseline of each child in a row to align with the last baseline of the tallest child in the row. This value is applicable only when all children in a row contain textual data.

XmALIGNMENT_BASELINE_TOP
Causes the first baseline of each child in a row to align with the first baseline of the tallest child in the row. This value is applicable only when all children in a row contain textual data.

XmALIGNMENT_BOTTOM
Causes the bottom edge of the last line of text contained in each child to align with the bottom edge of the last line of text of the tallest child in the row.

XmALIGNMENT_CENTER
: Causes the center of each child to align vertically with the center point established by the tallest child in the row.

XmALIGNMENT_TOP
: Causes the top edge of the first line of text contained in each child to align with the top edge of the first line of text of the tallest child in the row.

XmNisAligned
: When True, RowColumn sets the **XmNalignment** resources of children that are subclasses of **XmLabel** or **XmLabelGadget** to the value specified by **XmNentryAlignment**.

14.4 Managing Geometry Using BulletinBoard and DrawingArea

BulletinBoard and DrawingArea are two container widgets with similar geometry policies. These widgets have three geometry-related resources in common:

XmNmarginHeight
: Specifies the amount of space between the top shadow of the widget and the top edge of any child, and between the bottom shadow of the widget and the bottom edge of any child. When the value of this resource is greater than 0, the widget ensures that the top edges of all children are below the widget's top margin.

XmNmarginWidth
: Specifies the amount of space between the left shadow of the widget and the left edge of any child, and between the right shadow of the widget and the right edge of any child. When the value of this resource is greater than 0, the widget ensures that the left edges of all children are to the right of the widget's left margin.

XmNresizePolicy
: Determines the widget's policy with regard to resize requests from its children. Following are the possible values:

XmRESIZE_NONE
: The widget has a fixed size determined by its **XmNwidth** and **XmNheight**. The widget does not accept any

geometry requests that would cause it to grow, but it may accept requests (without changing its own size) that would not cause it to grow. The widget also reports its current size as its own preferred size.

XmRESIZE_GROW
The widget can grow but not shrink. If its own parent approves, the widget accepts geometry requests that cause it to grow in order to enclose its children. It may accept requests (without changing its own size) that would not cause it to grow. When queried about its own preferred size, the widget calculates its layout and reports as its preference the greater of the calculated width and height and the current width and height.

XmRESIZE_ANY
The widget tries to accommodate geometry requests that would cause it to grow or shrink in order to enclose its children, requesting changes to its own size when necessary. When queried about its own preferred size, the widget calculates its layout and reports the calculated width and height as its preference.

In addition to these policies, BulletinBoard has geometry facilities that allow it to interact with subclasses in laying out complex collections of children. For example, SelectionBox has a List containing choices, a Text selection area, labels for the list and selection area, and three or four buttons. Usually the list appears above the selection area. The buttons appear equally spaced in a row below the selection area.

Additional children may be added to the SelectionBox after creation. The first child is used as a work area. The value of **XmNchildPlacement** determines if the work area is placed above or below the Text area, or above or below the List area. Additional children are laid out in the following manner:

MenuBar	The first MenuBar child is placed at the top of the window.
Buttons	All **XmArrowButton**, **XmDrawnButton**, **XmPushButton**, and **XmToggleButton** widgets or gadgets, and their subclasses are placed after the OK button in the order of their creation.
Others	The layout of additional children that are not in the above categories is undefined.

14.5 Managing Geometry Using Form

Form is a container widget that provides the most comprehensive facilities for controlling the layout of children. Constraints are placed on children of the Form to define attachments for each of the child's four sides. These attachments can be to the Form, to another child widget or gadget, to a relative position within the Form, or to the initial position of the child. The attachments determine the layout behavior of the Form when resizing occurs. Form is a subclass of BulletinBoard, so the resources and general geometry policies of BulletinBoard apply to Form as well.

Each child has 17 Form constraint resources, four for each side of the child and one, **XmNresizable**, that applies to the child as a whole. Following is a description of **XmNresizable** and the constraint resources that apply to the top side of a child:

XmNresizable

This Boolean resource specifies whether or not a child's request for a new size is (conditionally) granted by the Form. If this resource is set to True, the request is granted if possible. If this resource is set to False, the request is always refused.

If a child has both left and right attachments, its width is completely controlled by the Form, regardless of the value of the child's **XmNresizable** resource. Similarly, if a child has both top and bottom attachments, its height is completely controlled by the Form, regardless of the value of the child's **XmNresizable** resource. If a child has a left or right attachment but not both, the child's **XmNwidth** is used in setting its width if the value of the child's **XmNresizable** resource is True. These conditions are also true for top and bottom attachments, with height acting like width.

XmNtopAttachment

Specifies attachment of the top side of the child. It can have the following values:

XmATTACH_NONE

Do not attach the top side of the child. If **XmNbottomAttachment** is also **XmATTACH_NONE**, this value is ignored and the child is given a default top attachment.

XmATTACH_FORM

Attach the top side of the child to the top side of the Form.

XmATTACH_OPPOSITE_FORM
Attach the top side of the child to the bottom side of the Form. **XmNtopOffset** can be used to determine the visibility of the child.

XmATTACH_WIDGET
Attach the top side of the child to the bottom side of the widget or gadget specified in the **XmNtopWidget** resource. If **XmNtopWidget** is NULL, **XmATTACH_WIDGET** is replaced by **XmATTACH_FORM**, and the child is attached to the top side of the Form.

XmATTACH_OPPOSITE_WIDGET
Attach the top side of the child to the top side of the widget or gadget specified in the **XmNtopWidget** resource.

XmATTACH_POSITION
Attach the top side of the child to a position that is relative to the top side of the Form and in proportion to the height of the Form. This position is determined by the **XmNtopPosition** and **XmNfractionBase** resources.

XmATTACH_SELF
Attach the top side of the child to a position that is proportional to the current y value of the child divided by the height of the Form. This position is determined by the **XmNtopPosition** and **XmNfractionBase** resources. **XmNtopPosition** is set to a value proportional to the current y value of the child divided by the height of the Form.

XmNtopOffset
Specifies the constant offset between the top side of the child and the object to which it is attached. The relationship established remains, regardless of any resizing operations that occur.

XmNtopPosition
This resource is used to determine the position of the top side of the child when the child's **XmNtopAttachment** is set to **XmATTACH_POSITION**. In this case, the position of the top side of the child is relative to the top side of the Form and is a fraction of the height of the Form. This fraction is the value of the

child’s **XmNtopPosition** resource divided by the value of the Form’s **XmNfractionBase**. For example, if the child’s **XmNtopPosition** is 50, the Form’s **XmNfractionBase** is 100, and the Form’s height is 200, the position of the top side of the child is 100.

XmNtopWidget

Specifies the widget or gadget to which the top side of the child is attached. This resource is used if **XmNtopAttachment** is set to either **XmATTACH_WIDGET** or **XmATTACH_OPPOSITE_WIDGET**.

These constraint resources interact with the following resources of the Form itself:

XmNfractionBase

Specifies the denominator used in calculating the relative position of a child widget using **XmATTACH_POSITION** constraints. The value must not be 0.

If the value of a child’s **XmNleftAttachment** (or **XmNrightAttachment**) is **XmATTACH_POSITION**, the position of the left (or right) side of the child is relative to the left side of the Form and is a fraction of the width of the Form. This fraction is the value of the child’s **XmNleftPosition** (or **XmNrightPosition**) resource divided by the value of the Form’s **XmNfractionBase**.

If the value of a child’s **XmNtopAttachment** (or **XmNbottomAttachment**) is **XmATTACH_POSITION**, the position of the top (or bottom) side of the child is relative to the top side of the Form and is a fraction of the height of the Form. This fraction is the value of the child’s **XmNtopPosition** (or **XmNbottomPosition**) resource divided by the value of the Form’s **XmNfractionBase**.

XmNhorizontalSpacing

Specifies the offset for right and left attachments.

XmNrubberPositioning

Indicates the default near (left) and top attachments for a child of the Form.

Note: Whether this resource actually applies to the left or right side of the child and its attachment may depend on the value of the **XmNstringDirection** resource.)

The default left attachment is applied whenever initialization or **XtSetValues** leaves the child without either a left or right attachment. The default top attachment is applied whenever initialization or **XtSetValues** leaves the child without either a top or bottom attachment.

If this Boolean resource is set to False, **XmNleftAttachment** and **XmNtopAttachment** default to **XmATTACH_FORM**, **XmNleftOffset** defaults to the current x value of the left side of the child, and **XmNtopOffset** defaults to the current y value of the child. The effect is to position the child according to its absolute distance from the left or top side of the Form.

If this resource is set to True, **XmNleftAttachment** and **XmNtopAttachment** default to **XmATTACH_POSITION**, **XmNleftPosition** defaults to a value proportional to the current x value of the left side of the child divided by the width of the Form, and **XmNtopPosition** defaults to a value proportional to the current y value of the child divided by the height of the Form. The effect is to position the child relative to the left or top side of the Form and in proportion to the width or height of the Form.

XmNverticalSpacing
Specifies the offset for top and bottom attachments.

Following are some important considerations in using a Form:

- Every child must have an attachment on either the left or the right. If initialization or **XtSetValues** leaves a widget without such an attachment, the result depends upon the value of **XmNrubberPositioning**.

  If **XmNrubberPositioning** is False, the child is given an **XmNleftAttachment** of **XmATTACH_FORM** and an **XmNleftOffset** equal to its current x value.

  If **XmNrubberPositioning** is True, the child is given an **XmNleftAttachment** of **XmATTACH_POSITION** and an **XmNleftPosition** proportional to the current x value divided by the width of the Form.

  In either case, if the child has not been previously given an x value, its x value is taken to be 0, which places the child at the left side of the Form.

- If you want to create a child without any attachments, and then later (for example, after creating and managing it, but before realizing it) give it a right attachment using **XtSetValues**, you must set its **XmNleftAttachment** to **XmATTACH_NONE** at the same time.

- The **XmNresizable** resource controls only whether a geometry request by the child will be granted. It has no effect on whether the child's size can be changed because of changes in geometry of the Form or of other children.
- Every child has a preferred width, based on geometry requests it makes (whether they are granted or not).
- If a child has attachments on both the left and the right sides, its width is completely controlled by the Form. It can be shrunk below its preferred width or enlarged above it, if necessary, due to other constraints. In addition, the child's geometry requests to change its own width may be refused.
- If a child has attachments on only its left or right side, it will always be at its preferred width (if resizable, otherwise at is current width). This may cause it to be clipped by the Form or by other children.
- If a child's left (or right) attachment is set to **XmATTACH_SELF**, its corresponding left (or right) offset is forced to 0. The attachment is then changed to **XmATTACH_POSITION**, with a position that corresponds to the x value of the child's left (or right) edge. To fix the position of a side at a specific x value, use **XmATTACH_FORM** or **XmATTACH_OPPOSITE_FORM** with the x value as the left (or right) offset.
- Unmapping a child has no effect on the Form except that the child is not mapped.
- Unmanaging a child unmaps it. If no other child is attached to it, or if all children attached to it and all children recursively attached to them are also all unmanaged, all of those children are treated as if they did not exist in determining the size of the Form.
- When using **XtSetValues** to change the **XmNx** resource of a child, you must simultaneously set its left attachment to either **XmATTACH_SELF** or **XmATTACH_NONE**. Otherwise, the request is not granted. If **XmNresizable** is False, the request is granted only if the child's size can remain the same.
- A left (or right) attachment of **XmATTACH_WIDGET**, where **XmNleftWidget** (or **XmNrightWidget**) is NULL, acts like an attachment of **XmATTACH_FORM**.
- If an attachment is made to a widget that is not a child of the Form, but an ancestor of the widget is a child of the Form, the attachment is made to the ancestor.

All these considerations are true of top and bottom attachments as well, with top acting like left, bottom acting like right, y acting like x, and height acting like width.

Chapter 15
Graphics and Text in a DrawingArea

Most Motif widgets have specific functions. A PushButton activates an action; a ScrollBar moves a scroll with respect to a viewport; a RowColumn contains a menu, a RadioBox or CheckBox, or a collection of widgets laid out in rows and columns. In contrast, DrawingArea does not have a specific function. It is useful for implementing a canvas, a specialized text editor, or other customized portions of an application.

This chapter takes a detailed look at some of the ways in which you can use DrawingArea in an application. Throughout this chapter, we focus on the **draw.c** example, which is stored online in the **demos/programs/draw** directory.

15.1 DrawingArea: A General-Purpose Widget

DrawingArea is a manager with little specific behavior of its own. It provides basic geometry management for widget and gadget children. It also has callback lists that provide the application with low-level event handling. An application can use these features to implement a canvas or a more specialized widget. Of course, you may also want to write your own widget rather than trying to add specific features to DrawingArea. (See the *Motif 2.1—Widget Writer’s Guide* for details.)

By default, a DrawingArea attempts to adjust its size to contain all its children just inside its margins. The DrawingArea resource **XmNresizePolicy** determines how the DrawingArea responds to geometry requests from its children. This resource has three possible values:

XmRESIZE_ANY
: The DrawingArea tries to accept requests that would cause the DrawingArea to grow or shrink to enclose all its children. This is the default.

XmRESIZE_GROW
: If its parent approves, the DrawingArea accepts requests from its children that would cause the DrawingArea to grow. It may accept requests that would cause it to shrink, but it does not reduce its size.

XmRESIZE_NONE
: The DrawingArea has a fixed size determined by its **XmNheight** and **XmNwidth** resources. It rejects geometry requests from its children that would cause the DrawingArea to grow. It may accept requests that would cause it to shrink, but it does not reduce its size.

The DrawingArea resources **XmNmarginHeight** and **XmNmarginWidth** also affect geometry management. When the value of **XmNmarginHeight** is greater than 0, the DrawingArea ensures that the top edges of all children are inside the top margin. When the value of **XmNmarginWidth** is greater than 0, the DrawingArea ensures that the left edges of all children are inside the left margin.

See Chapter 14 for more information on DrawingArea’s geometry management.

15.2 Event Handling and Callbacks

DrawingArea has callbacks, translations, and actions that inform the application when the DrawingArea is resized or when it receives an exposure event or one of many input events. DrawingArea has the following callbacks:

XmNexposeCallback
DrawingArea invokes these callbacks whenever its *expose* widget class procedure is called. The callback reason is **XmCR_EXPOSE**.

XmNinputCallback
DrawingArea invokes these callbacks from the DrawingAreaInput() action. With the default translations, this action is called when the DrawingArea receives a key press, key release, button press, or button release event. The callback reason is **XmCR_INPUT**.

XmNresizeCallback
DrawingArea invokes these callbacks whenever its *resize* widget class procedure is called. The callback reason is **XmCR_RESIZE**.

Each callback procedure is passed a pointer to an **XmDrawingAreaCallbackStruct**, which includes the reason, the event (NULL for **XmNresizeCallback**), and the DrawingArea's window.

15.2.1 Handling Resize Events

A widget's *resize* procedure is invoked when the widget is resized by its parent or when the widget's width or height changes as a result of **XtSetValues**. DrawingArea also invokes its own *resize* procedure when it has made a successful geometry request of its parent to change its width or height.

For most widgets, the *resize* procedure recomputes the widget's layout to take account of the new size. DrawingArea's *resize* procedure does no layout of its own. It simply invokes the **XmNresizeCallback** callbacks. It is the responsibility of these callback procedures to resize or reposition children or to recompute other contents of the DrawingArea. The callback procedures essentially take the place of the DrawingArea's *resize* procedure.

Note that a *resize* procedure can be called when the widget is not realized.

15.2.1.1 Moving and Resizing Children

An **XmNresizeCallback** procedure should reposition or resize children by calling **XtMoveWidget**, **XtResizeWidget**, or **XtConfigureWidget**. Use of these functions is usually restricted to widget class methods, but for DrawingArea the **XmNresizeCallback** procedures act as part of the widget class *resize* procedure.

A callback procedure could also resize or reposition a child by invoking **XtSetValues** on one or more of the child’s geometry resources (**XmNx**, **XmNy**, **XmNheight**, **XmNwidth**, and **XmNborderWidth**). This causes **XtSetValues** to generate a geometry request on behalf of the child. This request in turn might cause the DrawingArea to make a geometry request of its own parent. In particular, when a child’s request would cause the DrawingArea to change size and when the **XmNresizePolicy** of the DrawingArea is **XmRESIZE_GROW** or **XmRESIZE_ANY**, the DrawingArea is likely to make a geometry request.

However, the Intrinsics forbid a widget’s *resize* procedure from making geometry requests. Therefore, an **XmNresizeCallback** procedure must take care not to reposition or resize a child in such a way that the DrawingArea makes a geometry request. The easiest way to avoid this problem is to use **XtMoveWidget**, **XtResizeWidget**, and **XtConfigureWidget**, which are guaranteed not to make geometry requests.

An **XmNresizeCallback** procedure must take care not to call the *resize* procedure for a child that is in the midst of making a geometry request. This situation can arise when a child makes a geometry request, perhaps as a result of **XtSetValues**, that would cause the DrawingArea to change size. If the DrawingArea’s **geometry_manager** procedure issues a successful geometry request, it invokes its own *resize* procedure, which in turn calls the **XmNresizeCallback** procedures.

When this situation arises, the **XmNresizeCallback** procedure must not call the requesting child’s *resize* procedure, whether it does this directly, as a result of calling **XtResizeWidget** or **XtConfigureWidget**, or as a result of a call to **XtSetValues** that changes the child’s width or height. If an application causes a DrawingArea child to make a geometry request—for example, by calling **XtSetValues** for one of the child’s geometry resources—it should store information in an internal data structure that identifies that child as making a geometry request. The **XmNresizeCallback** procedure should check this information and take care not to call that child’s *resize* procedure.

15.2.1.2 Resizing and Redisplay

A *resize* procedure often recomputes the layout of the widget but does not actually perform the redisplay. In many cases, the act of resizing the widget generates one or more subsequent exposure events, and these in turn cause Xt to invoke the widget's *expose* procedure. In general, the *expose* procedure is responsible for redisplay.

However, resizing a widget does not always generate exposure events, particularly when the widget is made smaller. This is not a problem when the widget's contents consist solely of child widgets or gadgets. The *resize* procedure can reposition or resize the children, and these actions generate the appropriate exposure events for both the children and the parent.

A resizing without an exposure event presents a problem when the contents of the widget include graphics, text, or other decoration outside child widgets. For example, if the widget displays a shadow or other decoration around its inside edge, it must redisplay that decoration when the widget becomes smaller. An application using a DrawingArea in this way must arrange to redisplay the window contents when the DrawingArea becomes smaller. Following are two possible approaches:

- In an **XmNresizeCallback** procedure, compare the DrawingArea's width and height with their previous values. If either width or height has decreased, redisplay the appropriate portions of the DrawingArea's contents. In an internal data structure, store the width and height as the previous width and height for use by the next invocation of the **XmNresizeCallback** procedure.
- In an **XmNexposeCallback** procedure, when the procedure is first invoked, set the window's bit gravity to **ForgetGravity**. This causes the window's contents to be lost and an exposure event to be generated anytime the window is resized. If the application does not set the bit gravity of the DrawingArea's window, the default set by the toolkit is **NorthWestGravity**. This usually causes the server not to generate an exposure event when the window is made smaller.

DrawingArea itself does not draw shadows, and the default **XmNshadowThickness** is 0. It is not practical for an application to draw Motif shadows itself in a DrawingArea, because the Motif shadow-drawing interface is not public. An application that wants shadows with a DrawingArea should place the DrawingArea inside a Frame.

15.2.1.3 Example of a Resize Procedure

The following code from the **DrawCB** callback procedure handles an **XmNresizeCallback**. The procedure spreads or contracts the layout of children and lines in proportion to the increase or decrease in size of the DrawingArea. It uses an internal data structure to hold information about the end points of the lines and the previous width and height of the DrawingArea.

```
static void DrawCB (w, client_data, call_data)
Widget          w;              /*  widget id           */
caddr_t         client_data;    /*  data from application   */
caddr_t         call_data;      /*  data from widget class  */
{

    XmDrawingAreaCallbackStruct * dacs =
        (XmDrawingAreaCallbackStruct *) call_data;
    Arg args[5];
    int n;
    Dimension width, height;
    Graphic * graph = (Graphic *) client_data;

    switch (dacs->reason) {
    ...
    case XmCR_RESIZE:
        n = 0;
        XtSetArg (args[n], XmNwidth, &width);  n++;
        XtSetArg (args[n], XmNheight, &height);  n++;
        XtGetValues (w, args, n);
        ReSize(graph, width, height);
        break;
    ...
```

The **ReSize** method contains the following code:

```
static void ReSize(graph, width, height)
Graphic * graph;
Dimension width, height;
{
 Widget w = graph->work_area;
 Cardinal i,j;
 Arg args[5];
 int n;
 Widget * children;
 Cardinal num_children;
 Position x,y;

    float xratio = (float) width / graph->old_width,
          yratio = (float) height / graph->old_height;

    /* reposition and resize the graphic units */
    for (i=0; i < graph->num_graphics; i++) {
        for (j=0; j < graph->graphics[i].num_points; j++) {
            graph->graphics[i].points[j].x *= xratio;
            graph->graphics[i].points[j].y *= yratio;
        }
    }

    /* reposition the pushbutton children */
    /* I can use XtMoveWidget here since it’s like being part of the
       widget resize class method... */
    n = 0;
    XtSetArg (args[n], XmNnumChildren, &num_children);  n++;
    XtSetArg (args[n], XmNchildren, &children);  n++;
    XtGetValues (w, args, n);
    for (i=0; i < num_children; i++) {
        n = 0;
        XtSetArg (args[n], XmNx, &x);  n++;
        XtSetArg (args[n], XmNy, &y);  n++;
        XtGetValues (children[i], args, n);
        XtMoveWidget(children[i],
                     (Position) (x * xratio),
                     (Position) (y * yratio));
    }

    graph->old_width = width;
    graph->old_height = height;
}
```

15.2.2 Handling Exposure Events

Xt calls a widget's *expose* procedure when the widget receives an exposure event. The precise types of events that cause Xt to invoke the *expose* procedure are determined by the widget class **compress_exposure** field. For **XmDrawingArea**, the value of this field is **XtExposeNoCompress**. This means that Xt invokes the *expose* procedure when the widget receives an **Expose** event.

When the *expose* procedure is called, some part of the contents of the widget's window has been lost, and the window needs to be redisplayed. Xt redisplays the contents of widget children by calling their *expose* procedures. DrawingArea's *expose* procedure calls the **XmNexposeCallback** procedures. These callbacks are responsible for redisplaying any contents of the DrawingArea that are outside the DrawingArea's children. DrawingArea's *expose* procedure then redisplays the contents of gadget children by calling their *expose* procedures.

The X server generates **Expose** events when parts of a window are exposed for a variety of reasons, as when the window is raised or resized. The server determines which portions of the window are exposed and decomposes these into a series of rectangles. The server generates a series of **Expose** events, one for each rectangle.

DrawingArea does not compress exposure events. The *expose* procedure, and therefore the **XmNexposeCallback** list, is called for each rectangle in an exposure series. A simple callback procedure may redisplay the entire window on each exposure series. Such a procedure should examine the *count* member of the *XExposeEvent* structure for the event. A nonzero *count* indicates that more events are to follow in the exposure series. The callback procedure should ignore these events and redisplay the entire window when *count* reaches 0.

A more complex procedure may redisplay only the exposed rectangles. Such a procedure should extract the bounds of each rectangle from the *x*, *y*, *width*, and *height* members of each *XExposeEvent* structure. The procedure can either redisplay each rectangle immediately or accumulate all the rectangles in an exposure series into a region, using **XtAddExposureToRegion**, and then redisplay the region.

An application that draws directly into the DrawingArea must be sure to regenerate the window contents correctly when the DrawingArea becomes smaller. Making the DrawingArea smaller does not always generate **Expose** events. The application can either perform the redisplay in an **XmNresizeCallback** procedure or, on the first invocation of the **XmNexposeCallback** list, set the window's bit gravity to **ForgetGravity**. This ensures that each resizing of the DrawingArea generates an **Expose** event, so the application can safely leave all redisplay to the **XmNexposeCallback** procedure. However, it also means that application must regenerate the entire contents of the window every time the window is resized.

15.2.2.1 Example of an Expose Procedure

The following code from the **DrawCB** callback procedure handles an **XmNexposeCallback**. The first time the procedure is invoked, it sets the window's bit gravity to **ForgetGravity** so that resizing the window generates **Expose** events. It uses an internal data structure to hold information about the end points of the lines.

```
static void DrawCB (w, client_data, call_data)
Widget          w;              /*  widget id           */
caddr_t         client_data;    /*  data from application   */
caddr_t         call_data;      /*  data from widget class  */
{

    XmDrawingAreaCallbackStruct * dacs =
        (XmDrawingAreaCallbackStruct *) call_data;
    XSetWindowAttributes xswa;
    Graphic * graph = (Graphic *) client_data;

    static Boolean first_time = True;

  static Boolean first_time = True;
    switch (dacs->reason) {
    case XmCR_EXPOSE:
        if (first_time) {
            /* Change once the bit gravity of the Drawing Area; default
               is north west and we want forget, so that resize
               always generates exposure events */
            first_time = False;
            xswa.bit_gravity = ForgetGravity;
            XChangeWindowAttributes(XtDisplay(w), XtWindow(w),
                                    CWBitGravity, &xswa);
        }
        ReDraw(graph, dacs->event);
        break;
     ...
```

The **ReDraw** method looks as follows:

```
static void ReDraw(graph, event)
Graphic * graph;
XEvent * event;
{
    Cardinal i;
    Widget w = graph->work_area;

    for (i=0; i < graph->num_graphics; i++) {
        if (graph->graphics[i].type == POLYLINE)
            XDrawLines(XtDisplay(w), XtWindow(w),
                       XDefaultGC(XtDisplay(w),
                       XDefaultScreen(XtDisplay(w))),
                       graph->graphics[i].points,
                       graph->graphics[i].num_points,
                       CoordModeOrigin);
    }
}
```

15.2.3 Handling Input Events

As with any manager, DrawingArea may have three general kinds of input events within its borders:

- Events that belong to a widget child
- Events that belong to a gadget child
- Events that belong to no child

Xt dispatches events to widget children when appropriate, and the DrawingArea does not process these. DrawingArea inherits Manager’s translations for dispatching events to gadget children. Before calling any Manager action as a result of a button press or release or a key press or release, DrawingArea calls its own DrawingAreaInput() action. DrawingArea also calls this action whenever it receives a button press or release or a key press or release that does not have an associated Manager action.

The DrawingAreaInput() action simply returns if the input event is not of type **KeyPress**, **KeyRelease**, **ButtonPress**, **ButtonRelease**, or **MotionNotify**. If the event is of one of these types, and if the event does not take place within a gadget child of the DrawingArea, the action calls the **XmNinputCallback** callbacks.

With the default translations, the result is that the **XmNinputCallback** procedures are invoked whenever the DrawingArea receives a **KeyPress**, **KeyRelease**, **ButtonPress**, or **ButtonRelease** event that does not occur within a child.

The default translations do not invoke the DrawingAreaInput() action, and therefore the **XmNinputCallback** procedures, when the DrawingArea receives a **MotionNotify** event. An application that wants its **XmNinputCallback** procedures invoked on pointer motion events must install the appropriate translations. When installing a translation for **BtnMotion**, the application must override the existing translations. The following translations cause a motion event to be sent to any gadget child in which it takes place. If the event does not take place within a child, the **XmNinputCallback** procedures are invoked:

```
<BtnMotion>:DrawingAreaInput() ManagerGadgetButtonMotion()\n\
<Motion>:DrawingAreaInput()
```

There is one problem with these translations: Because DrawingArea has translations for Btn1 click and double click, the **BtnMotion** actions are not invoked when the user moves the pointer while pressing Btn1. In order to receive these events, the application must replace the DrawingArea translations, omitting the translations for Btn1 click and double click.

15.2.3.1 Example of an Input Procedure

Following is the portion of the **DrawCB** that handles the **XmNinputCallback** procedure. The DrawingArea contains button children and lines connecting them. The procedure takes action on **ButtonPress** and **MotionNotify** events. When the user presses a mouse button, the procedure retrieves the text from a TextField elsewhere in the application. If the user has entered text here, the input procedure creates a PushButton with the text as the label and places it at the point of the click. If the TextField contains no text and the user has pressed a button over a line or PushButton while holding the Shift key, the procedure deletes the line or PushButton.

If the TextField is empty and the user presses a button without holding the Shift key, the procedure either starts or finishes drawing a line. The application uses a rubber-banding effect for line drawing. When it starts a line, the procedure sets a flag indicating it is drawing a line; when it finishes the line, the procedure clears this flag. When the procedure receives a MotionNotify event and is in the process of drawing a line, it erases the previous line (using XOR) and draws a new line from the anchor point to the current pointer position.

```
case XmCR_INPUT:
    if (dacs->event->type == ButtonPress) {
        name = XmTextFieldGetString(graph->textf); /* textfield */
        if (strcmp ("", name) != 0) {
            n = 0;
            XtSetArg (args[n], XmNx, dacs->event->xbutton.x);  n++;
            XtSetArg (args[n], XmNy, dacs->event->xbutton.y);  n++;
            newpush = XmCreatePushButton(w, name, args, n);
            XtAddCallback (newpush, XmNactivateCallback, PushCB, NULL);
            XtManageChild (newpush);
        } else
        if ((dacs->event->xbutton.state & ShiftMask) &&
            (!graph->in_drag)) {
            DeleteUnit (graph, dacs->event->xbutton.x,
                        dacs->event->xbutton.y);
        } else {
            if (!graph->in_drag) {
                StartUnit(graph, dacs->event->xbutton.x,
                          dacs->event->xbutton.y);
            } else {
                EndUnit(graph, dacs->event->xbutton.x,
                        dacs->event->xbutton.y);
            }
        }
        XtFree(name);
    } else  /* need to get motion events here: app_default should
               modified DrawingArea translation with both Motion
               and BtnMotion addition */
    if (dacs->event->type == MotionNotify) {
        /* this one just exits if in_drag is False */
        DragUnit(graph, dacs->event->xbutton.x,
                 dacs->event->xbutton.y);
    }
    break;
}
```

15.3 Using a DrawingArea in a ScrolledWindow

The ScrolledWindow widget provides a viewport onto a virtual scroll and allows the user to move the scroll with respect to the viewport by manipulating ScrollBars. ScrolledWindow offers two scrolling policies: automatic and application-defined. In automatic scrolling, the application provides the scroll widget; ScrolledWindow creates a fixed-size viewport and handles user interaction with the ScrollBars. In application-defined scrolling, the application provides the scroll widget and, if necessary, the viewport, and it handles all user interaction with the ScrollBars.

When using separate viewport and scroll widgets with either scrolling policy, an application can use a default DrawingArea as the scroll widget. When the **XmNresizePolicy** is **XmRESIZE_ANY**, the application can use **XtSetValues** of **XmNx** and **XmNy** to place children within the DrawingArea. The DrawingArea adjusts its size as necessary to enclose all the children. The application can also use **XtSetValues** of the DrawingArea's **XmNwidth** and **XmNheight** to change the size of the scroll widget.

An application can also use a DrawingArea as the viewport widget in application-defined scrolling. One approach is not to use a separate scroll widget but to maintain a virtual scroll, keeping the contents in internal data structures and displaying as much of the contents as will fit into the viewport. The application can use a default DrawingArea as the viewport widget.

Another approach to application-defined scrolling is to create one widget as a viewport and another, a child of the viewport, as the scroll. The application can expand the scroll widget as necessary to contain all the data. In response to user manipulation of the ScrollBars, the application can reposition the scroll widget with respect to the viewport. The viewport acts as a clipping region for its child, the scroll.

In this approach the application can use a DrawingArea as the viewport, the scroll widget, or both. When using a DrawingArea as the viewport, the application must position and resize the scroll child by using **XtMoveWidget**, **XtResizeWidget**, or **XtConfigureWidget**. Using **XtSetValues** for the child's geometry resources does not work because the parent's geometry manager does not permit the child to move or grow beyond the bounds of the parent.

When a DrawingArea is the viewport widget in a ScrolledWindow with application-defined scrolling, the **XmNresizeCallback** procedure must recompute the ScrollBars' **XmNsliderSize** and **XmNpageIncrement** and possibly other resources to reflect the new relation between the viewport and the scroll. It may also need to reposition and resize the scroll with respect to the viewport.

See Chapter 8 for more information on ScrolledWindow, including examples using DrawingAreas as scrolls in both automatic and application-defined scrolling.

15.4 Using a DrawingArea for Graphics

DrawingArea is an appropriate widget to use as a canvas or as a manager that requires graphics operations in addition to children. An application can use Xlib graphics facilities to draw into a DrawingArea. See *Xlib—C Language X Interface* for more information on Xlib graphics operations.

An interactive graphics application can use the **XmNinputCallback** procedure to respond to user input. For example, when the user presses a mouse button, drags, and then releases the button, this procedure might draw a line from the point of the button press to the point of the button release. The **XmNinputCallback** procedures are invoked on button press and release events and on key press and release events. To receive pointer motion events, the application can provide translations that invoke the DrawingAreaInput() action.

An application that needs to produce graphics but does not require children or interaction with the user in the canvas might use a DrawnButton instead of a DrawingArea. DrawnButton has no input callbacks, but it does provide exposure and resize callbacks.

Following is some of the drawing code from the **draw.c** demo. This example implements the rubber-band effect in which a line starts at an anchor point and follows the pointer as the user moves it.

The example maintains an internal data structure with information about the DrawingArea and its graphic objects. The application initially stores a GC for use in drawing and erasing the rubber-band lines. This GC uses a foreground color that results from combining the DrawingArea’s foreground and background using XOR. The GC also uses the **GXxor** function.

The remainder of the example code updates the internal data structures and draws lines as appropriate when the user starts a line, moves the pointer, and ends a line.

```
/* Initialize data structures */
static void InitDraw(graph, app_data)
Graphic * graph;
ApplicationData * app_data;
{
    XGCValues val;
    Arg args[5];
    int n;
    Cardinal i;
    Dimension width, height;
    String pstr, wstr;
    int x, y;
    Widget newpush;

    /* create the gc used for the rudder banding effect */
    n = 0;
    XtSetArg (args[n], XmNforeground, &val.foreground);  n++;
    XtSetArg (args[n], XmNbackground, &val.background);  n++;
    XtGetValues (graph->work_area, args, n);

    val.foreground = val.foreground ^ val.background;
    val.function = GXxor;
    graph->drag_gc = XtGetGC(graph->work_area,
             GCForeground | GCBackground | GCFunction, &val);

    /* initialize the graphic stuff */
    graph->in_drag = False;

    graph->num_graphics = 0;
    for (i=0; i < MAX_GRAPH; i++) {
        graph->graphics[i].num_points = 0;
    }
```

```
/* polylines syntax:
   draw.lines: 10_10, 20_30, 28_139. 11_112, 145_60. 211_112, 45_60
*/
wstr = XtNewString(app_data->lines);
for(pstr = (char *) strtok(wstr, ".,"); pstr;
    pstr = (char *) strtok( NULL, ".,")) {
    while (*pstr && isspace(*pstr)) pstr++;
    if (*pstr == ’ ’) break;

    sscanf(pstr, "%d_%d", &x, &y);
    graph->graphics[graph->num_graphics].points
        [graph->graphics[graph->num_graphics].num_points].x = x;
    graph->graphics[graph->num_graphics].points
        [graph->graphics[graph->num_graphics].num_points].y = y;
    graph->graphics[graph->num_graphics].num_points ++;
    graph->graphics[graph->num_graphics].type = POLYLINE;

    /* look in the original to see if it is a new unit */
    if (app_data->lines[pstr - wstr + strlen(pstr)] == ’.’)
        graph->num_graphics ++;
}
XtFree(wstr);

if (strlen(app_data->lines)) graph->num_graphics ++;

/* Towns syntax:
     draw.towns: Boston, Woburn, SanJose
     draw*Boston.x: 30
     draw*Boston.y: 30
     draw*Woburn.x: 130
     draw*Woburn.y: 30
     draw*SanJose.x: 30
     draw*SanJose.y: 130
*/
wstr = XtNewString(app_data->towns);
for(pstr = (char *) strtok(wstr, ".,"); pstr;
    pstr = (char *) strtok( NULL, ".,")) {
    while (*pstr && isspace(*pstr)) pstr++;
    if (*pstr == ’ ’) break;
    newpush = XmCreatePushButton(graph->work_area, pstr, NULL, 0);
```

```
        XtAddCallback (newpush, XmNactivateCallback, PushCB, NULL);
        XtManageChild (newpush);
    }
    XtFree(wstr);
}

static void StartUnit(graph, x, y)
Graphic * graph;
Position x, y;
{
    Widget w = graph->work_area;

    graph->drag_point.x = graph->anchor_point.x = x;
    graph->drag_point.y = graph->anchor_point.y = y;
    graph->in_drag = True;
    XDrawLine(XtDisplay(w), XtWindow(w),
              graph->drag_gc,
              graph->anchor_point.x, graph->anchor_point.y,
              graph->drag_point.x, graph->drag_point.y);
}

static void DragUnit(graph, x, y)
Graphic * graph;
Position x, y;
{
    Widget w = graph->work_area;

    if (!graph->in_drag) return;

    XDrawLine(XtDisplay(w), XtWindow(w),
              graph->drag_gc,
              graph->anchor_point.x, graph->anchor_point.y,
              graph->drag_point.x, graph->drag_point.y);

    graph->drag_point.x = x;
    graph->drag_point.y = y;

    XDrawLine(XtDisplay(w), XtWindow(w),
              graph->drag_gc,
              graph->anchor_point.x, graph->anchor_point.y,
              graph->drag_point.x, graph->drag_point.y);
}


static Boolean NearPoint (point, x, y)
XPoint point;
Position x, y;
{
#define ERROR 5
    if ((point.x > x - ERROR) &&
        (point.x < x + ERROR) &&
        (point.y > y - ERROR) &&
        (point.y < y + ERROR)) return True;
    else return False;
}


static void EndUnit(graph, x, y)
Graphic * graph;
Position x, y;
{
    Widget w = graph->work_area;
    Cardinal num_points;

    /* no matter what happens, we need to remove
     * the current rubber band */
    XDrawLine(XtDisplay(w), XtWindow(w),
              graph->drag_gc,
              graph->anchor_point.x, graph->anchor_point.y,
              graph->drag_point.x, graph->drag_point.y);

    /* if the given point if the same as the anchor, we’re done with
       this polyline, exit drag mode and be ready for the next
       graphic unit, i.e increment num_graphics */

    if (NearPoint(graph->anchor_point, x, y)) {
        graph->in_drag = False;
        /* now see if a new unit needs to be created */
        if (graph->graphics[graph->num_graphics].num_points) {
            graph->graphics[graph->num_graphics].type = POLYLINE;
            if (graph->num_graphics < MAX_GRAPH)
              graph->num_graphics ++;
            else
              printf("The graphic buffer is full,
                                             overwrite the last...\n");
        }
    } else {

        /* draw the real line and store it in the structure */
        XDrawLine(XtDisplay(w), XtWindow(w),
          XDefaultGC(XtDisplay(w),XDefaultScreen(XtDisplay(w))),
          graph->anchor_point.x, graph->anchor_point.y,
          x, y);

        /* first point in a unit is actually special */
        num_points = graph->graphics[graph->num_graphics].num_points;
        if (num_points == 0) {
            graph->graphics[graph->num_graphics].points[num_points].x =
                graph->anchor_point.x;
            graph->graphics[graph->num_graphics].points[num_points].y =
                graph->anchor_point.y;
            graph->graphics[graph->num_graphics].num_points ++;
            num_points ++;
        }
        graph->graphics[graph->num_graphics].points[num_points].x = x;
        graph->graphics[graph->num_graphics].points[num_points].y = y;
        if (graph->graphics[graph->num_graphics].num_points < MAX_POINT)
            graph->graphics[graph->num_graphics].num_points ++;
        else printf("The unit buffer is full, overwrite the last...\n");

        /* now start the new unit */
        graph->drag_point.x = graph->anchor_point.x = x;
        graph->drag_point.y = graph->anchor_point.y = y;
        XDrawLine(XtDisplay(w), XtWindow(w),
                  graph->drag_gc,
                  graph->anchor_point.x, graph->anchor_point.y,
                  graph->drag_point.x, graph->drag_point.y);
    }
}
```

```
static void DeleteUnit(graph, x, y)
Graphic * graph;
Position x, y;
{
    Widget w = graph->work_area;
    Cardinal i,j;
    int a = -1;

    /* try to find a unit under this point */
    for (i=0; (i < graph->num_graphics) && (a == -1); i++) {
        for (j=0; j < graph->graphics[i].num_points; j++) {
            if (NearPoint(graph->graphics[i].points[j], x, y)) {
                a = i;
                break;
            }
        }
    }

    if (a != -1) {
        /* found a unit under the current point, delete and redisplay */
        for (i = a; i < graph->num_graphics; i++) {
            graph->graphics[i] = graph->graphics[i+1];
        }
        graph->num_graphics --;

        XClearArea(XtDisplay(w), XtWindow(w),
                   0, 0, graph->old_width, graph->old_height, True);
    }
}
```

15.5 DrawingArea and Advanced Text Editing

Some applications may need text-editing capabilities beyond those provided by the Motif Text widget. For example, the application may want to display text using different fonts or colors within the same editor. Such an application might use a DrawingArea to implement a text editor based on compound strings.

15.5.1 Text Output

An application that uses compound strings can use **XmStringDraw** or **XmStringDrawImage** to display the compound string text in a DrawingArea. These functions use different Xlib routines to display compound string segments, depending on whether the segments are associated with font sets or font structs in the font list. **XmStringDraw** uses **XmbDrawString** to display segments associated with font sets. It uses **XDrawString** or **XDrawString16** to display segments associated with font structs. **XmStringDrawImage** uses **XmbDrawImageString** to display segments associated with font sets. It uses **XDrawImageString** or **XDrawImageString16** to display segments associated with font structs.

An application that does not use compound strings may call the Xlib text-drawing routines directly. In addition to those mentioned previously, these include **XDrawText** for text associated with a font and **XmbDrawText** for text associated with a font set. Wide-character versions exist for all the *Xmb* routines.

An application that draws text must determine where to place the text, what the width and height of the text will be, and how to move to the origin of the next text it will draw. For compound strings, an application can use **XmStringExtent**, **XmStringHeight**, **XmStringWidth**, and **XmStringBaseline** to determine the extents of the text.

An application that does not use compound strings may call Xlib routines. To determine the extents of a font struct, the application can examine the *ascent*, *descent*, **max_bounds**, and **min_bounds** members of the *XFontStruct*. To determine the width and extents of text, the application can call *XStringWidth*, **XTextExtents**, and **XTextExtents16**.

To determine the extents of a font set, the application can call *XExtentsOfFontSet*. To determine the width and extents of text, the application can call **XmbTextEscapement**, **XmbTextExtents**, and **XmbTextPerCharExtents**. Wide-character versions exist for all the *Xmb* routines.

For more information about the Xlib text facilities, see *Xlib—C Language X Interface*.

15.5.2 Text Input

To obtain text input in a DrawingArea, an application should use the Xlib input method facilities. These facilities allow the application to open an input method and an input context and to obtain input from the input method. For more information, see Chapter 11 and *Xlib—C Language X Interface*.

Chapter 16

Data Transfer with UTM

Motif permits experienced application programmers to extend or override the data transfer capabilities of many Motif widgets. However, data transfer is typically the responsibility of widgets rather than applications. For this reason, complete details of UTM data transfer are found in the *Motif 2.1—Widget Writer's Guide*. The UTM information in that book is a prerequisite for understanding this chapter. Nevertheless, we do begin this chapter with a brief overview of UTM from the application programmer's perspective. After this overview, the chapter suggests a few relatively straightforward ways in which an application might use UTM.

16.1 What is UTM?

Motif widgets support none, some, or all of the following data transfer mechanisms:

- Primary transfer
- Secondary transfer
- Clipboard transfer

- Drag and drop

The Uniform Transfer Method (UTM) is not the fifth mechanism. Rather, UTM unifies and simplifies the way in which the four mechanisms are implemented. For example, consider an application that uses UTM to implement primary transfer. To add support for a UTM clipboard transfer, the application programmer need only write a little extra code. That is because UTM makes it easy for the different data transfer mechanisms to share code.

As of Motif Release 2.0, all widgets and applications that implement data transfer should use UTM for that implementation. As of Motif Release 2.0, the Motif toolkit implements data transfer exclusively through UTM.

16.2 Why Use UTM?

Most applications will be satisfied with the supported target lists. However, some applications may want to support additional targets. In fact, this is the primary reason for using UTM.

Consider the **XmDrawingArea** widget. This widget does not know how to convert any targets. Therefore, the standard **XmDrawingArea** cannot serve as a useful source for data transfer. However, **XmDrawingArea** does provide a UTM infrastructure. This infrastructure allows an application programmer to use UTM to supplement the data transfer capabilities of **XmDrawingArea**. For example, an application could supplement **XmDrawingArea** such that a user could copy text from an **XmText** widget to an **XmDrawingArea** widget. Similarly, an application could supplement **XmDrawingArea** such that a user could copy a pixmap from one **XmDrawingArea** to another.

16.3 Ideas For Applications

Your application can use two new callbacks, **XmNconvertCallback** and **XmNdestinationCallback**, to extend or override the data transfer capabilities of any Motif widget. In order to override the data transfer capability of a widget, an application has to provide the same data transfer code that a widget would. So, if those are your intentions, we refer you back to the *Motif 2.1—Widget Writer's Guide* for complete details.

If, however, your intention is to extend the data transfer capabilities of a widget, then the section you are now reading should be helpful. The following suggest several ways in which an application could extend the data transfer capabilities of a widget:

- Your application can extend the list of targets supported by a widget. For example, perhaps your application can support a target called **_FORMATTED_TEXT**, which transfers text in some kind of justified formatted.
- Your application can extend the widget to support a new form of data transfer. For example, perhaps your application could extend a particular widget so that it supports Drag and Drop.
- Your application can provide a customized data transfer that improves runtime performance. For example, Motif toolkit widgets typically transfer only one target at a time. If your application always needs to transfer the same group of three targets, your application could request a special multiple transfer.

Before undertaking a UTM project in your application, you should consider the following:

- Data transfer code is relatively complicated to write.
- By extending a widget, you may run into some consistency problems. For example, suppose your application extends the data transfer capabilities of an **XmLabel** widget. Since other applications probably will not extend **XmLabel** in the same way yours has, you run the risk of confusing users. Your application's documentation will have to explain the ways in which **XmLabel** widgets in your application are different from standard **XmLabel** widgets.

16.4 Implementation Overview

This section provides an overview of UTM from an application programmer's perspective. For complete details, see the *Motif 2.1—Widget Writer's Guide*. This section explores the following three kinds of routines that an application might write in order to implement a UTM data transfer:

- An **XmNconvertCallback** procedure associated with the source widget

- An **XmNdestinationCallback** procedure associated with the destination widget
- A transfer procedure associated with the **XmNdestinationCallback** procedure

In brief, an **XmNdestinationCallback** procedure requests specific targets. An **XmNconvertCallback** procedure converts selections to the requested targets. A transfer procedure pastes the converted selections into the destination widget.

16.4.1 XmNconvertCallback Procedure

In order to be the source of a UTM data transfer, a widget must provide a callback resource named **XmNconvertCallback**. The following Motif toolkit widgets hold this resource:

- **XmPrimitive** and all its subclasses
- **XmContainer**
- **XmDrawingArea**
- **XmScale**

Your application can attach an **XmNconvertCallback** procedure to any widget holding the **XmNconvertCallback** resource. UTM calls any **XmNconvertCallback** procedures in your application whenever the associated widget is asked to convert a selection. After calling these callbacks, UTM typically calls the widget's own internal conversion routine (the **convertProc** trait method of the *XmQTtransfer* trait)

When UTM calls your application's **XmNconvertCallback** procedures, UTM passes a pointer to a **XmConvertCallbackStruct** as the *client_data* argument. This callback structure 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;
```

reason Indicates why the callback was invoked.

event Points to the *XEvent* that triggered the callback. It can be NULL.

selection Indicates the selection for which conversion is being requested. Possible values are *CLIPBOARD*, *PRIMARY*, *SECONDARY*, and _MOTIF_DROP.

target Indicates the target to convert.

source_data Contains information about the selection source. When the selection is _MOTIF_DROP, *source_data* is the DragContext. Otherwise, *source_data* is NULL.

location_data
Contains information about the location of data to be converted. If the value is NULL, the data to be transferred consists of the widget's current selection. Otherwise, the type and interpretation of the value are specific to the widget class.

flags Indicates the status of the conversion. Following are the possible values:

XmCONVERTING_NONE
This flag is currently unused.

XmCONVERTING_PARTIAL
The target widget was able to be converted, but some data was lost.

XmCONVERTING_SAME
The conversion target is the source of the data to be transferred.

XmCONVERTING_TRANSACT
This flag is currently unused.

parm Contains parameter data for this target. If no parameter data exists, the value is NULL.

When *selection* is *CLIPBOARD* and *target* is _MOTIF_CLIPBOARD_TARGETS or _MOTIF_DEFERRED_CLIPBOARD_TARGETS, the value is the requested operation (**XmCOPY**, **XmMOVE**, or **XmLINK**).

parm_format
Specifies whether the data in *parm* should be viewed as a list of *char*, *short*, or *long* quantities. Possible values are 0 (when *parm* is NULL), 8 (when the data in *parm* should be viewed as a list of *char*s), 16 (when the data in *parm* should be viewed as a list of *short*s), or 32 (when the data in *parm* should be viewed as a list of *long*s). Note that *parm_format* symbolizes a data type, not the number of bits in each list element. For example, on some machines, a *parm_format* of 32 means that the data in *parm* should be viewed as a list of 64-bit quantities, not 32-bit quantities.

parm_length Specifies the number of elements of data in *parm*, where each element has the size specified by *parm_format*. When *parm* is NULL, the value is 0.

parm_type Specifies the parameter type of *parm*.

status An IN/OUT member that specifies the status of the conversion. The initial value is **XmCONVERT_DEFAULT**. The callback procedure can set this member to one of the following values:

XmCONVERT_DEFAULT
This value means that the widget's **convertProc** trait method, if any, is called after the callback procedures return. If the widget's **convertProc** trait method produces any data, it overwrites the data provided by the callback procedures in the *value* member.

XmCONVERT_MERGE
This value means that the widget's **convertProc** trait method, if any, is called after the callback procedures return. If the widget's **convertProc** trait method produces any data, it appends its data to the data provided by the

callback procedures in the *value* member. This value is intended for use with targets that result in lists of data, such as *TARGETS*.

XmCONVERT_DONE
This value means that the callback procedure has successfully finished the conversion. The widget's **convertProc** trait method, if any, is not called after the callback procedures return.

XmCONVERT_REFUSE
This value means that the callback procedure has terminated the conversion process without completing the requested conversion. The widget's **convertProc** trait method, if any, is not called after the callback procedures return.

value An IN/OUT parameter that contains any data that the callback procedure produces as a result of the conversion. The initial value is NULL. If the callback procedure sets this member, it must ensure that the *type*, *format*, and *length* members correspond to the data in *value*. The callback procedure is responsible for allocating memory when it sets this member. The toolkit frees this memory when it is no longer needed.

type An IN/OUT parameter that indicates the type of the data in the *value* member. The initial value is *INTEGER*.

format An IN/OUT parameter that specifies whether the data in *value* should be viewed as a list of *char*, *short*, or *long* quantities. The initial value is 8. The callback procedure can set this member to 8 (for a list of *char*), 16 (for a list of *short*), or 32 (for a list of *long*).

length An IN/OUT member that specifies the number of elements of data in *value*, where each element has the size symbolized by *format*. The initial value is 0.

If your application does not define any **XmNconvertCallback** procedures, UTM automatically calls the widget's **convertProc** trait method. If your application does define at least one **XmNconvertCallback** procedures, UTM conditionally calls the widget's **convertProc** trait method. Here are the conditions:

- If your **XmNconvertCallback** procedures are taking full responsibility for converting the selection to a particular target, then UTM will not call the widget’s **convertProc** trait method. An **XmNconvertCallback** procedure notes that it is taking full responsibility by setting the *status* member of the **XmConvertCallbackStruct** to **XmCONVERT_DONE** or **XmCONVERT_REFUSE**.

- If your **XmNconvertCallback** procedures are not taking full responsibility for converting the selection to a particular target, then UTM will call the widget’s **convertProc** trait method. An **XmNconvertCallback** procedure notes that it is not taking full responsibility by setting the *status* member of the **XmConvertCallbackStruct** to **XmCONVERT_DEFAULT** or **XmCONVERT_MERGE**.

If UTM does call your widget’s **convertProc** trait method, UTM passes it the same **XmConvertCallbackStruct** used by the application’s **XmNconvertCallback** procedures. Thus, the **XmConvertCallbackStruct** provides a convenient conduit for the **XmNconvertCallback** procedures to communicate with the widget’s **convertProc** trait method.

16.4.2 XmNdestinationCallback Procedure

In order to be the destination for a UTM data transfer, a widget must provide a callback resource named **XmNdestinationCallback**. Appendix B lists the widgets in the standard Motif toolkit that hold this callback resource.

Your application can attach an **XmNdestinationCallback** procedure to any widget holding the **XmNdestinationCallback** resource. UTM calls any **XmNdestinationCallback** procedures in your application whenever the associated widget is the destination of a data transfer. After calling these callbacks, UTM typically calls the destination widget’s own internal destination routine (the **destinationProc** trait method of the *XmQTtransfer* trait).

When UTM calls your application’s **XmNdestinationCallback** procedures, UTM passes a pointer to a **XmDestinationCallbackStruct** as the *client_data* argument. This callback structure is defined as follows:

```
typedef struct
{
        int     reason;
        XEvent  *event;
        Atom selection;
        XtEnum operation;
        int flags;
        XtPointer transfer_id;
        XtPointer destination_data;
        XtPointer location_data;
        Time time;
} XmDestinationCallbackStruct;
```

reason Indicates why the callback was invoked.

event Points to the *XEvent* that triggered the callback. It can be NULL.

selection Indicates the selection for which data transfer is being requested. Possible values are *PRIMARY*, *SECONDARY*, *CLIPBOARD*, and _MOTIF_DROP.

operation Indicates the type of transfer operation requested.

- When the *selection* is *PRIMARY* or *SECONDARY*, possible values are **XmMOVE**, **XmCOPY**, and **XmLINK**.
- When the selection is *CLIPBOARD*, possible values are **XmCOPY** and **XmLINK**.
- When the selection is _MOTIF_DROP, possible values are **XmMOVE**, **XmCOPY**, **XmLINK**, and **XmOTHER**. A value of **XmOTHER** means that the callback procedure must get further information from the **XmDropProcCallbackStruct** in the *destination_data* member.

flags Indicates whether or not the destination widget is also the source of the data to be transferred. Following are the possible values:

XmCONVERTING_NONE
The destination widget is not the source of the data to be transferred.

XmCONVERTING_SAME
The destination widget is the source of the data to be transferred.

transfer_id Serves as a unique ID to identify the transfer transaction.

destination_data

Contains information about the destination. When the selection is _MOTIF_DROP, the callback procedures are called by the drop site's **XmNdropProc**, and *destination_data* is a pointer to the **XmDropProcCallbackStruct** passed to the **XmNdropProc** procedure. When the selection is *SECONDARY*, *destination_data* is an Atom representing a target recommended by the selection owner for use in converting the selection. Otherwise, *destination_data* is NULL.

location_data

Contains information about the location where data is to be transferred. The interpretation of this value varies considerably from widget to widget. Once *XmTransferDone* procedures start to be called, **location_data** will no longer be stable.

time Indicates the time when the transfer operation began.

The fields in an **XmConvertCallbackStruct** are far more dynamic than the fields in an **XmDestinationCallbackStruct**. In other words, an **XmNconvertCallback** procedure typically modifies many of the fields of its **XmConvertCallbackStruct**; however, an **XmNdestinationCallback** procedure typically leaves all the fields of its **XmDestinationCallbackStruct** untouched.

A typical **XmNdestinationCallback** procedure requests a list of targets supported by the source widget. To make this request, the procedure calls *XmTransferValue*. For example, here is a typical **XmNdestinationCallback** procedure:

```
void DestinationCallback(w, ignore, cs)
       Widget w;
       XtPointer ignore;
       XmDestinationCallbackStruct *cs;
{
 Atom TARGETS = XInternAtom(XtDisplay(w), "TARGETS", False);

 /* Request TARGETS that the source widget can convert. */
   XmTransferValue(cs -> transfer_id, TARGETS,
                   (XtCallbackProc) TransferProcedure,
                   NULL, XtLastTimestampProcessed() );
}
```

16.4.3 Transfer Procedure

The transfer procedure is a companion routine to the **XmNdestinationCallback** procedure. Typically,

1. The **XmNdestinationCallback** procedure calls *XmTransferValue* to ask for a list of targets supported by the source.
2. When the source completes the conversion, UTM returns control to the transfer procedure.
3. The transfer procedure examines the returned list of targets. If the list contains a desired target, the transfer procedure requests it by calling *XmTransferValue*.

Steps 2 and 3 may repeat several times.

Unlike the **XmNconvertCallback** and **XmNdestinationCallback** routines, the transfer procedure is not identified by a resource. Rather, the transfer procedure is identified by the third argument in a call to *XmTransferValue*.

When UTM calls your application's transfer procedure, UTM passes a pointer to an **XmSelectionCallbackStruct** as the *client_data* argument. This callback structure is defined 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;
```

reason	Indicates why the callback was invoked.
event	Points to the *XEvent* that triggered the callback. It can be NULL.

selection	Specifies the selection that has been converted.
target	Specifies the target to which *XmTransferValue* requested conversion. The value is the same as the value of the *target* argument to *XmTransferValue*.
type	Specifies the type of the selection value. This is not the target, but the type used to represent the target. The value *XT_CONVERT_FAIL* means that the selection owner did not respond to the conversion request within the Intrinsics selection timeout interval.
transfer_id	Specifies a unique indentifier for the data transfer operation. The value is the same as the value of the **transfer_id** argument to *XmTransferValue*.
flags	This member is currently unused. The value is always **XmSELECTION_DEFAULT**.
remaining	Indicates the number of transfers remaining for the operation specified by **transfer_id**.
value	Represents the data transferred by this request. The application is responsible for freeing the value by calling **XtFree**.
length	Indicates the number of elements of data in *value*, where each element has the size symbolized by *format*. If *value* is NULL, *length* is 0.
format	Indicates whether the data in *value* should be viewed as a list of *char*, *short*, or *long* quantities. Possible values are 8 (for a list of *char*), 16 (for a list of *short*), or 32 (for a list of *long*).

16.5 Case Study: Adding an Extra Target to XmText

This section illustrates how an application can use UTM to supplement the list of targets that an **XmText** widget can convert.

Here is the scenario. The application instantiates two **XmText** widgets. We will refer to one of these widgets as **TextSource** and the other as **TextDestination**. **TextDestination** has a rather unusual requirement; namely, it expects that any text transferred to it must not contain any lowercase letters. In other words, the source of a data transfer is responsible for converting any lowercase letters to uppercase before transferring the data.

The standard **XmText** widget does not provide any targets that can do this conversion. Therefore, the application must temporarily extend **XmText** to handle a new target named *MYTEXT*. The application must do the following:

- The application must associate an **XmNconvertCallback** procedure with **TextSource**. This procedure must be able to convert the *MYTEXT* target.
- The application must associate an **XmNdestinationCallback** procedure and a transfer procedure with **TextDestination**. These procedures are responsible for requesting that the selection be converted to *MYTEXT* and for pasting the transferred data into **TextDestination**.

The application will support three different forms of data transfer: primary, clipboard, and drag and drop.

The following subsections explains how the application accomplishes its goals.

16.5.1 The XmNconvertCallback Procedure

The application must associate an **XmNconvertCallback** procedure with widget **TextSource**. The following code does just that:

```
XtAddCallback(TextSource, XmNconvertCallback, ConvertCallback, NULL);
```

The application must provide an **XmNconvertCallback** procedure named **ConvertCallback**. This procedure must handle the following requests:

- The destination could request the list of targets that the source can convert.
- The destination could request that the selection be converted to *MYTEXT*.

Following are the relevant parts of the **ConvertCallback** routine:

```
void
ConvertCallback(Widget  w,
                     XtPointer ignore,
                     XtPointer call_data)
{
 XmConvertCallbackStruct  *ccs = (XmConvertCallbackStruct *)call_data;
 char    *selected_text;
 char    *copy_of_selected_text;
```

```
  Atom TARGETS = XInternAtom(XtDisplay(w), "TARGETS", False);
  Atom _MOTIF_CLIPBOARD_TARGETS = XInternAtom(XtDisplay(w),
                               "_MOTIF_CLIPBOARD_TARGETS", False);
  Atom MYTEXT = XInternAtom(XtDisplay(w), "MYTEXT", False);
  int   n=0;
  Atom *targs = (Atom *)XtMalloc(sizeof(Atom) * 2);

   if ((ccs->target == TARGETS) ||
      (ccs->target == _MOTIF_CLIPBOARD_TARGETS)) {

   /* Use targs to hold a list of targets that my application can
      convert. This list will be merged with the targets that the
      XmText widget can convert. */
     targs[n] = MYTEXT; n++;
     ccs->value = (XtPointer) targs;
     ccs->type = XA_ATOM;
     ccs->format = 32;
     ccs->length = n;
     ccs->status = XmCONVERT_MERGE;
   }
   else if (ccs->target == MYTEXT)  {
   /* Get the selection. */
     selected_text = XmTextGetSelection(w);
     copy_of_selected_text = selected_text;

   /* Convert any lowercase letters in the selection to uppercase. */
     while (*selected_text++)  {
        if (islower(*selected_text))
          *selected_text = toupper(*selected_text);
     }

   /* Place the converted text into the XmConvertCallbackStruct. */
     ccs->value = copy_of_selected_text;
     ccs->type = ccs->target;
     ccs->format = 8;
     ccs->length = strlen(copy_of_selected_text);
     ccs->status = XmCONVERT_DONE;
   }
 }
```

Notice how **ConvertCallback** sets the value of the *status* member of the **XmConvertCallbackStruct**. For example, when the destination requests *TARGETS* or _MOTIF_CLIPBOARD_TARGETS, **ConvertCallback** sets *status* to **XmCONVERT_MERGE**. This *status* tells UTM to call the **convertProc** trait method of **XmText**. This trait method will add *MYTEXT* to the list of supported targets. As another example, consider that **ConvertCallback** is taking full responsibility to convert the selection to *MYTEXT*. For this case, **ConvertCallback** sets *status* to **XmCONVERT_DONE**. This status tells UTM not to bother calling the **convertProc** trait method of **XmText**. Finally, consider what happens when **ConvertCallback** is asked to convert a target other than *TARGETS*, _MOTIF_CLIPBOARD_TARGETS, or *MYTEXT*. In this case, the *status* member will retain its default value of **XmCONVERT_DEFAULT**. This *status* tells UTM to call the **convertProc** trait method of **XmText**, and that the trait method will be reponsible for converting the target.

16.5.2 The XmNdestinationCallback Procedure

The application must associate an **XmNdestinationCallback** procedure with widget **TextDestination**. The following code does just that:

```
XtAddCallback(TextDestination, XmNdestinationCallback,
              DestinationCallback, NULL);
```

The application must provide an **XmNdestinationCallback** procedure named **DestinationCallback**. We have implemented this procedure in a very basic fashion. This procedure merely requests a list of the targets that the source widget supports:

```
void
DestinationCallback(Widget  w,
                        XtPointer ignore,
                        XtPointer call_data)
{
 XmDestinationCallbackStruct *dcs =
           (XmDestinationCallbackStruct *)call_data;
 Atom TARGETS = XInternAtom(XtDisplay(w), "TARGETS", False);

 /* Ask the source to return a list of all the targets supported. */
   XmTransferValue(dcs->transfer_id, TARGETS,
                   (XtCallbackProc)TransferProc,
                   NULL, XtLastTimestampProcessed() );
}
```

16.5.3 The Transfer Procedure

When the source returns the list of *TARGETS*, UTM calls the application's transfer procedure. The transfer procedure, **TransferProc** in this case, is named by the third argument to *XmTransferValue*.

Here is the algorithm for the transfer procedure. The transfer procedure must examine the returned list of targets to see if the list contains *MYTEXT*. If it does, ask the source to convert the selection to *MYTEXT*. When the source completes the conversion, the transfer procedure must paste the converted text into the **TextDestination** widget.

Following is the code for **TransferProc**:

```
void
TransferProc(Widget  w,
             XtPointer ignore,
             XtPointer call_data)
{
 XmSelectionCallbackStruct *scs =
       (XmSelectionCallbackStruct *) call_data;
 Atom TARGETS = XInternAtom(XtDisplay(w), "TARGETS", False);
 Atom MYTEXT = XInternAtom(XtDisplay(w), "MYTEXT", False);
 Atom  *targets = (Atom *)scs->value;
 int    MYTEXT_is_supported = 0;
 unsigned long    n;


   if ((scs->target == TARGETS) && (scs->type == XA_ATOM))  {
     for (n=0; n<=scs->length; n++)  {
      /* Look through list of returned TARGETS to see if
         MYTEXT is there. */
        if (targets[n] == MYTEXT)
          MYTEXT_is_supported = 1;
     }

     if (MYTEXT_is_supported)
     /* Now ask the source widget to convert the selection
          to MYTEXT. */
       XmTransferValue(scs->transfer_id, MYTEXT,
                   (XtCallbackProc)TransferProc,
                   NULL, XtLastTimestampProcessed() );
    }

  if ((scs->target == MYTEXT)) {
    XmTextPosition current_insertion_position;
  /* Source widget has converted MYTEXT, paste it into the
     destination widget. */
    current_insertion_position = XmTextGetInsertionPosition(w);
    XmTextInsert(w, current_insertion_position, (char *)scs->value);
    XmTransferDone(scs->transfer_id, XmTRANSFER_DONE_SUCCEED);
  }
}
```

Notice how **TransferProc** calls *XmTransferDone*. This call marks the end of the transfer. Therefore, UTM will not call the **destinationProc** trait method inside the **TextDestination** widget.

If the source does not know how to convert *MYTEXT*, UTM will call the **destinationProc** trait method inside the **TextDestination** widget.

Chapter 17

Drag and Drop

As of Release 2.0, Motif uses the Uniform Transfer Method (UTM) to implement all drag and drop operations. Any drag and drop applications you write for Release 2.0 should also use UTM. Drag and drop applications written prior to Release 2.0 will work without modification at Release 2.0.

You should read Chapter 16 before reading this chapter. Chapter 16 explains UTM. Drag and drop is merely one kind of UTM transfer. That is, drag and drop uses the same convert and destination callbacks as any other UTM transfer. However, drag and drop supports richer visuals and protocols than other kinds of data transfer, and it is these topics that are the focus of this chapter.

This chapter first provides an overview of the drag and drop process and concepts from both the user's and the application developer's perspectives, then explains the actions of both initiator and receiver clients during the drag and at the drop, giving code samples.

Note: Throughout this chapter, we refer to Btn2 as the button that activates drag and drop operations. Although Btn2 is the traditional Motif drag and drop button, some users may prefer to use Btn1 instead of Btn2 when doing drag and drop operations on Text, TextField, List, and Container widgets. Users choose their preferred drag and drop button with the **XmNenableBtn1Transfer** resource of **XmDisplay**.

17.1 User Overview of Drag and Drop

Drag and drop allows the user to "pick up" objects on the screen, "drag" them around the display, and "drop" them at a new location, possibly in another application.

With drag and drop the user can

- Move text or other information between windows.
- Cause application-specific actions to occur.
- Obtain help information about drop sites.

This section describes what the user does and sees during a drag and drop transaction.

17.1.1 Overview of User Interaction

A drag and drop transaction consists of the following actions:

1. To start a drag transaction, a user typically presses and holds Btn2, over a source object. The application owning that object is the initiator of the drag. The current pointer is replaced by a drag icon—a picture representing the item being dragged.
2. The user moves the pointer. From now until a drop occurs, the drag icon replaces the mouse pointer. The user may move the pointer by moving the mouse, or with the arrow keys.
3. The user drops the object, usually by releasing the mouse button.

   Locations on the screen that can accept drops are drop sites, and the application owning that drop site is the destination or receiver.

   The drag icon can be dropped anywhere on the screen. However, only certain widgets have registered themselves as drop sites and are able to process the drop.

The receiver application usually performs some action on the information represented by the dragged icon. The initiator application may also perform some action based on the results of a drag transaction.

A drop can be between applications or within the same application. An application can be both source and destination of a drop, source only, destination only, or not participate in drag and drop at all.

The user can request help about a drop site, if available, by dragging to the drop site and pressing osfHelp (usually **F1**).

The user can cancel the drag at any time by pressing osfCancel, usually **Escape**.

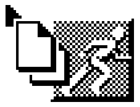
17.1.2 Overview of Drag-Over Effects

The drag icon consists of three parts:

- The source icon is a picture representing the type of the source object, such as text.
- The state icon can be used to show whether or not the object being dragged can be dropped at its current location on the screen.
- The operation icon can be used to show what action should happen when the drop takes place.

In the following illustration, the running figure is the source icon, the arrow in the upper left is the state icon, and the rectangles with the corner folded over indicate a Copy is desired.

Figure 17–1. A Drag Icon



These parts can be combined (blended) and attached to each other in different ways. The default blending and attachment are shown in the previous illustration.

Parts of the drag icon may change shape or color as it is being dragged through potential drop sites, providing visual feedback about possible drop sites to the user. These changes are drag-over effects.

Applications can use the default drag icon effects, or provide more sophisticated or custom drag icons. The application or user can customize these drag-over effects in resource files.

Note: This chapter contains several illustrations of default Motif drag and drop icons. In reality, there are two different sets of default Motif icons. The **XmNenableDragIcon** resource of **XmDisplay** determines which set is used. If **XmNenableDragIcon** is False (the default), the traditional Motif drag and drop icons are used. If **XmNenableDragIcon** is True, alternate Motif drag and drop icons are used. We have used only the traditional Motif drag and drop icons for the illustrations in this chapter.

17.1.2.1 Drag States

During a drag, there are three states that describe the relationship of a drag icon to what is under it at the time:

Valid drop site
: The drag icon is over a drop site on which it can potentially be dropped (this is only a hint; when the drop is actually attempted, further processing may show that the drop cannot actually be done).

Invalid drop site
: The drag icon is over a drop site, but it cannot be dropped there.

No drop site
: The drag icon is not over a registered drop site.

The default state icon for all three states is the same: an arrow in the upper left corner of the drag icon. Because the icon is the same for all three states, it appears not to change during the drag. The application or the user can provide custom state icons or colors in a resource file.

17.1.2.2 Drag Operations

The user specifies what action is to take place when the drop occurs by pressing certain keys when the drag starts or while the drag is in process:

Shift only	Force a move from the initiator to the receiver client (Move)
Ctrl only	Force a copy from the initiator to the receiver client (Copy)
Shift and Ctrl	Force a link between the initiator and receiver clients (Link)

The operation chosen by the user must be valid for both the drag source and the drop site, or the drop site will be considered invalid.

If the user does not specify an operation, one is chosen by the toolkit. It choses an operation that is valid for both the drag source and drop site. Move is the first choice, Copy is the second, and Link is the third. If the system cannot find a valid operation, the drop site is considered invalid.

The operation icon reflects the operation chosen by the user or by the system. If the operation is changed by the user during the drag, the operation icon changes also.

The operation icon may change as the drag icon moves to different drop sites if the drop sites accept different operations.

17.1.3 Overview of Drag-Under Effects

A widget registered as a drop site may change visually as a drag icon passes over it. These visual cues are drag-under effects. The sensitive area of the widget is the part that responds to drag and drop. By default it is the whole widget, but applications can specify that only parts of the widget respond to drag and drop.

In addition, a widget may support *pause drag*. This means that if a drag icon pauses over a specifed part of a widget, some action will occur, such as scrolling. See Section 17.4.2.2 for more information about this feature.

Various highlighting styles are possible:

- A border around the sensitive area of the drop site widget. This is the default value.
- The sensitive area of the drop site widget looks pushed out.
- The sensitive area of the drop site widget looks pushed in.
- A special pixmap is displayed within the sensitive area of the drop site widget, overwriting what is normally there.
- No drag-under effects are used for the drop site widget.

Applications can use the default drag-under visual effects, or create more sophisticated or custom effects, such as special animation or sound effects.

17.1.4 Overview of Drop Effects

Visual effects also take place during the drop:

- The drag icon appears to sit over the drop site while the processing for the drop is finishing, but the standard cursor is restored and can be used normally.
- The source icon appears to melt into the drop site if the drop is successful.
- The source icon appears to snap back to the source if the drop is unsuccessful.
- The source icon appears to snap back to the source and the previous X cursor returns if Cancel is requested. All drag-under and drag-over effects are removed.

These drop effects cannot be changed by the application or the user. However, if the receiver wants to cancel these effects completely, the receiver should ask the source to convert the special _MOTIF_CANCEL_DROP_EFFECT target. For example, the receiver can make a call like the following to cancel the effects:

```
XmTransferValue(cs->transfer_id,
     XInternAtom(XtDisplay(wid), XmS_MOTIF_CANCEL_DROP_EFFECT, False),
     NULL, NULL, XtLastTimeStampProcessed(XtDisplay(wid)));
```

The Motif toolkit will intercept this request and fulfill it by cancelling all visual effects. The drag source should not provide any code to handle this target.

The application or widget should supply a DialogBox containing information about a drop site if the user has requested help and the receiver client provides help.

17.2 Application Programmer’s Overview of Drag and Drop

This section explains some drag and drop concepts, and provides a general view of the initiator and receiver duties during the drag and at the drop.

The Motif toolkit for drag and drop consists of

- Widgets and widget classes that provide resources containing details about the source and destination of the transfer
- Functions that applications use to manage the widgets and widget classes
- Protocols that specify how interactions between source and destination clients are to take place
- Functions that manage messages, call callbacks, decide on the valid operations for a potential drop, and keep the drop site status updated

If the initiator and receiver are in the same client, they share the same toolkit. If the initiator and receiver are different clients, each client has a version of the toolkit.

An application can allow any widget to be a drag source or initiator by specifying a translation for Btn2Down in that widget. The corresponding action creates a DragContext, which starts the drag and drop transaction. The toolkit on the initiator side is in charge during the drag and manages all drag messages and callbacks.

An application can register any widget as a drop site. The drop site widget may change visually as a drag icon moves in and out of it, providing drag-under visual clues to the status of the drag. The application controlling the current drop site is known as the receiver. The toolkit on the receiver side is in charge of the drop operation, and manages all drop messages and callbacks.

Each drag source and drop site specifies the types of data it is prepared to handle and what operations it can perform on that data.

The state of the drag indicates whether the drag icon is over a valid drop site, an invalid drop site, or no drop site. For a drop site to be valid, there must be at least one target type and one operation in common between the drag source and drop site.

17.2.1 Ways an Application Can Provide Drag and Drop

Applications can use drag and drop functionality on any of several levels:

- Appendix B lists the widgets that are already defined as drag sources and as drop sites. Therefore, at the simplest, an application can compile with the Motif libraries, and have those widgets participate in drag and drop. For example, text from a Text widget could be selected from one application and moved into the Text widget in another application.
- On a slightly more advanced level, applications can let the toolkit do most of the work, but provide some customization. For example, an application could register an **XmPushButton** as a drop site, but still use default visual effects. In this case, the application would register a widget as a DropSite and provide code to handle drop and transfer duties. The example programs **simple_drag** and **simple_drop** in the following subsections are at that level.
- A complex application can take much of the control of the drag and drop itself. It can provide custom visuals for both drag icon and drop site. It can manage overlapping drop sites and can include complex transfers of information. The online example program **DNDDemo.c** contains extensive customization.

17.2.2 Extending a Widget To Support Drop

Any UTM widget that provides an **XmNdestinationCallback** resource has the potential to be a drop site. Conversely, if a widget does not provide an **XmNdestinationCallback** resource, it cannot be a UTM drop site.

Of the widgets that provide an **XmNdestinationCallback** resource, many already contain all the required code to become a drop site. That is, many of the standard widgets can serve as drop sites without you, the application programmer, doing anything. However, some of the widgets that do provide an **XmNdestinationCallback** resource do not provide the drop site code. You can extend these widgets to become drop sites. For example, the **XmDrawingArea** widget provides an **XmNdestinationCallback** resource but does not provide any drop site code. You can write an application that makes **XmDrawingArea** into a drop site.

Consider the **simple_drop** sample program stored online in **simple_drop.c**. This program displays both a Label widget and a DrawingArea widget. The Label widget displays whatever pixmap you pass on the invocation command line. For example, if you invoke this application as follows:

```
$ simple_drop mypixmap.xpm
```

then the Label widget will display the contents of the **mypixmap.xpm** file. To experiment with this application, simply drag the pixmap from the Label and drop it into the framed DrawingArea.

The standard Label widget knows how to be a drag source without any help from the application. However, the standard DrawingArea does not know how to be a drop site. Therefore, **simple_drop** provides both of the following procedures in order to add pixmap drop support to the DrawingArea:

- An **XmNdestinationCallback** procedure
- A transfer procedure

The only drop target that our DrawingArea supports is *PIXMAP*. The Label widget in the application provides a convenient *PIXMAP* source. In addition, you can drop a pixmap from any widget on the screen that knows how to convert a *PIXMAP* target.

The *main* routine instantiates the Label and DrawingArea widgets. *main* also calls **XmeDropSink** to establish the DrawingArea as a drop site.

```
int
main(int argc, char **argv)
{
 static Widget  MainWindow;
 XtAppContext   app_context;
 Widget         Frame1, RC1, Label1, DrawingArea1;
 Pixmap         pixmap;
 GC             gc;

   toplevel = XtAppInitialize(&app_context, "Test", NULL, 0,
                              &argc, argv, NULL, NULL, 0);

   MainWindow = XtVaCreateManagedWidget("MainWindow1",
                                   xmMainWindowWidgetClass, toplevel,
                                       NULL);

   CreateMenus(MainWindow);
 RC1 = XtVaCreateManagedWidget("RC1", xmRowColumnWidgetClass,
                               MainWindow, NULL);

 if (!argv[1]) {
      printf("usage: %s bitmap-file\n", *argv);
      exit(1);
 }

 /* 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 instantiate an XmLabel widget that displays pixmap. */
  Label1 = XtVaCreateManagedWidget("Label1",
      xmLabelWidgetClass, RC1,
      XmNlabelType,   XmPIXMAP,
      XmNlabelPixmap, pixmap,
      NULL);

  Frame1 = XtVaCreateManagedWidget("Frame1",
      xmFrameWidgetClass, RC1,
      XmNshadowThickness, 3,
      NULL);

  DrawingArea1 = XtVaCreateManagedWidget("DrawingArea1",
      xmDrawingAreaWidgetClass, Frame1,
      XmNwidth, 150,
      XmNheight, 150,
      XmNresizePolicy, XmRESIZE_NONE,
      NULL);
  XmeDropSink(DrawingArea1, NULL, 0);
  XtAddCallback(DrawingArea1, XmNdestinationCallback,
                DestinationCallback, NULL);
```

```
    XtRealizeWidget(toplevel);
    XtAppMainLoop(app_context);
}
```

When the user attempts a drop on the DrawingArea, UTM will call the **DestinationCallback** routine. This routine will call *XmTransferValue*, asking the source widget (the drag initiator) to convert its list of _MOTIF_EXPORT_TARGETS.

```
void
DestinationCallback(Widget  w,
                    XtPointer ignore,
                    XtPointer call_data)
{
 XmDestinationCallbackStruct *dcs =
           (XmDestinationCallbackStruct *)call_data;
 Atom TARGETS = XInternAtom(XtDisplay(w), "TARGETS", False);
 Atom _MOTIF_EXPORT_TARGETS = XInternAtom(XtDisplay(w),
                             "_MOTIF_EXPORT_TARGETS", False);

 /* Ask the source to return a list of all the export targets that
    it knows how to convert. */
 XmTransferValue(dcs->transfer_id, _MOTIF_EXPORT_TARGETS,
                 (XtCallbackProc)TransferProc, NULL, NULL);
}
```

UTM calls the **TransferProc** routine when the source widget has finished converting _MOTIF_EXPORT_TARGETS. If the source widget knows how to convert the *PIXMAP* target, the **TransferProc** routine will request it. When the source widget finishes converting *PIXMAP*, the **TransferProc** routine will paste a copy of the returned pixmap into the DrawingArea.

```
void
TransferProc(Widget  w,
             XtPointer ignore,
             XtPointer call_data)
{
 XmSelectionCallbackStruct *scs =
       (XmSelectionCallbackStruct *) call_data;
 Atom TARGETS = XInternAtom(XtDisplay(w), "TARGETS", False);
 Atom _MOTIF_EXPORT_TARGETS = XInternAtom(XtDisplay(w),
                             "_MOTIF_EXPORT_TARGETS", False);
 Atom PIXMAP = XInternAtom(XtDisplay(w), "PIXMAP", False);
 Atom  *targets = (Atom *)scs->value;
 int    PIXMAP_is_supported = 0;
 unsigned long    n;
 Widget Label2;

   if ((scs->target == _MOTIF_EXPORT_TARGETS) && (scs->type == XA_ATOM))
     {
     printf("Number of supported targets is %ld\n", scs->length);
     ListAllTheTargets(w, targets, scs->length);
     for (n=0; n<=scs->length; n++)  {
  /*Look through list of returned TARGETS to see if PIXMAP is there.*/
        if (targets[n] == PIXMAP)  {
          PIXMAP_is_supported = 1;
        }
     }

     if (PIXMAP_is_supported)  {
       XmTransferValue(scs->transfer_id, PIXMAP,
                   (XtCallbackProc)TransferProc, NULL, NULL);
     }
   }

  if ((scs->target == PIXMAP)) {
    Pixmap        transferred_pixmap = *(Pixmap*) scs->value;
    Pixmap        copy_of_transferred_pixmap;
    Window        root_return;
    int           x, y;
    unsigned int  width, height;
    unsigned int  border_width, depth;
    GC            gc;
    XtGCMask      valueMask;
    XGCValues     values;

    printf("TransferProc: source has transferred the PIXMAP.\n");
    printf("We need to paste it into the DrawingArea.\n");

  /* It is better to display a copy of the returned pixmap than
     to display the returned pixmap itself. The following code
```

```
     creates the copy. */
    XGetGeometry(XtDisplay(w), (Drawable)transferred_pixmap,
                 &root_return, &x, &y, &width, &height,
                 &border_width, &depth);
    copy_of_transferred_pixmap =
                 XCreatePixmap(XtDisplay(w), XtWindow(w), width,
                               height, depth);
    valueMask = GCFunction;
    values.function = GXcopy;

    gc = XtGetGC(w, valueMask, &values);
    XCopyArea(XtDisplay(w), transferred_pixmap,
              copy_of_transferred_pixmap, gc, x, y,
              width, height, x, y);

    Label2 = XtVaCreateManagedWidget("Label2", xmLabelWidgetClass, w,
        XmNlabelType,   XmPIXMAP,
        XmNlabelPixmap, copy_of_transferred_pixmap,
        NULL);
    XtReleaseGC(w, gc);
    XmTransferDone(scs->transfer_id, XmTRANSFER_DONE_SUCCEED);
  }
}
```

17.2.3 Extending a Widget to Support Drag

Any widget that provides an **XmNconvertCallback** resource has the potential to be a drag source. However, not all the potential drag sources are actual drag sources. For example, **XmScrollBar** supports an **XmNconvertCallback** resource (through its superclass, **XmPrimitive**) but does not provide any code to become a drag source. However, your application can turn an **XmScrollBar** into a drag source. In fact, that is just what the sample program, **simple_drag** does. This program is stored in **simple_drag.c**.

The normal action for Button 2 Press has been overridden to cause it to call the **StartDrag** routine, which causes the drag to begin. The program allows only the Copy operation, and will reply to requests for compound text.

When a drag is started on the ScrollBar, the default drag icons are used.

The *main* routine of the **simple_drag** program instantiates a ScrollBar and a Text widget. The *main* routine also alters the translations of the ScrollBar widget, which is detailed later on in this chapter.

```
int
main(int argc, char **argv)
{
 static Widget  MainWindow;
 XtAppContext   app_context;
 Widget         Frame1, RC1;
 Widget         Text1, ScrollBar1;
 char    dragTranslations[] = "#override <Btn2Down>: StartDrag()";
 static  XtActionsRec  dragActions[] =
   {
    {"StartDrag", (XtActionProc)StartDrag}
   };
 XtTranslations            parsed_xlations;


   toplevel = XtAppInitialize(&app_context, "Test", NULL, 0,
                              &argc, argv, NULL, NULL, 0);

   MainWindow = XtVaCreateManagedWidget("MainWindow1",
                                   xmMainWindowWidgetClass, toplevel,
                                       NULL);

   CreateMenus(MainWindow);


 /* Create a RowColumn to contain the ScrollBar and Text widgets. */
    RC1 = XtVaCreateManagedWidget("RC1",
                                   xmRowColumnWidgetClass, MainWindow,
                                   NULL);

   /* Create a ScrollBar. */
    parsed_xlations = XtParseTranslationTable(dragTranslations);
    XtAppAddActions(app_context, dragActions, XtNumber(dragActions));
    ScrollBar1 = XtVaCreateManagedWidget("SB1",
                                   xmScrollBarWidgetClass, RC1,
                                   XmNorientation, XmHORIZONTAL,
                                   XmNtranslations, parsed_xlations,
                                   NULL);
 /* Associate a convert callback with the ScrollBar. */
   XtAddCallback(ScrollBar1, XmNconvertCallback, ConvertCallback, NULL);


 /* Create a text widget; it will be a potential drop site. */
   Text1 = XtVaCreateManagedWidget("Text",
                                   xmTextWidgetClass, RC1,
                                   XmNeditMode, XmMULTI_LINE_EDIT,
                                   XmNrows, 25,
                                   XmNcolumns, 25,
                                   NULL);

   XtRealizeWidget(toplevel);
   XtAppMainLoop(app_context);
}
```

When the user presses <Btn2Down>, **StartDrag** will be called. The **StartDrag** routine calls **XmeDragSource** as follows:

```
void
StartDrag(Widget w,
          XEvent *event)
{
 Arg       args[2];
 Cardinal  n=0;

  XtSetArg(args[n], XmNdragOperations, XmDROP_COPY); n++;
  XmeDragSource(w, NULL, event, args, n);
}
```

The **simple_drag** program provides an **XmNconvertCallback** procedure named **ConvertCallback**. This routine handles requests to convert targets. It knows how to convert the following targets:

- *MOTIF_EXPORT_TARGETS*
- *TARGETS*

• *COMPOUND_TEXT*

```
void
ConvertCallback(Widget  w,
                XtPointer ignore,
                XtPointer call_data)
{
 XmConvertCallbackStruct  *ccs = (XmConvertCallbackStruct *)call_data;
 int  *value;
 Atom COMPOUND_TEXT=XInternAtom(XtDisplay(w), XmSCOMPOUND_TEXT, False);
 Atom TARGETS = XInternAtom(XtDisplay(w), "TARGETS", False);
 Atom MOTIF_EXPORT_TARGETS =
   XInternAtom(XtDisplay(w), XmS_MOTIF_EXPORT_TARGETS, False);
 int n;

 /* XmeDragSource is going to call ConvertCallback and ask
    it to convert MOTIF_EXPORT_TARGETS. */
  if ( (ccs->target == MOTIF_EXPORT_TARGETS) ||
       (ccs->target == TARGETS))
    {
 /* We have to create a list of targets that ConvertCallback
    can convert. For simplicity, we are going to restrict
    that list to one target: COMPOUND_TEXT. */
     Atom *targs = (Atom *) XtMalloc(sizeof(Atom) * 1);
     if (targs == NULL) {
      ccs->status = XmCONVERT_REFUSE;
      return;
     }
     n = 0;
     targs[n] = COMPOUND_TEXT; n++;

     ccs->value = (XtPointer) targs;
     ccs->type = XA_ATOM;
     ccs->length = n;
     ccs->format = 32;
     ccs->status = XmCONVERT_DONE;  /* Yes, we converted the target. */
   }


 /* If the drop site supports COMPOUND_TEXT as an import target, then
```

```
    the drop site will ask ConvertCallback to convert the
    value to COMPOUND_TEXT format. */
   else if (ccs->target == COMPOUND_TEXT) {
     char     *passtext;
     char     *ctext;
     XmString  cstring;
     char      ValueAsAString[10];

  /* The part of the ScrollBar that we are transferring is its
     XmNvalue resource. */
     XtVaGetValues(w, XmNvalue, &value, NULL);

  /* Convert XmNvalue to COMPOUND_TEXT. */
       sprintf(ValueAsAString, "%d", value);
       cstring = XmStringCreateLocalized(ValueAsAString);
       ctext = XmCvtXmStringToCT(cstring);
       passtext = XtMalloc(strlen(ctext)+1);
       memcpy(passtext, ctext, strlen(ctext)+1);
  /* Assign converted string to XmConvertCallbackStruct. */
     ccs->value  = (XtPointer)passtext;
     ccs->type   = XA_STRING;
     ccs->length = strlen(passtext);
     ccs->format = 8;
     ccs->status = XmCONVERT_DONE;
   }

   else  {
     /* Unexpected target. */
     ccs->status = XmCONVERT_REFUSE;
   }
}
```

17.2.4 The DropSite Registry

The DropSite registry contains information about widgets that have been registered as drop sites. Although the drag icon can be dropped anywhere on the screen, only widgets that have been registered as drop sites can accept information from the initiator. The receiver is the application controlling the current drop site.

The "sensitive area" is the part of the widget that responds to drag and drop. By default, the sensitive area is the whole widget. However, the application can specify that only part of the widget is sensitive.

Widgets that are drop sites can be stacked on each other, with one widget partially or completely within the boundary of another. The sensitive areas of lower drop sites are clipped if they are covered by a higher widget.

The stacking order of the widgets with drop sites can be changed by the application.

17.2.5 Protocols

The protocol describes how the initiator and receiver clients interact through the toolkit with each other.

17.2.5.1 Drag Protocols

There are two types of drag protocol:

Preregister	Does not require messaging
Dynamic	Requires messaging; this is the default and is the recommended drag protocol.

Applications can support either, both, or neither. At Motif Release 2.0, dynamic is the default; you should try to use this. The Motif toolkit automatically supports both unless a user or client sets resources to force the use of one or the other.

The user can specify which drag protocol to use when the client is the initiator or receiver. The application can specify drag protocol in an application-class defaults file. If neither the application nor the user specifies a protocol, the dynamic drag protocol is used.

The toolkit uses the requested protocols and the protocols allowed by the initiator and receiver clients to arrive at the protocol actually being used. Therefore, the protocol can change as the drag icon moves from window to window, depending on which protocols each window supports. If the initiator and receiver cannot agree on a protocol, static default drag-over effects are generated.

Even if no drag-over or drag-under visual effects are shown, a drop can still occur with the drop protocol, unless a client has specified that that window does not participate in drag and drop.

17.2.5.2 Drop Protocol

Motif Release 2.0 uses UTM as the drop protocol. See Chapter 16 for details.

17.2.6 Drag and Drop Widget Classes

Motif provides a number of Xt objects and widgets to encapsulate the underlying protocol; however, these are not mapped onto the screen:

XmDisplay An object that contains display-specific information, such as the initiator and receiver protocol styles.

XmScreen An object that describes screen-specific information, such as font and default drag-over icons.

XmDragIcon
A widget that describes the pixmap, mask, and attachment of an icon. The source icon, state icon, operation icon, and the resulting blended drag icon are all Drag Icons.

XmDragContext
A widget that describes the resources specified by each drag initiator, such as target type, custom icons, custom colors, blending model, permitted operations, and callback routines for various situations encountered during the drag and drop transaction.

XmDropSite
A drop site database that maintains a registry of the resources unique to each drop site, such as animation for drag-under effects, valid target types and operations, and callback routines for situations encountered during a drag and drop transaction. It is not an Xt object, although it acts like one with respect to resource fetching.

XmDropTransfer
A widget that describes the information desired from the initiator client and the procedure used to process the results.

17.2.7 Drag and Drop Functions

Motif provides many functions to support drag and drop processing. However, many drag and drop functions were developed prior to UTM and, although not obsolete, no longer get much direct use. Application programmers will find the following drag and drop functions to be most useful:

XmeDragStart
Initiates a drag.

XmeDropSink
Establishes a widget as a drop site.

XmCreateDragIcon
Creates any of the parts of a drag icon (status icon, operation icon, or source icon) from a cursor or pixmap. This allows custom icons for all or part of the drag icon, rather than the default icons.

XmDragCancel
Cancels a drag that is in progress. This function is called when the user presses osfCancel.

XmDragStart
This function is typically called by **XmeDragSource** when the user asks to start a drag. This function creates a DragContext object, which is referenced by other functions whenever information about the drag initiator is needed. The **XmDragStart** function also calls the DragStart callback procedure, if one has been nominated with the **XmNdragStartCallback** resource of the **XmDisplay** widget.

XmDropSiteConfigureStackingOrder
Sets the order of overlapping drop sites. The default order is with the first-registered drop site on the bottom and the last-declared drop site on top.

XmDropSiteEndUpdate
Causes the **XmDropSiteUpdate** requests made after **XmDropSiteStartUpdate** to take place.

XmDropSiteQueryStackingOrder
Provides information about the stacking order of overlapping drop sites. The order can be changed with **XmDropSiteConfigureStackingOrder**.

XmDropSiteRegister
Registers a drop site. Resources describing the drop site are defined. **XmeDropSink** calls **XmDropSiteRegister**.

XmDropSiteRetrieve
Retrieves the values of drop site resources.

XmDropSiteStartUpdate
Signals the toolkit to wait until **XmDropSiteEndUpdate** is called to process drop site changes requested by **XmDropSiteUpdate**. This provides a more efficient way to update several drop sites than changing them one at a time.

XmDropSiteUpdate
Updates drop site resources for a single drop site. If a series of **XmDropSiteUpdate** requests are surrounded by **XmDropSiteStartUpdate** and **XmDropSiteEndUpdate**, then the changes will be made all at once after the end update request.

XmDropSiteUnregister
Removes a drop site. After a drop site has been unregistered, it is unavailable as a destination for a drag.

XmDropTransferAdd
Adds additional transfer requests once a transfer has started.

XmDropTransferStart
Specifies what information should be requested from the drag initiator, and starts the process to get the information.

XmGetDragContext
Returns the DragContext ID associated with a particular time stamp.

XmGetXmDisplay
Returns the ID for the specified display.

XmGetXmScreen
Returns the ID for a specified screen. Some resources, such as the drag icons, are screen-specific.

XmTargetsAreCompatible
Checks if there are any matching targets between the initiator and destination to help determine the correct drag state.

17.2.8 Targets

Each drag source and drop site specifies what kinds of data types it can process, called targets. These targets are atoms, such as *XA_STRING*.

The DragContext resources **XmNexportTargets** and **XmNnumExportTargets** provide a list and number of the data types provided by the drag source. These are export targets.

The DropSite resources **XmNimportTargets** and **XmNnumImportTargets** provide a list and number of the data types accepted by the drop site. These are known as import targets. The primary purpose of the targets in **XmNimportTargets** is to provide visual feedback about the validity of the drop site. The **XmNdestinationCallback** procedure need not use the same targets as those specified by **XmNimportTargets**.

Any number of targets may be listed for each source and site. A drop site is considered valid for a particular drag if at least one of its targets matches any of the source’s targets and if the source and drop site operations are compatible.

An application can define anything it wants as a target. Be aware, however, that other applications might not recognize that target.

17.2.9 Operations

There are three ways that the initiator and receiver can interact with each other:

- Data can be moved from the initiator to the receiver (Move).
- Data can be copied from the initiator to the receiver (Copy).
- Data can be linked from the receiver to the initiator (Link).

When a drag is started, the initiator provides a list of valid operations in the DragContext **XmNdragOperations** resource. When a drop site is registered, the receiver provides a list of operations it supports in the DropSite **XmNdropSiteOperations** resource. These lists are the values **XmDROP_MOVE**, **XmDROP_COPY**, or **XmDROP_LINK**, connected by the bitwise OR operator (|). For example, the following value means that Move and Copy are valid operations, but Link is not:

```
XmDROP_MOVE | XmDROP_COPY
```

The value **XmDROP_NOOP** indicates that there are no operations possible for a drop at the current site.

The user can specify an operation by using key combinations discussed earlier in this chapter. The user can also change the operation at any time until the drop starts.

The initiator and the receiver need to be able to handle all the operations their application supports. If the operation is Move, the receiver first gets a copy of the data, then tells the initiator that it can delete the data. If the operation is Copy, both applications have the data, making two copies of it. If the operation is Link, there is only one copy of the data, and the receiver establishes a link to that copy.

17.2.9.1 Drop Site Status

The drag and drop callbacks for both receiver and initiator contain a **dropSiteStatus** field. This field is initialized and maintained by the receiver through the toolkit, although the receiver's drag and drop procedures can update it if they wish. This field is used by the toolkit to determine what drag-over and drag-under visual effects to use.

The **dropSiteStatus** field indicates the relationship of the drag source to the drop site over which the drag icon is located:

XmDROP_SITE_VALID
A drop can take place. There is at least one matching target and operation between the drag source and the drop site.

XmDROP_SITE_INVALID
A drop cannot take place. Either there were no matching targets, no matching operations, or the receiver's **XmNdragProc** or **XmNdropProc** discovered some other problem that would make a drop impossible.

XmNO_DROP_SITE
The drag icon is not over a drop site.

If the toolkit on the receiver's side has set either the *operation* or the *operations* field to **XmDROP_NOOP**, it also sets the **dropSiteStatus** field to **XmDROP_SITE_INVALID**.

17.3 Drag and Drop Protocols

The protocols refer to how the initiator and receiver clients use the toolkit to communicate with each other. There are two drag protocol styles that are available. The drop protocol is based on the Xt selection protocol.

17.3.1 Drag Protocols

The toolkit on the initiator side is in charge during the drag. The protocol in effect determines how it will find the information about drop sites that it needs to manage visuals, and how extensively the initiator and receiver clients are involved during the drag.

There are two kinds of drag protocol styles:

Dynamic	Uses messages from the toolkit to the receiver to find out drop site information. The toolkit on the receiver side can reply to the messages, or the application can take action based on these messages. The receiver manages the drag-under effects.
Preregister	Stores drop site information in a database when the drop site is registered. The receiver is not involved in the drag until a drop occurs. All drag-over and drag-under visual effects are managed by the toolkit on the initiator side.

The code for the initiator is the same regardless of the protocol. The code for the receiver applications is the same except that in the dynamic mode, or in the preregister mode when the initiator and receiver are the same client, the receiver's **XmNdragProc** is called.

The drag protocol in use can change during the course of a drag. When the drag icon enters or leaves a top-level window, the source and potential drop receiver negotiate a mutually acceptable drag protocol, as described in Section 17.3.2.

The toolkit uses pixmap source icons if the client provides them. If not, the toolkit uses bitmap source icons if the client provides them. If the client provides neither, the toolkit uses **XmScreen** icons. The **XmScreen** icons can be either the default icons or ones provided by the client or user.

17.3.1.1 The Dynamic Drag Protocol

With the dynamic drag protocol, the initiator and receiver communicate with messages through the toolkit.

As the drag icon moves within the receiver's window, messages are sent from the toolkit on the initiator side to the toolkit on the receiver side. Based on these messages, the receiver determines whether the drag icon is entering, within, or leaving a drop site. Although the toolkit on the receiver side initializes state and operation information, the receiver can check and update this information further if it registers an **XmNdragProc** for the drop site. The initiator receives the updated message in one of its drag-related callbacks (described later in this chapter), and can take action accordingly.

The dynamic drag protocol allows the receiver to provide more sophisticated visual effects using the **XmNdragProc** than the toolkit can provide alone.

With the dynamic drag procotol

- The receiver can provide custom drag processing and drag-under visual effects.
- The drag icon can be any size supported by the system on which the application is running.

The dynamic protocol is the default for drag and drop operations.

17.3.1.2 The Preregister Drag Protocol

When a receiver supports the preregister protocol, the toolkit on the receiver side stores drop site information in a database. The toolkit on the initiator side manages all drag-under effects based on the information in the drop site database. By setting some DropSite resources appropriately, the receiver can have the toolkit use different highlighting or pixmaps, but the receiver only participates directly in the drag-under effects when it is the same as the initiator and an **XmNdragProc** procedure has been registered with the drop site.

With the preregister protocol

- The server is grabbed.
- The only customization a receiver can perform is providing custom values for the DropSite visual resources.
- The drag icon can be any size supported by the system on which the application is running.

17.3.2 Choosing the Protocol and Visual Style

The user can specify which drag protocol to use or the application can specify the drag protocol in resource file.

The preregister drag protocol can be used with a minimum of additional coding in an application, because the toolkit manages the drag-over visual effects using the default drag icons specified in the **XmScreen** object. Or the application can override the default **XmScreen** icons with custom icons, but still allow the toolkit to manage the effects.

The dynamic drag protocol requires more work for the application program, but allows a receiver application to provide visual effects beyond the capabilities of the toolkit.

The drag protocol in use has an effect on the system performance as described later in this section.

17.3.2.1 Specifying Drag Protocols

Two Display resources specify which protocol the toolkit should try to use when a client is an initiator or receiver. These resources can be set by the client in a resource file or by the user.

- **XmNdragInitiatorProtocolStyle**
- **XmNdragReceiverProtocolStyle**

These resources can take the following values (the letter in brackets following the value is used in Table 17-1):

XmDRAG_NONE [N]
: Does not participate in drag and drop. There are no drag-under effects. The drag-over effects depend on the value of **XmNdragInitiatorProtocolStyle**.

XmDRAG_DROP_ONLY [X]
: Does not support either the preregister mode or the dynamic mode, but does data transfer after the drop occurs. There are no drag-over or drag-under visual effects.

XmDRAG_PREREGISTER [P]
: Supports the preregister mode only. The visual effects are managed by the toolkit.

XmDRAG_PREFER_PREREGISTER [PP]
: Supports both protocols, but prefers the preregister protocol. This is the default for receivers. The visual effects are determined by the protocol actually used.

XmDRAG_PREFER_RECEIVER [R]
: Used by initiators only. Uses the protocol that the receiver specifies. This is the default for initiators. The visual effects are determined by the protocol actually used.

XmDRAG_PREFER_DYNAMIC [PD]
: Supports both protocols, but prefers the dynamic mode. The visual effects are determined by the protocol actually used.

XmDRAG_DYNAMIC [D]
: Supports the dynamic protocol only. The drag-over and drag-under visual effects are managed by the clients.

For example:

```
myclient*dragInitiatorProtocolStyle: DRAG_PREFER_DYNAMIC
myclient*dragReceiverProtocolStyle: DRAG_PREFER_DYNAMIC
```

If the initiator and receiver have specified the same protocol, that protocol is used. If they specify different protocols, the protocol that is used is shown in the following table.

Table 17–1. Initiator and Receiver Protocols

Initiator Protocol	**Receiver Protocol**						
		P	*PP*	*PD*	**D**	**X**	**N**
P	P	P	P	X	X	N	
PP	P	P	P	D	X	N	
R	P	P	D	D	X	N	
PD	P	D	D	D	X	N	
D	X	D	D	D	X	N	
X	X	X	X	X	X	N	
N	N	N	N	N	N	N	

The **XmGetXmDisplay** function returns the Display object ID associated with a specific display. **XtGetValues** can be used to check the protocol style resources.

If an **XmNdragProc** is specified for a drop site, it will be performed only if the protocol is dynamic, or, if the protocol is preregister, only if the initiator and receiver are the same client. In this case, the application should set the **XmNdragReceiverProtocolStyle** resource to the value **XmDRAG_PREFER_DYNAMIC** in the application-class defaults file rather than use the default value.

17.3.2.2 Protocols and Visuals

When the resulting protocol is preregister, a preregister visual style is used, and the server is grabbed by the initiator. With the dynamic protocol, the initiator and the receiver exchange messages. The visual look of the two protocols is very similar. The most significant visual difference between the two protocols is associated with sophisticated drag-under effects, such as autoscroll drag, or any drag-under operation that depends on widget data not displayed at the initiation of the drag. These effects are only available with the dynamic protocol, or with the preregister protocol, under the condition that the initiator and the receiver are the same client.

When the resulting protocol is dynamic, a dynamic visual style is always used. The drag-over visual can be a pixmap with an arbitrary size whose depth and colormap are the same as those of the widget associated with the drag source. When a pixmap is used in dynamic mode, a shaped window is used to contain it. The pixmap, if desired, is specified in the DragContext **XmNsourcePixmapIcon** resource. Otherwise, the drag-over visual is implemented with the X cursor, which must be a bitmap, and often has limited size (use **XQueryBestSize** to find out the largest size available per screen). This cursor is specified using **XmNsourceCursorIcon**.

Consequently, when the resulting protocol is **XmDRAG_NONE** or **XmDRAG_DROP_ONLY**, the visual style depends upon the value of **XmNdragInitiatorProtocolStyle**. When it is **XmDRAG_DYNAMIC** or **XmDRAG_PREFER_DYNAMIC**, the dynamic visual style is used; otherwise, the preregister visual style is used.

17.3.3 Drop Protocol

When a drop is made, the receiver checks what action should happen:

- If the user requested help, the receiver should display a dialog explaining the consequences of a drop on the site and determine if the user wants to continue or cancel the drop.
- If the user requests a cancel from the help dialog or presses osfCancel, or if the receiver determines that the drop cannot continue, the receiver must call *XmTransferDone* to terminate the transfer. The *status* argument to *XmTransferDone* must be **XmTRANSFER_DONE_FAIL**.
- If the drop can continue normally, the receiver starts a transfer.

See the Chapter 16 for details on how UTM implements a drop.

17.4 Drop Receiver Responsibilities for Dragging

In the dynamic protocol, motion messages go first to the receiver client. The receiver evaluates the state of the drag and sends an updated message to the initiator, which then manages its drag-over visuals based on the results.

If the drag protocol in effect is preregister, the drop site information is put in the database as the drop sites are registered and the receiver client does nothing until a drop is made. All visual effects are handled by the toolkit.

If the drag protocol is dynamic, messaging begins when the pointer enters the window containing the drop site. The receiver is given the opportunity to provide additional processing in its **XmNdragProc**. If the protocol is preregister, the opportunity is still available, as long as the initiator and receiver are the same client. The **XmNdragProc**

- Receives messages when the drag icon enters or leaves the drop site, the operation changes, the drag icon is in motion, or the drag is cancelled.
- Provides information back to the toolkit about the state of the drag (valid drop site, invalid drop site, no drop site) and the operation to be performed when a drop is made.
- Manages any custom drag-under visual effects.

17.4.1 Establishing a Drop Site

The widgets shown in Appendix B are already registered as drop sites. An application must register any other widget it wants to use for a drop site, but only if the widget provides an **XmNdestinationCallback** procedure. A widget may be registered as only one drop site.

XmeDropSink registers a widget as a drop site, establishes callbacks to be used when a drag is made through the drop site or a drop is made in the drop site, and provides target and operation information. If the protocol is preregister, the information is stored in a database, which is read by the toolkit during the drag. If the drag protocol is dynamic, messaging is used to check for possible drop sites within a widget.

The optional **XmNdragProc** routine is executed only if the drag protocol is dynamic, or, if the protocol is preregister, only if the initiator and receiver are the same client.

It is called in response to events during the drag, and allows the receiver to provide additional drag-under effects or additional drag processing.

The **XmNdropSiteOperations** resource lists all operations that the drop site will support, combined by the bitwise OR operations (|). For example, the default value

```
XmDROP_COPY | XmDROP_MOVE
```

means that Copy and Move are valid operations, but Link is not. During a drag, the toolkit on the receiver side compares this list with the DragContext's **XmNdragOperations** list and the user-selected operation to arrive at the operation that will be performed if a drop occurs on this site, along with a list of all operations possible between the initiator and the current drop site.

If an application wants to use only one operation, such as Copy, it should set the **XmNdropSiteOperations** field to just that operation to ensure that the toolkit chooses the correct operation and drag icon during the drag and drop transaction.

Drop sites that represent copying devices, such as printers, or transformation devices, such as compilers, should perform a Copy rather than a Move if both are possible.

The **XmNdropSiteActivity** resource indicates whether the drop site is available for use:

XmDROP_SITE_ACTIVE
The drop site is available for use. This is the default value.

XmDROP_SITE_INACTIVE
The drop site is not available for use. If the drag icon is moved over the drop site, both the icon and drop site act as if the icon were not over a drop site.

XmDROP_SITE_IGNORE
Ignore this drop site for all purposes.

Use the **XmDropSiteRegistered** function to determine whether a widget is already a registered drop site. The function accepts a widget as an argument, and returns a Boolean value, which is TRUE if the widget is a registered drop site.

The **XmDropSiteUnregister** function removes a widget from the DropSite registry. Once a widget is unregistered, it displays no drag-under visual effects and cannot accept a drop.

The difference between an unregistered drop site and an inactive drop site is that the inactive drop site is still registered; it still uses memory, but does not engage in any drag and drop transactions. One use for inactive drop sites is to provide the correct clipping on overlapping drop sites. An unregistered drop site is no longer involved in the drag and drop system. It is the same as a widget that was never registered.

When the drag and drop is operating in preregister mode, and the receiving client is not the same as the initiating client, drop effects will be generated even if a drop site's widget is unmanaged. To avoid this, you can deactivate the drop site by calling **XmDropSiteUpdate** and setting **XmNdropSiteActivity** to **XmDROP_SITE_INACTIVE**. Specifying the resource as **XmDROP_SITE_IGNORE** at creation time will disable the drop site completely, producing some savings of computational overhead, since inactive drop sites are still used for clipping drag-under effects.

The following code registers a DrawingArea widget as a drop site. The only operation it will accept is Copy. The only import target it will accept is *COMPOUND_TEXT*. The other resources, including drag-under effects, are left at their default values.

```
DrawingArea = XmCreateDrawingArea(Frame, "Draw1", NULL, 0);
XtManageChild(DrawingArea);
Atom targets[2];
Arg  args[4];
int  n, nt;
Atom COMPOUND_TEXT = XInternAtom(XtDisplay(w), "COMPOUND_TEXT", False);

  /* Register the DrawingArea as a drop site */
  nt=0
  targets[nt++] = COMPOUND_TEXT;
  n = 0;
  XtSetArg(args[n], XmNdropSiteOperations, XmDROP_COPY); n++;
  XtSetArg(args[n], XmNimportTargets, targets); n++;
  XtSetArg(args[n], XmNnumImportTargets, nt); n++;
  XmeDropSink(DrawingArea, args, n);
  XtAddCallback(DrawingArea1, XmNdestinationCallback,
                DestinationCallback1, NULL);
```

17.4.1.1 Changing a Drop Site

The **XmDropSiteUpdate** function is used to change drop site resources for a single drop site. For multiple requests, **XmDropSiteStartUpdate** signals that a series of **XmDropSiteUpdate** requests will follow, and **XmDropSiteEndUpdate** ends the series and processes the requests at one time.

XmDropSiteUpdate can also be used to change the resource values of the widgets that register themselves as drop sites.

17.4.1.2 Specially Shaped Drop Sites

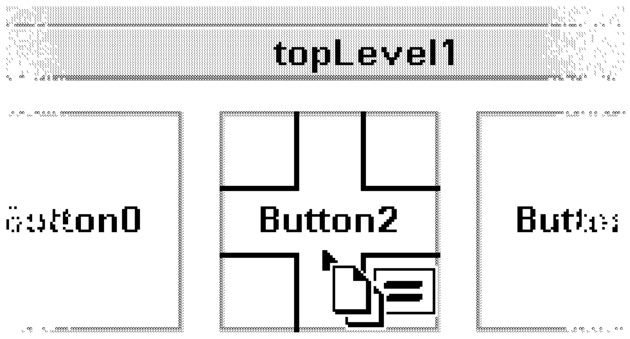
The shape of a simple drop site can be specified as the union of a set of specified rectangles clipped by the associated widget.

If only part of the widget is to be sensitive to a drop, it is defined by a list of rectangles in the **XmNdropRectangles** resource. If the resource is NULL, the drop site is the smallest enclosing widget and the shape of the drop site is the shape of the widget.

The rectangles that make up the drop site do not need to be contiguous. All the noncontiguous segments of the drop site act as one; they are all highlighted the same way at the same time. A drop on one segment is the same as a drop on any of the other segments. This might look to the user as if there were several drop sites on a single widget, but the application handles nested drop sites differently from drop sites made of noncontiguous segments. Nested drop sites, whether simulated or real, may have different drag-under effects, targets, operations, or callback procedures.

The following example establishes a sensitive area shaped like a plus sign on a DrawnButton widget named Button2. Even if the drag icon is within the Button2 widget, no drag-under effects are shown until the drag icon is within the sensitive area. The area is visible only when a drag icon enters it and highlighting occurs. The sensitive area is the only part of the widget that accepts a drop.

Figure 17–2. Specially Shaped Drop Site



```
XRectangle plus[] = {
    {30, 0, 30, 30},
    {0, 30, 90, 30},
    {30, 60, 30, 30},
    };
...
n = 0;
XtSetArg(args[n], XmNimportTargets, importList); n++;
XtSetArg(args[n], XmNnumImportTargets, 1); n++;
XtSetArg(args[n], XmNdropRectangles, plus); n++;
XtSetArg(args[n], XmNnumDropRectangles, 3); n++;
XmeDropSink(Button2, args, n);
```

17.4.1.3 Nested Drop Sites

A widget can be registered as only one drop site. However, widgets that are registered as drop sites can be nested within each other, providing nested drop sites.

The **XmNdropSiteType** indicates the complexity of the drop site:

XmDROP_SITE_SIMPLE
The drop site contains no other drop sites.

XmDROP_SITE_COMPOSITE
The drop site contains other drop sites. This value is generally associated with a Manager.

A composite drop site must be registered before any of its children are registered. If a composite drop site is inactive, so are all of its children.

The composite and children drop sites do not need to have the same operations or targets.

A manager that contains a number of widgets with their associated drop sites does not need to be a composite drop site unless it is possible to drop in the background of the manager.

It is possible for an application to simulate nested drop sites on a single widget, for example, a DrawingArea. The process is described as part of the discussion of the duties of the optional **XmNdragProc** routine in Section 15.4.2.

17.4.1.4 Overlapping Drop Sites

Drop sites can overlap. Their stacking order is assumed to correspond to the order in which they are registered, with the first-registered one on top. **XmDropSiteQueryStackingOrder** checks the stacking order, whereas **XmDropSiteConfigureStackingOrder** changes it.

When a drop site is overlapped by another drop site, the drag-under effects of the drop site underneath are clipped as appropriate by the obscuring drop site.

A widget or gadget that is not a drop site can overlap and partially obscure a drop site. To ensure that the drop-site's drag-under visuals are appropriately clipped by the obscuring widget, such sibling widgets should be registered as inactive drop sites. Parent widgets, whether drop sites or not, will clip their children's drop site visuals. If a parent has some active and some inactive drop site children, it should be registered as an active drop site.

17.4.1.5 Drag-Under Visual Effects

Drag-under visual effects are displayed only when the pointer is within the sensitive area of the drop site widget. An application can either handle all or none of the animation effects for a particular drop site. That is, an application should never do a partial job of animation on a particular drop site. Various drag-under styles can be chosen in the **XmNanimationStyle** DropSite resource:

XmDRAG_UNDER_HIGHLIGHT
A solid border around the sensitive area of the drop site is used to show the drop site is valid. This is the default value.

XmDRAG_UNDER_SHADOW_OUT
The sensitive area of the drop site looks pushed out when it is valid.

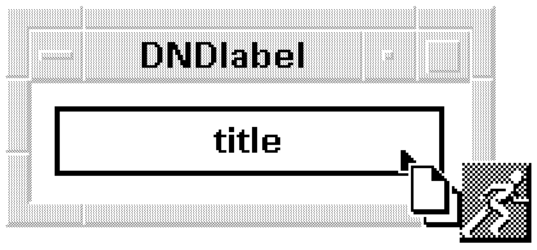
XmDRAG_UNDER_SHADOW_IN
The sensitive are of the drop site looks pushed in when it is valid.

XmDRAG_UNDER_PIXMAP
A custom pixmap is used to indicate the drop site is valid. The pixmap is specified in **XmNanimationPixmap**.

XmDRAG_UNDER_NONE
No indication is given that the drop site is valid.

The following illustration shows the default drag-under animation around the Label widget drop site.

Figure 17–3. Default Drag-Under Animation



If the value of **XmNanimationStyle** is **XmDRAG_UNDER_PIXMAP**, the resources **XmNanimationPixmap**, **XmNanimationMask**, and **XmNanimationPixmapDepth** are used to provide more information about the pixmap. If the depth does not match the depth of the window controlling the drop site widget, no animation occurs. Except for **XmDRAG_UNDER_PIXMAP**, the colors used for the visual effects are based on the colors of the widget associated with the drop site.

The dynamic protocol provides the most control over the drop site animation. It is the only way to get visual effects that do not remain the same for the duration of the drag icon's stay in the drop site; for example, a background that flashes.

17.4.2 XmNdragProc

The procedure registered in the DropSite's **XmNdragProc** resource is called only when the dynamic protocol is in effect, or, in the case of the preregister protocol, if the initiator and receiver are the same client. This procedure is optional. Applications that need to provide special drag-under effects or other special processing during a drag can do so with this procedure.

The **XmNdragProc** procedure is called in response to messages from the toolkit, before the initiator's equivalent drag callback. The pointer from the drop site's **XmNclientData** resource is passed as the client data to this procedure. Applications may use this resource to pass any manner of information desired to the **XmNdragProc** procedure. Fields in the callback structure provide information to the receiver about the drag.

The *reason* field in the callback structure indicates why the procedure was called.

XmCR_DROP_SITE_ENTER_MESSAGE
The drag icon hotspot has entered the drop site.

XmCR_DROP_SITE_LEAVE_MESSAGE
The drag icon hotspot has left the drop site.

XmCR_DRAG_MOTION_MESSAGE
The drag icon hotspot has moved.

XmCR_OPERATION_CHANGED_MESSAGE
The operation has changed.

The *operations* field lists all the operations that are valid for the drop site with the current drag source. The *operations* field is initialized by the toolkit as follows:

- If the user has selected an operation, the value of *operations* is initialized to that operation if it is in the DragContext's **XmNdragOperations** list.
- Otherwise, the *operations* field is initialized to the list in the DragContext's **XmNdragOperations** list.

The *operation* field indicates the type of action a successful drop will perform. The toolkit initializes the *operation* field by taking the following steps, in order of precedence from highest to lowest:

1. If Move is a valid operation (in both the *operations* field and the DropSite's **XmNdropSiteOperations** list), *operation* is initialized to **XmDROP_MOVE**.
2. If Copy is a valid operation, *operation* is initialized to **XmDROP_COPY**.
3. If Link is a valid operation, *operation* is initialized to **XmDROP_LINK**.
4. Otherwise, *operation* is initialized to **XmDROP_NOOP**.

The **dropSiteStatus** field provides an indication of whether a transfer between the initiator and this drop site could occur. The value that the toolkit selects for the **dropSiteStatus** field depends on the reason the **XmNdragProc** procedure was entered:

- If the reason is motion or drop site leave, and the drop site is the same as in the last call to **XmNdragProc**, the **dropSiteStatus** field is the same as at the end of the previous call.
- Otherwise, if there is at least one target in common and at least one operation in common, the value is initialized to **XmDROP_SITE_VALID**. If not, the value is initialized to **XmDROP_SITE_INVALID**.
- If the *operation* field is **XmDROP_NOOP**, the **dropSiteStatus** field is initialized to **XmDROP_SITE_INVALID**.

The **XmNdragProc** procedure can update *operation*, *operations*, or **dropSiteStatus** further during its execution. The final values for these fields are available to the initiator in its drag callback structures. If the receiver's **XmNdragProc** procedure is called more than once while the drag icon is within the drop site (for example, because of motion events), the values used by the toolkit when it initializes the drag callback *operations*, *operation*, and **dropSiteStatus** fields are the ones at the end of the previous call to **XmNdragProc**.

The *animate* field tells the toolkit who should provide the drag-under visual effects. It is initially set to True, but the **XmNdragProc** routine can set it to False.

True	The toolkit provides the drag-under visuals as if the protocol were preregister.

False	The receiver provides the drag-under visuals. The application can provide special visual effects, such as a blinking background, that are not possible with the toolkit.

The **DragProcCallback** routine in the **DNDDemo.c** program is an example of a **DragProc** routine. It can process every drag message, changes the *operations*, *operation*, and **dropSiteStatus** as necessary, and sets the *animate* field to **True**, allowing the toolkit to manage the drag-under effects. The **DragProcCallback** routine is shown in the next section of this chapter.

17.4.2.1 Simulating Nested Drop Sites

A widget can be registered as only a single drop site. However, if the application needs one or more drop sites entirely enclosed within another drop site, there are two ways to accomplish this:

- Widgets that contain other widgets that are drop sites should be registered as composite drop sites as described earlier in this chapter.

  This method allows the toolkit to manage drop site messages and drag-under effects for each nested drop site.

- An application can simulate multiple drop sites on a single widget in the **XmNdragProc** and **XmNdropProc** routines. Because the **XmNdragProc** routine is executed only in the dynamic drag protocol mode, this method would only work if the drag procotol chosen is preregister in the case where the initiator and receiver are the same client.

  This method requires that the application manage all drag-under effects, because the toolkit is not aware of the simulated nesting.

To simulate nested drop sites on a single widget:

1. Register the widget as a single active drop site. Set **XmNdropSiteOperations** to all the operations possible for any of the nested drop sites. Set **XmNimportTargets** to all the targets possible for any of the nested drop sites. Register an **XmNdragProc** routine to provide any special drag-under effects for the simulated drop sites.

The *operations*, *operation*, and **dropSiteStatus** fields are initialized by the toolkit only when this outer drop site is entered or left. The simulated drop sites must be managed by the application.

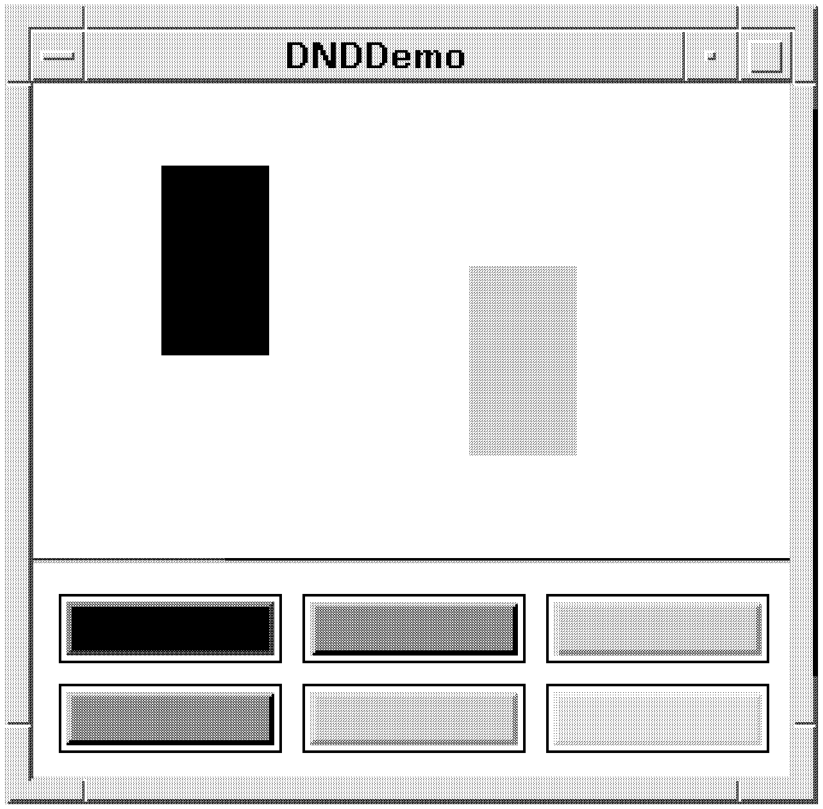
2. When either **XmNdragProc** or **XmNdropProc** is called, check the *x* and *y* fields in the callback structure to determine which of the nested drop sites contains the pointer.
3. If the pointer is within a simulated nested drop site, update the callback fields as follows:
   - When the pointer enters the simulated nested drop site, save the value of the *operations* and *operation* fields.
   - Remove any operations from the *operations* field that do not apply to the simulated drop site.
   - Set *operation* to the valid operation preferred by the simulated drop site, or to **XmDROP_NOOP** if the *operations* list does not contain the preferred operation.
   - The **dropSiteStatus** field must reflect the status of the simulated drop site so that the initiator can manage drag-over effects correctly:
     - — Set the **dropSiteStatus** to **XmDROP_SITE_VALID** if the *operation* is allowed in the simulated drop site and if there is at least one target in common between the simulated drop site and the initiator. (Use the **XmTargetsAreCompatible** routine to check the targets.)
     - — Set the **dropSiteStatus** to **XmDROP_SITE_INVALID** if the *operation* is not allowed in the simulated drop site, if there are no targets in common, or if the *operation* is **XmDROP_NOOP**.
   - Display appropriate drag-under visual effects.
   - When the pointer leaves the simulated drop site, restore the original values of *operations* and *operation* that apply to the outer drop site.
4. If the pointer is not within a simulated drop site, but drops are allowed in the outer drop site, update the fields as described in the previous step.
5. If the pointer is not within a simulated drop site, and drops are not allowed in the outer drop site, set the **dropSiteStatus** field to **XmDROP_SITE_INVALID**.

If the preregister protocol is in effect, the simulated drop sites cannot be managed during the move, because **XmNdragProc** is not performed; but they can be managed at the drop with **XmNdropProc**.

In the following example, only the top-level window, **DNDDemo**, is registered as a drop site. The user can create rectangles within the window that then act like drop sites themselves. The user can drag and drop colors from one of the six buttons in the lower part of the window onto the rectangles to change the color of the rectangle. However, these rectangles are not registered drop sites; they are simulated.

The user can also drag these rectangles to new locations.

Figure 17–4. Simulated Drop Sites



The **RegisterDropSite** routine registers the DrawingArea widget as a drop site. The list of operations and targets may not be valid for each simulated drop site, but are valid for other simulated drop sites.

```
static void
RegisterDropSite(Widget w)
{
    Display *display = XtDisplay(w);
    Atom    targets[3];
    Arg     args[5];
    int     n = 0;

    /* Only accept moves or copies */
    XtSetArg(args[n], XmNdragOperations, XmDROP_COPY | XmDROP_MOVE); n++;

    /* set all possible targets for any of the nested drop sites */
    targets[0] = XmInternAtom(display, "_MY_RECTANGLE", False);
    targets[1] = XmInternAtom(display, "BACKGROUND", False);
    targets[2] = XmInternAtom(display, "PIXMAP", False);
    XtSetArg(args[n], XmNimportTargets, targets); n++;
    XtSetArg(args[n], XmNnumImportTargets, 3); n++;

    /*register a dragProc - necessary for simulating nested drop sites*/
    XtSetArg(args[n], XmNdragProc, DragProcCallback); n++;
    XmeDropSink(w, args, n);
}
```

The **XmNdragProc** routine, **DragProcCallback**, is called whenever a drag icon enters the registered drop site (the top-level window). The **RectFind** routine from **DNDDraw.c** determines if the pointer is in a simulated drop site. The **CheckTargets** routine determines if the object being dragged is one of the six colors (**bgFound**) or one of the created rectangles (**rectFound**). (The value **pixFound** to represent a pixmap being dragged is coded in this routine, but not in the rest of the program.)

The only drag-under visual is displayed when a color is dragged to a rectangle. The outline of the rectangle is highlighted.

The entire **DragProcCallback** routine is too long to be listed in its entirety here. The section dealing with the drop site enter message is used as an example.

```
static void
DragProcCallback(Widget w, XtPointer client, XtPointer call)
{

    XmDragProcCallbackStruct    *cb = (XmDragProcCallbackStruct *) call;
    Display                     *display = XtDisplay(w);
    Boolean                     rectFound, bgFound, pixFound;
    static unsigned char        initial_operations;
    static unsigned char        initial_operation;
    RectPtr                     rect;
```

```
CheckTargets(cb->dragContext,display,&rectFound,&bgFound,&pixFound);

switch(cb->reason) {

    case XmCR_DROP_SITE_ENTER_MESSAGE:

        /* save the value of the operations and operation fields */
        initial_operations = cb->operations;
        initial_operation = cb->operation;

        rect = RectFind(cb->x, cb->y);

        /* Remove any operations for the operations field
         * which do not apply to the simulated drop site.
         */
        if (rect) {

            if (bgFound || pixFound) {
                cb->operations = XmDROP_COPY;
                RectHighlight(w, rect);
            }
            else if (rectFound) {
                cb->operations = cb->operations &
                                 (XmDROP_COPY | XmDROP_MOVE);
                RectUnhighlight(w);
            }

        }
        else {
            cb->operations = initial_operations &
                             (XmDROP_COPY | XmDROP_MOVE);
            RectUnhighlight(w);
        }

        /* Set operation to the valid operation preferred by
         * the simulated drop site or to XmDROP_NOOP if the
         * operations list does not contain the preferred operation.
         */
        if (rect) {
```

```
    if (bgFound || pixFound) {

        if (cb->operations & XmDROP_COPY)
            cb->operation = XmDROP_COPY;
        else
            cb->operation = XmDROP_NOOP;

    }
    else if (rectFound) {

        if (cb->operations & XmDROP_MOVE)
            cb->operation = XmDROP_MOVE;
        else if (cb->operations & XmDROP_COPY)
            cb->operation = XmDROP_COPY;
        else
            cb->operation = XmDROP_NOOP;

    }

}
else {

    if (rectFound) {

        if (cb->operations & XmDROP_MOVE)
            cb->operation = XmDROP_MOVE;
        else if (cb->operations & XmDROP_COPY)
            cb->operation = XmDROP_COPY;
        else
            cb->operation = XmDROP_NOOP;

    }
    else
        cb->operation = initial_operation;

}

/*
 * Set dropSiteStatus to XmDROP_SITE_INVALID
```

```
             * if the operation
             * field is XmDROP_NOOP, or if there are no common targets
             * between the source and the nested drop site. Otherwise,
             * set dropSiteStatus to XmDROP_SITE_VALID.
             */
            if (cb->operation == XmDROP_NOOP ||
                (rect && (!rectFound && !bgFound && !pixFound)) ||
                (!rect && !rectFound))
                cb->dropSiteStatus = XmINVALID_DROP_SITE;
            else
                cb->dropSiteStatus = XmVALID_DROP_SITE;

            /*
             * Display appropriate drag under visuals. Only highlight
             * the rectangle if we are changing rectangle attributes.
             */
            if (rect && bgFound || pixFound &&
                cb->dropSiteStatus == XmVALID_DROP_SITE)
                RectHighlight(w, rect);
            break;

        case XmCR_DROP_SITE_LEAVE_MESSAGE:
            ...
```

17.4.2.2 Autoscrolling (or Pause) Drag

Using the **XmNdragProc** routine, drag and drop can support *pause drag*. This is when, during a drag, a dragged object is held over a drop site (without releasing), with some drag-under effect resulting. The most common form of this is when a scrolled window will scroll in some direction while a dragged object is held over the border of the window, or in the scrollbar.

Since this feature depends on the **XmNdragProc** routine, it is only available with the dynamic protocol, or, with the preregister protocol, when the initiator and receiver share the same top-level shell.

This feature is fully implemented in the Motif ScrolledWindow widget. That widget has a resource called **XmNautoDragModel**, which, when set to **XmAUTO_DRAG_ENABLED**, will allow autoscroll drag. This is the default setting. The scrolling characteristics are set with the **XmNinitialDelay**, **XmNrepeatDelay**, and the *XmNincrementFactor* resources of that widget. Please refer to the documentation for **XmScrolledWindow** in the *Motif 2.1—Programmer's Reference* for more information about these resources.

17.5 Drag Initiator Responsibilities for Dragging

This section explains what an application can do on behalf of the widget that initiates the drag. The drag initiator can do any of the following:

- If a widget does not already contain drag code, an application must recognize the start of a drag (Btn2Down) within a widget controlled by the application.
- If an application wants a widget to act as a drag source, the application must attach an **XmNconvertCallback** to the widget.
- If a widget does not already contain drag code, an application can establish a DragContext for the widget, providing information about operations, targets, and drag-over visuals, using the **XmeDragSource** function. If a widget does already contain drag code, an application can customize the DragContext.
- Optionally, an application can provide special drag-over effects.

These steps are described in the following sections.

If the user tries to drag objects from a widget that is not recognized as a drag source by either the toolkit or the source application, nothing happens.

17.5.1 Recognizing a Drag Has Started

The initiator client must be able to recognize the Btn2Down event within a widget it allows to be a drag source. It may have to override an already-assigned translation for the widget.

The following example from the main routine of **simple_drag** in Section 17.2.1.2 overrides the existing mouse button 2 translation for the ScrollBar widget, and maps it to the **StartDrag** routine, which will start the drag transaction.

```
char    dragTranslations[] = "#override <Btn2Down>: StartDrag()";
static  XtActionsRec  dragActions[] =
   {
    {"StartDrag", (XtActionProc)StartDrag}
   };
XtTranslations            parsed_xlations;
...
/* override button two press to start a drag */
parsed_xlations = XtParseTranslationTable(dragTranslations);
XtAppAddActions(app_context, dragActions, XtNumber(dragActions));

/* create a scroll bar widget */
    ScrollBar1 = XtVaCreateManagedWidget("SB1",
        xmScrollBarWidgetClass, RC1,
        XmNorientation, XmHORIZONTAL,   /* for cosmetic reasons */
        XmNtranslations, parsed_xlations,
        NULL);
```

Translation may be more complicated in some editable widgets, in which Btn2DownBtn2Up is used for primary transfer, and Btn2Motion is used for drag and drop.

17.5.2 Establishing an XmNconvertCallback Procedure

If your widget is to act as a drag initator, it must provide an **XmNconvertCallback** procedure that converts target requests. Your application must attach an **XmNconvertCallback** to the drag initiator widget; for example:

```
/* Associate a convert callback with the ScrollBar. */
  XtAddCallback(ScrollBar1, XmNconvertCallback, ConvertCallback, NULL);
```

The **XmNconvertCallback** procedure must be able to convert two kinds of requests:

- A request to convert _MOTIF_EXPORT_TARGETS. The callback procedure must respond to this request by providing a list of all the targets that the callback procedure knows how to convert. For example, in the in the **simple_drag** demo, the only target that the **ConvertCallback** procedure can convert the value to is *COMPOUND_TEXT*. Therefore, the list of targets that **ConvertCallback** returns consists solely of the *COMPOUND_TEXT* target.

- A request to convert all of the targets in the list returned by _MOTIF_EXPORT_TARGETS. For example, the **ConvertCallback** of **simple_drag** must be able to return the current ScrollBar **XmNvalue** in *COMPOUND_TEXT* format.

17.5.3 Starting a Drag With XmeDragSource

XmeDragSource initiates a drag and creates an **XmDragContext** widget. In the **simple_drag** demo, **XmeDragSource** is called as follows from the **StartDrag** routine:

```
void
StartDrag(Widget w,
          XEvent *event)
{
 Arg       args[2];
 Cardinal  n=0;

  XtSetArg(args[n], XmNdragOperations, XmDROP_COPY); n++;
  XmeDragSource(w, NULL, event, args, n);
```

StartDrag is called when the user preses Btn2Down on the **XmScrollBar**.

See the *Motif 2.1—Widget Writer's Guide* for syntactic details about **XmeDragSource**. In brief, **XmeDragSource** is the UTM replacement for the **XmDragStart** call. The *args* passed to **XmeDragSource** will eventually become resources of the DragContext.

If **XmDisplay** has the **XmNdragStartCallback** resource, the nominated procedure is called at the beginning of **XmeDragSource**. This allows the client to cancel the drag initiation, as well as perform any other necessary operation.

The **XmNdragOperations** resource lists all the operations that the initiator will support for this drag source, combined by the bitwise OR operation (|). During a drag, the toolkit compares this list with the receiver's **XmNdropSiteOperations** list and the user-selected operation to arrive at the operation that will be performed if a drop occurs on this site.

If an application wants to use only one operation, it should set the **XmNdragOperations** resource to just that operation to ensure that the correct operation and drag icon are chosen by the toolkit during the drag and drop transaction.

This drag source does not have any custom drag icons or any drag callbacks.

17.5.4 Overriding Existing Drag Sources

XtGetValues is used to check the values of widgets resources established as drag sources earlier in the application, and **XtSetValues** is used to update these values. The widget ID used is the DragContext, not the source widget ID, so that the change applies only to the widget during the drag.

If the widget is a predefined drag source (for example, **XmText**), overriding the default behavior becomes more complex. The widget calls **XmeDragSource** when the drag starts, and the application cannot call **XmeDragSource** again for the widget. Instead, it must update the existing DragContext. First it must find the DragContext for the widget, then establish the new behavior. One possible means to accomplish this is as follows:

- Override the existing Btn2Down translation with a new translation that calls the widget's action and also an action supplied by the application. For the Text widget, this new translation might look as follows:

```
<Btn2Down>: process-bdrag() my-drag-start()
```

- Register the new action, using **XtAppAddActions**.
- In the new action procedure, call **XmGetDragContext** to get the DragContext, and then call **XtSetValues** to change resource values. The timestamp argument to **XmGetDragContext** can be the timestamp from the event passed to the action routine.

For instance, Text allows the Copy and Move operations. If an application can support only Copy, it must update the DragContext's **XmNdragOperations** resource.

17.5.5 Drag-Over Visual Effects

When the user moves the mouse, a drag icon representing the object being dragged moves around the screen instead of the usual pointer. As the icon is dragged over portions of the screen, the icon may change to show the status of a possible drop. These drag-over visual effects help the user know how to proceed with the drag.

There are four ways to provide drag-over visual effects:

- Use the default drag-over visuals, specified in the **XmScreen** object. The toolkit manages all the drag-over effects.
- Put custom icons and pixmaps in the **XmScreen** visual resources to be used as default icons for all drag and drop transactions running on that **XmScreen**. The toolkit manages all the drag-over effects using these new icons. These resources can be modified by the application or the user in a resource file.
- Put custom icons and pixmaps in the DragContext visual resources for source, state, or operation icons. The application must monitor the state of the drag using the drag callbacks and update the DragContext icon values as necessary. The default icons specified in the **XmScreen** object are used only if the value for the equivalent DragContext visual resource is NULL.
- Manage the drag-over effects entirely in the application by drawing directly to the screen. The toolkit is not used, nor are the **XmScreen** and DragContext visual resources.

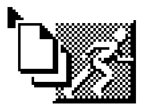
If the application provides custom icons and they are unsuitable for some reason, the toolkit defaults to the **XmScreen** drag-over visuals.

The drag icon consists of a source icon, optionally combined with a state icon and an operation icon.

Each drag icon has a hotspot. Since a drag icon could be quite large, the hotspot provides a single pixel that is used in providing drag-over and drag-under effects. For example, if the drag icon moves into the area of a valid drop site, neither the drag icon or the drop site will provide visual clues until the hotspot has moved into the area. By default, the hotspot is the upper left corner of the state icon.

In the following illustration, the running figure is the source icon, the state icon is the arrow in the corner, and the operation icon shows a Copy will happen if a drop is made. The default blending and attachment values are used.

Figure 17–5. A Drag Icon



17.5.5.1 Source Icon

The source icon is a picture representing the object being dragged. It can be either a pixmap or cursor. The client can create a custom pixmap through the **XmNsourcePixmapIcon** resource or a custom cursor through the **XmNsourceCursorIcon** resource. If these resources are NULL or not usable (too large, not a bitmap, or created on a different screen, for example), the default cursor given in the **XmScreen** resource **XmNdefaultSourceCursorIcon** is used.

For example, the following code establishes the **Pixmap** stored in the *pixmap* variable as the custom pixmap:

```
/* Create a drag icon using "pixmap" as the DragIcon pixmap. */
  XtSetArg(wargs[n],XmNpixmap,pixmap); n++;
  drag_icon = XmCreateDragIcon((Widget)cw,"dragIcon",wargs,n);

/* Set the XmNsourcePixmapIcon resource to the drag icon widget */
  n = 0;
  XtSetArg(wargs[n],XmNsourcePixmapIcon,drag_icon); n++;
```

The pixmap icon is used with the preregister visual style. The colormap is based on the source widget. The cursor icon is used for the dynamic visual style.

The following illustration shows the default source icons for general purpose, List, Label, and Text widgets.

Figure 17–6. Source Icons

17.5.5.2 State Icon

The state icon is a cursor that indicates if the drag is over a valid drop site, invalid drop site, or no drop site. The default state icons are in the **XmScreen** resources **XmNdefaultValidCursorIcon**, **XmNdefaultInvalidCursorIcon**, and **XmNdefaultNoneCursorIcon**.

A custom state icon can be specified in the DragContext resource **XmNstateCursorIcon**. This icon must be changed appropriately as the state of the drag changes, using the drag callbacks to change the CursorForeground resources outlined below. If **XmNstateCursorIcon** is NULL, not a bitmap, or not defined on the same screen as **XmScreen**, the default **XmScreen** icons are used.

The default state icon for all three states is an arrow, usually shown at the upper left corner of the operation icon.

Three DragContext resources can be used to change the color of the drag icon based on the state of the drag: **XmNvalidCursorForeground**, **XmNinvalidCursorForeground**, **XmNnoneCursorForeground**. This allows visual feedback about the drag to the user, without changing the icon shape. For example, the following lines in a resource file would make the drag icon green when it was over a valid drop site, red when it was over an invalid drop site, and yellow when it was not over any drop site:

```
*.validCursorForeground:          green
*.invalidCursorForeground:        red
*.noneCursorForeground:           yellow
```

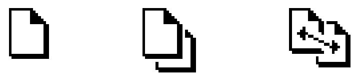
17.5.5.3 Operation Icon

The operation icon is a cursor that indicates what operation is to happen when the drop is made. The default operation icons are the values of the **XmScreen** resources **XmNdefaultMoveCursorIcon**, **XmNdefaultCopyCursorIcon**, and **XmNdefaultLinkCursorIcon**.

A custom operation icon can be specified in the DragContext resource **XmNoperationCursorIcon**. The icon should be changed as the operation changes, using the drag callbacks. If **XmNoperationCursorIcon** is NULL, not a bitmap, or not defined on the same screen as **XmScreen**, the default **XmScreen** icons are used.

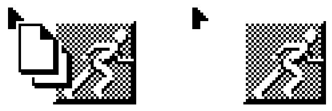
The following illustration shows the default Copy, Link, and Move operation icons.

Figure 17–7. Operation Icons



If the operation in effect is **XmDROP_NOOP**, meaning that no operation is possible, then the operation icon is left blank, as shown in the following illustration. This condition also sets the **dropSiteStatus** to **XmDROP_SITE_INVALID**.

Figure 17–8. Copy and Noop Drag Icons



17.5.5.4 Drag Icon Blending and Attachment

The client can specify which of the three icons to mix together to form the drag icon with the **XmNblendModel** DragContext resource:

XmBLEND_ALL
: Use the source icon, state icon, and operation icon. The hotspot comes from the state icon. This is the default value. The order listed is also the order of the blend.

XmBLEND_STATE_SOURCE
: Use only the source icon and state icon. The hotspot comes from the state icon.

XmBLEND_JUST_SOURCE
: Use only the source icon. The hotspot comes from the source icon.

XmBLEND_NONE
: Do not display any drag icon. The client handles all drag-over effects.

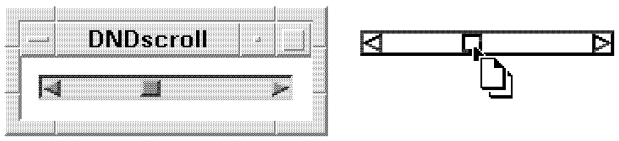
The **XmNattachment** DragIcon resource specifies where the state and operation icons will be placed on the source icon. The default placement is both the state and operation icons at the attachment point of the source icon, with the operation icon on top. The default value is **XmATTACH_NORTH_WEST**.

XmNoffsetX and **XmNoffsetY** are used to place the icon relative to the attachment point.

If the attachment point is **XmATTACH_HOT**, the state and operation icons are attached to the source icon at a point the same x and y distance from the upper left corner of the source icon as the pointer is from the upper left corner of the widget containing the source. This attachment style is particularly useful when the application makes a custom source icon that exactly reflects the source widget at the time the drag starts.

In the following illustration, the custom source icon is an outline of the scrollbar. When the drag was started, the pointer was on the slider. The operation and state icons are placed at the same location on the source icon.

Figure 17–9. An Attach_Hot Icon



When the state or operation icon is blended with a source icon, a specified point of the icon's **XmNpixmap** is aligned with the upper left corner of the source icon. The resulting **XmNpixmap** is large enough to include both, and the resulting **XmNmask** has 1 bits wherever either the source icon or source mask did.

If a dynamic cursor style is being used, and the resulting blended cursor is too large for the screen, the blending is done with the **XmScreen XmNdefaultSourceCursorIcon** instead of the DragContext's **XmNsourceCursorIcon**. If it is still too large, it is clipped relative to the hotspot (that is, if the hotspot is at an edge, the other edge is clipped; if the hotspot is in the center, opposite edges are clipped equally).

17.5.5.5 Visual Style Notes

If **XmNsourcePixmapIcon** is used, the colormap used for rendering is that of the DragContext's reference widget.

If the DragContext **XmNblendModel** is **XmBLEND_NONE**, and the dynamic cursor style is in use, the application must use **XChangeActivePointerGrab** to change the cursor. If **XmBLEND_NONE** is specified, and the preregister cursor style is in use, the application can render the cursor directly onto the screen, saving and restoring the image underneath.

The cursor style can change as the pointer moves from window to window. An application can tell which style is in use by looking at the **dragProtocolStyle** field in the **XmNtopLevelEnterCallback** structure, or looking at the **XmNdragInitiatorProtocolStyle** Display resource in the case of **XmDRAG_NONE** or **XmDRAG_DROP_ONLY**.

The resolution and best cursor size can vary from screen to screen. This is why the default cursor icons are **XmScreen** resources. An application that wants its source cursor or pixmap to be screen dependent can look for changes in the *screen* field in the **XmNtopLevelEnterCallback** struct, and update the various icon DragContext resources appropriately.

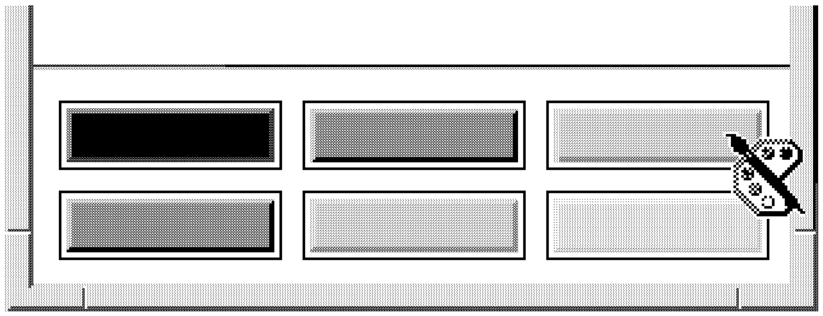
17.5.5.6 Creating a Drag Icon

Any of the three parts of a drag icon can be customized: the source icon, the state icon, and the operation icon.

Use the **XmCreateDragIcon** function to create any of these parts. The **XmNattachment** resource is not used for the source icon. The other resources specify pixmap, size, and hotspot details. The DragContext **XmNblendModel** resource indicates which hotspot is used for the entire drag icon.

The following example from **DNDDemo.c** creates a source icon from a bitmap. The source icon is the palette and the state icon is the paintbrush. (Actually, the state icon is not shown when the drag starts, because the blend style is **XmBLEND_JUST_SOURCE**. It is shown here as if the blend style were **XmBLEND_ALL**.)

Figure 17–10. Custom Source Icon



The **ColorRect** function is called when a drag starts from one of the color rectangles in the lower portion of the window. Among its other duties, it establishes the drag icon from source bits from the **DNDDraw.c** file.

```
void
ColorRect(Widget w, XEvent *event, String *params, Cardinal *num_params)
{
    ...
    Atom        targets[1];
    Widget      sourceIcon, stateIcon;
    Pixel       background, foreground;
    char        *source_bits, *source_mask;
    char        *state_bits, *state_mask;
    Dimension   width, height;
    Arg         args[16];
    int         n = 0;

    n = 0;
    XtSetArg(args[n], XmNbackground, &background); n++;
    XtSetArg(args[n], XmNforeground, &foreground); n++;
    XtGetValues(w, args, n);

    /* If the server will handle a large icon, create one */
    if (appInfo->maxCursorWidth >= ICON_WIDTH &&
        appInfo->maxCursorHeight >= ICON_HEIGHT) {
```

```
        source_bits = (char *)SOURCE_ICON_BITS;
        source_mask = (char *)SOURCE_ICON_MASK;
        state_bits = (char *)STATE_ICON_BITS;
        state_mask = (char *)STATE_ICON_MASK;
        width = ICON_WIDTH;
        height = ICON_HEIGHT;

    }
    else {

        /* If the server will handle a small icon, create one */
        source_bits = (char *)SMALL_SOURCE_ICON_BITS;
        source_mask = (char *)SMALL_SOURCE_ICON_MASK;
        state_bits = (char *)SMALL_STATE_ICON_BITS;
        state_mask = (char *)SMALL_STATE_ICON_MASK;
        width = SMALL_ICON_WIDTH;
        height = SMALL_ICON_HEIGHT;

    }

    /* Create the drag cursor icons */
    sourceIcon = GetDragIconFromBits(w, source_bits, source_mask, width,
                                     height, background, foreground);

    stateIcon = GetDragIconFromBits(w, state_bits, state_mask, width,
                                    height, background, foreground);

    /* Setup the arglist for the drag context that is created
     * at drag start */
    n = 0;
    ...
    /* set args for the drag cursor icons */
    XtSetArg(args[n], XmNsourceCursorIcon, sourceIcon); n++;
    XtSetArg(args[n], XmNstateCursorIcon, stateIcon); n++;

    /* identify the necessary callbacks */
    ...
    /* start the drag. This creates a drag context. */
    myDC = XmeDragSource(w, NULL, event, args, n);
```

```
}
```

The **GetDragIconFromBits** function turns the bits into a bitmap.

```
static Widget
GetDragIconFromBits(Widget w, char *bits, char *mask, Dimension width,
                    Dimension height, Pixel background, Pixel foreground)
{

    Pixmap     icon, iconMask;
    Display    *display = XtDisplay(w);

    icon = XCreateBitmapFromData(display, DefaultRootWindow(display),
                                                  bits, width, height);

    iconMask = XCreateBitmapFromData(display, DefaultRootWindow(display),
                                       mask, width, height);

    return(GetDragIcon(w, icon, iconMask, width, height,
                       background, foreground));

}
```

The **GetDragIcon** function uses the bitmap created by the **GetDragIconFromBits** function to create a drag icon.

```
static Widget
GetDragIcon(Widget w, Pixmap icon, Pixmap iconMask, Dimension width,
            Dimension height, Pixel background, Pixel foreground)
{

    Widget  dragIcon;
    Arg     args[10];
    int     n = 0;

    XtSetArg(args[n], XmNhotX, ICON_X_HOT); n++;
    XtSetArg(args[n], XmNhotY, ICON_Y_HOT); n++;
    XtSetArg(args[n], XmNwidth, width); n++;
    XtSetArg(args[n], XmNheight, height); n++;
    XtSetArg(args[n], XmNmaxWidth, appInfo->maxCursorWidth); n++;
    XtSetArg(args[n], XmNmaxHeight, appInfo->maxCursorHeight); n++;
    XtSetArg(args[n], XmNbackground, background); n++;
    XtSetArg(args[n], XmNforeground, foreground); n++;
    XtSetArg(args[n], XmNpixmap, icon); n++;
    if (iconMask != XmUNSPECIFIED_PIXMAP &&
        iconMask != None) {
      XtSetArg(args[n], XmNmask, iconMask); n++;
    }
    dragIcon = XmCreateDragIcon(w, "dragIcon", args, n);

    return(dragIcon);
}
```

17.5.6 Drag Callbacks

Callbacks notify the initiator of how the drag is proceeding. The receiver's **XmNdragProc** (if any) is first notified of the action and given a chance to update the *operation*, *operations*, and **dropSiteStatus** fields in its callback structure. The new values are available to the initiator's drag callback in the appropriate callback structure.

These drag callbacks are all optional. They enable the initiator to monitor the progress of the drag and manage its visual effects accordingly. Otherwise, the toolkit on the initiator side handles the drag-over effects.

XmNdragMotionCallback
: Called when the drag icon is in motion

XmNoperationChangedCallback
: Called when the user requests a different operation be performed on the drop than was previously in effect

XmNdropSiteEnterCallback
: Called when the drag icon enters a drop site

XmNdropSiteLeaveCallback
: Called when the drag icon leaves a drop site

XmNtopLevelEnterCallback
: Called when the drag icon enters a top-level window or root window (when changing screens)

XmNtopLevelLeaveCallback
Called when the drag icon leaves a top-level window or root window (when changing screens)

Callback structures for these routines contain information about the drag. The callback structures for **XmNdragMotionCallback**, **XmNoperationChangedCallback**, and **XmNdropSiteEnterCallback** contain the *operations*, *operation*, and **dropSiteStatus** fields (among others), which are initialized by the toolkit before the callback is called.

The *operations* field lists all operations possible for a drop on the current site, whether the site is registered as a DropSite or not. The toolkit initializes the *operations* field as follows:

- If the receiver's **XmNdragProc** was called, the value of *operations* is the list of operations common to the value of the **XmNdragProc**'s *operations* field at the end of **XmNdragProc** and the DropSite's **XmNdropSiteOperations** list.
- If the **XmNdragProc** routine was not called but the user selected an operation, *operations* is set to that operation if it is in the **XmNdragOperations** list. If it is not in the list, *operations* is set to **XmDROP_NOOP**.
- Otherwise, the *operations* field is initialized to the list in the DragContext's **XmNdragOperations** resource.

The *operation* field shows the operation that will occur if a drop happens at the current cursor location. It is initialized as follows:

- If the receiver's **XmNdragProc** was called, *operation* is initialized to the value of *operation* at the end of the **XmNdragProc**.
- If the **XmNdragProc** routine was not called but the pointer is in or entering an active drop site, the toolkit initializes *operation* by taking the following steps, in order of precedence from highest to lowest:
    1. If Move is in both the *operations* field and the DropSite's **XmNdropSiteOperations** list, *operation* is set to **XmDROP_MOVE**.
    2. If Copy is in both the *operations* field and the DropSite's **XmNdropSiteOperations** list, *operation* is set to **XmDROP_COPY**.
    3. If Link is in both the *operations* field and the DropSite's **XmNdropSiteOperations** list, *operation* is set to **XmDROP_LINK**.
    4. Otherwise, *operation* is set to **XmDROP_NOOP**.

- Otherwise, the toolkit initializes *operation* by taking the following steps, in order of precedence from highest to lowest:
  1. If Move is in the *operations* field, *operation* is set to **XmDROP_MOVE**.
  2. If Copy is in the *operations* field, *operation* is set to **XmDROP_COPY**.
  3. If Link is in the *operations* field, *operation* is set to **XmDROP_LINK**.
  4. Otherwise, *operation* is set to **XmDROP_NOOP**.

The **dropSiteStatus** field in the callback structure indicates if the drag icon is over a valid drop site, an invalid drop site, or no drop site. The callback procedure can use this information to display the appropriate drag-over visuals. The toolkit initializes the **dropSiteStatus** field as follows:

- If the pointer is over an active drop site:
  - — If the receiver's **XmNdragProc** was called, **dropSiteStatus** is initialized to the value of **dropSiteStatus** at the end of the **XmNdragProc** procedure.
  - — If the **XmNdragProc** routine was not called but the initiator and receiver have at least one target and one operation in common, **dropSiteStatus** is initialized to **XmDROP_SITE_VALID**.
  - — Otherwise, **dropSiteStatus** is initialized to **XmDROP_SITE_INVALID**.
- If the pointer is not over an active drop site, **dropSiteStatus** is initialized to **XmNO_DROP_SITE**.
- If the *operation* field is **XmDROP_NOOP**, **dropSiteStatus** is initialized to **XmDROP_SITE_INVALID**.

If the application has not stored the DragContext ID in a global location, these callbacks can find the DragContext ID by passing the **timeStamp** field from the callback structure to the **XmGetDragContext** function.

This example shows a callback that is called when a new drop site is entered. It checks the validity of the drop site, and uses one of three custom source icons, depending on the status.

```
static void EnterCB(w, client_data, call_data)
Widget          w;
XtPointer       client_data, call_data;
{
   XmDragContext                  dc;
   XmDropSiteEnterCallback        EnterData;
   Cardinal                       n;
   Arg                            args[MAX_ARGS];

   dc = (XmDragContext)w;
   EnterData = (XmDropSiteEnterCallback )call_data;

   n = 0;

   if (EnterData->dropSiteStatus == XmVALID_DROP_SITE) {
      XtSetArg(args[n], XmNsourceCursorIcon,
                 GetValidIcon(w)); n++;
      XtSetValues(dc, args, n);
      }
   if (EnterData->dropSiteStatus == XmINVALID_DROP_SITE) {
      XtSetArg(args[n], XmNsourceCursorIcon,
                 GetInvalidIcon(w)); n++;
      XtSetValues(dc, args, n);
      }
   if (EnterData->dropSiteStatus == XmNO_DROP_SITE) {
      XtSetArg(args[n], XmNsourceCursorIcon,
                 GetNeutralIcon(w)); n++;
      XtSetValues(dc, args, n);
      }
}
```

If a drag callback is desired, it is added to the DragContext’s callback resources. The following example adds a callback named **EnterCB** that is performed when the pointer enters an active drop site:

```
Widget       dc;

   dc = XmeDragSource(w, location_data, event, args, n);
   XtAddCallback(dc, XmNdropSiteEnterCallback, EnterCB,
                   NULL);
```

17.5.7 Getting Data about the Current Drop Site

The initiator can find information about the current drop site with the **XmDropSiteRetrieve** function. It must pass in the DragContext so that the toolkit knows what drop site the request is for. The initiator can find the value of any drop site resource except the callback routines.

The following example gets the number and list of import targets for a drop site. The example shows a drop site enter callback, but it could be in any of the initiator's drag callbacks.

```
XmDropSiteEnterCallback      DragData;
...
n = 0;
XtSetArg(args[n], XmNimportTargets, &importTargets); n++;
XtSetArg(args[n], XmNnumImportTargets,
           &numImportTargets); n++;
XmDropSiteRetrieve(DragData->DragContext, args, n);
```

17.5.8 Cancelling the Drag

The drag in progress can be cancelled in either of two ways. Both ways are treated the same by the toolkit.

- The user can press osfCancel.
- The initiator can call the **XmDragCancel** function if it decides the drag should not continue for some reason.

The initiator is notified of the cancel by the **XmNdropStartCallback** with a **dropAction** field value of **XmDROP_CANCEL**.

The receiver is notified by a **XmCR_DROP_SITE_LEAVE_MESSAGE** message. This message is processed by the **XmNdragProc** in the dynamic protocol mode. This allows any drag-under effects to be undone.

17.6 Drop Receiver Responsibilities for Dropping

The drop receiver's responsibilities for dropping are the same as any widget's responsibilities as the destination widget in a UTM transfer. See Chapter 16 for more details.

This chapter explains how a drop receiver can provide help to the user.

17.6.1 Providing Help

It might not always be obvious to the user what the result of dropping a particular source on a drop site might be. The user can request more information about the drop site by pressing osfHelp while the drag icon is over the drop site.

When the user presses osfHelp, UTM starts a drop by doing the following:

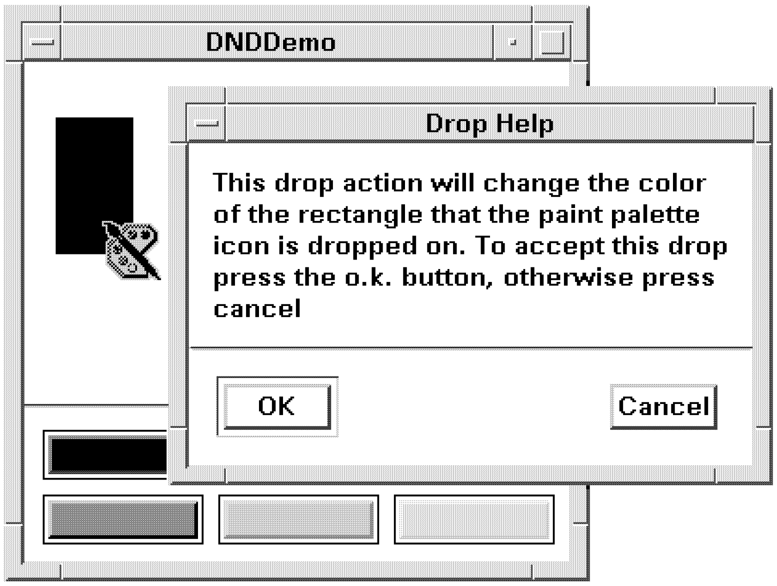
- Sets the **dropAction** member of the **XmDropProcCallbackStruct** to **XmDROP_HELP**.
- Sets the *operation* member of the **XmDestinationCallbackStruct** to **XmOTHER**.

When the user presses osfHelp, your **XmNdestinationCallback** procedure must realize a DialogBox. This DialogBox must provide help about the drop site, and must give users the choice of either continuing or cancelling the drop. While the DialogBox is realized, your **XmNdestinationCallback** procedure must neither perform any transfers nor call *XmTransferDone*. Your **XmNdestinationCallback** procedure must save the **transfer_id** member for any later *XmTransferValue* calls to be made.

After the user accepts or rejects the transfer, the processing continues from the callbacks on the dialog buttons. In the case of accepting the transfer, the transfer may be continued by using the **transfer_id** in calls to all the normal transfer calls in UTM. In the case of rejecting the transfer, the programmer *must* call *XmTransferDone* with a status of **XmTRANSFER_DONE_FAIL**. This will clean up the internal state of UTM and provide drag and drop zap effects to indicate the failure to the user.

The following example taken from **DNDDemo.c** shows how the help dialog shown in the illustration was created.

Figure 17–11. Help Dialog



This drop action will change the color of the rectangle that the paint palette icon is dropped on. To accept this drop press the o.k. button, otherwise press cancel

The **handleDestination** routine examines the *operation* member of the **XmDestinationCallbackStruct**. If *operation* holds **XmOTHER**, then the **HandleHelp** routine is called:

```
void
handleDestination(Widget w, XtPointer client, XtPointer call)
{
  XmDestinationCallbackStruct *cs = (XmDestinationCallbackStruct *) call;

  if (appInfo->highlightRect != NULL)
    RectUnhighlight(w);

  if (cs->operation != XmOTHER)
    HandleDrop(w, call, NULL);
  else
    HandleHelp(w, call, NULL);
}
```

The **HandleHelp** routine displays the help dialog. The text presented in the dialog depends on the drop site and on the requested operation. Callback routines are registered to be performed when either of the dialog pushbuttons is pressed.

```
static void
HandleHelp(Widget w, XtPointer call, XtPointer ignore)
{
  XmDestinationCallbackStruct *cs = (XmDestinationCallbackStruct *) call;
  XmDropProcCallbackStruct *ds =
    (XmDropProcCallbackStruct *) cs -> destination_data;
  Boolean                       rectFound, bgFound, pixFound;
  XmString                      helpStr;
  RectPtr                       rect;
  Arg                           args[5];
  XmString                      tempStr, buttonArray[2];
  int                           n = 0;

  savedCB = cs;

  /* if we haven’t created a help dialog, create one now */
  if (helpDialog == NULL) {
    XtSetArg(args[n],XmNdialogStyle,
                           XmDIALOG_FULL_APPLICATION_MODAL); n++;
    XtSetArg(args[n], XmNtitle, "Drop Help"); n++;
    helpDialog = XmCreateMessageDialog(topLevel, "Help", args, n);

    n = 0;
    buttonArray[0] = XmStringCreateSimple("Move");
    buttonArray[1] = XmStringCreateSimple("Copy");
    XtSetArg(args[n], XmNbuttons, buttonArray); n++;
    XtSetArg(args[n], XmNbuttonCount, 2); n++;
    XtSetArg(args[n], XmNbuttonSet, 0); n++;
    XtSetArg(args[n], XmNsimpleCallback, ChangeOperation); n++;
    tempStr = XmStringCreateSimple("Operations:");
    XtSetArg(args[n], XmNoptionLabel, tempStr); n++;
    helpMenu = XmCreateSimpleOptionMenu(helpDialog,
                                           "helpMenu", args, n);
    XmStringFree(tempStr);
    XmStringFree(buttonArray[0]);
    XmStringFree(buttonArray[1]);
    XtAddCallback(helpDialog, XmNokCallback,
                  (XtCallbackProc) HandleDrop, NULL);
    XtAddCallback(helpDialog, XmNcancelCallback,
                  (XtCallbackProc) CancelDrop, NULL);
    XtUnmanageChild(XmMessageBoxGetChild(helpDialog,
                                              XmDIALOG_HELP_BUTTON));
    XtRealizeWidget(helpDialog);
  }

  /* find the valid targets */
  CheckTargets(ds->dragContext, XtDisplay(w), &rectFound,
               &bgFound, &pixFound);

  /* determine the appropriate help message */
  if (rectFound) {
    if (ds->operations == XmDROP_MOVE | XmDROP_COPY) {
      XtManageChild(helpMenu);
      helpStr = XmStringCreateLtoR(HELP_MSG4, XmFONTLIST_DEFAULT_TAG);
      XtManageChild(XmMessageBoxGetChild(helpDialog,
                                                 XmDIALOG_OK_BUTTON));
    } else if (ds->operation == XmDROP_MOVE) {
      XtUnmanageChild(helpMenu);
      helpStr = XmStringCreateLtoR(HELP_MSG2, XmFONTLIST_DEFAULT_TAG);
      XtManageChild(XmMessageBoxGetChild(helpDialog,
                                                 XmDIALOG_OK_BUTTON));
    } else if (ds->operation == XmDROP_COPY) {
      XtUnmanageChild(helpMenu);
      helpStr = XmStringCreateLtoR(HELP_MSG3, XmFONTLIST_DEFAULT_TAG);
      XtManageChild(XmMessageBoxGetChild(helpDialog,
                                                 XmDIALOG_OK_BUTTON));
    }
  } else if (bgFound || pixFound && ds->operation == XmDROP_COPY) {
    XtUnmanageChild(helpMenu);
    rect = RectFind(ds->x, ds->y);
    if (rect) {
      helpStr = XmStringCreateLtoR(HELP_MSG1, XmFONTLIST_DEFAULT_TAG);
      XtManageChild(XmMessageBoxGetChild(helpDialog,
                                                XmDIALOG_OK_BUTTON));
    } else {
      helpStr = XmStringCreateLtoR(HELP_MSG5, XmFONTLIST_DEFAULT_TAG);
      XtUnmanageChild(XmMessageBoxGetChild(helpDialog,
                                           XmDIALOG_OK_BUTTON));
    }
  } else {
    XtUnmanageChild(helpMenu);
    helpStr = XmStringCreateLtoR(HELP_MSG5, XmFONTLIST_DEFAULT_TAG);
    XtUnmanageChild(XmMessageBoxGetChild(helpDialog,
                                               XmDIALOG_OK_BUTTON));
  }

  /* set the help message into the dialog */
  XtSetArg(args[0], XmNmessageString, helpStr);
  XtSetValues(helpDialog, args, 1);

  /* Free the XmString */
  XmStringFree(helpStr);

  /* map the help dialog */
  XtManageChild(helpDialog);
}
```

If the user selects the *OK* button to continue the drop, the program calls **HandleDrop** again. If the user selects the **Cancel** button, the program calls the **CancelDrop** routine. This routine calls *XmTransferDone* as follows to cancel the drop:

```
static void
CancelDrop(Widget w, XtPointer call, XtPointer ignore)
{
  XmDestinationCallbackStruct *cs;

  /* For drop help we save the XmDestinationCallbackStruct in a global
     variable named savedCB. */
  cs = savedCB;

  XmTransferDone(cs -> transfer_id, XmTRANSFER_DONE_FAIL);
}
```

17.7 Drag Initiator Responsibilities for Dropping

The drag initiator

- Registers an **XmNconvertCallback** procedure to format data and send the formatted data to the receiver.
- Optionally, registers an **XmNdropStartCallback** to be performed at the drop.
- Optionally, registers an **XmNdropFinishCallback** to be performed after the drop and transfer have finished.
- Optionally, registers an **XmNdragDropFinishCallback** to be performed after the entire drag and drop transaction has finished.

17.7.1 XmNdropStartCallback

The receiver's **XmNdropProc** routine receives the drop message first if the drop occurred over a widget that was registered as a drop site. It verifies that a drop is possible, and updates fields in its callback structure. These fields become available to the initiator in its **XmNdropStartCallback** callback structure. The initiator can perform any actions necessary before the information is transferred; for example, providing a new drag icon.

The toolkit initializes the *operation*, *operations*, and **dropSiteStatus** fields as described in Section 17.5.5, with one difference: the initialization for the drag callbacks uses the values at the end of the receiver's **XmNdragProc**, while the initialization for the drop callbacks uses the values at the end of the receiver's **XmNdropProc**.

The **dropAction** field indicates the action that the receiver has taken. **XmDROP** shows that a normal drop is in progress. **XmDROP_CANCEL** shows that the receiver has cancelled the drop. If the action is **XmDROP_HELP**, the initiator is not expected to do anything, although this callback provides the opportunity to do so if desired (for example, changing the drag icon to reflect the Help request).

This procedure will not know the resolution of the help dialog. However, if the user chooses to continue, the initiator's **XmNconvertProc** routine is called as part of the transfer process and, if the user chooses to cancel, the initiator's **XmNdropFinishCallback** is called with a **dropAction** of **XmDROP_CANCEL**.

17.7.2 Dealing with Requests for Transfer

As of Motif Release 2.0, the role of the **XmNconvertProc** can be done more easily through the UTM **XmNconvertCallback** procedures. Nevertheless, we provide the information in this subsection for those maintaining older drag and drop applications.

The drag initiator must register a callback to process transfers in the **XmNconvertProc** DragContext resource. This routine is called when the receiver client invokes **XmDropTransferStart**. Before calling **XmDropTransferStart**, the receiver makes a list of the target formats it wants.

The initiator's **XmNconvertProc** callback routine processes transfer requests from the receiver. The routine should be able to return information about each object being dragged in each possible target format for that item.

If the DropTransfer **XmNincremental** resource is True, information is transferred between the initiator and the toolkit using the Xt selection incremental protocol. If the value is False, the information is transferred between the initiator and the toolkit in one pass. The initiator and receiver need not be using the same incremental or nonincremental protocol.

The **XmNconvertProc** routine is called for each target type desired by the receiver, a single target type for each request. The **XmNconvertProc** routine should be able to perform any of the operations listed in the DragContext's **XmNdragOperations** resource on data in any of the target types listed in the **XmNexportTargets** resource:

- If the operation is Copy or Link, the **XmNconvertProc** returns a pointer to the data. The receiver will use this pointer to copy this data into its own storage, or establish a link using this pointer.
- If the operation is Move, the first transfer request has a normal *target* type. The **XmNconvertProc** routine should return a pointer to the data, as it would for a Copy.

  A second transfer request for the data has a *target* type of DELETE. The receiver does not issue this request until it has received the data and handled it appropriately (such as storing it in a file). Only then should the initiator delete the data.

17.7.3 XmNdropFinishCallback

The **XmNdropFinishCallback** is called when the receiver’s **XmNtransferProc** routine has finished processing all the transfers desired by the receiver.

The **completionStatus** field indicates whether the entire drop was successful or not.

The *operations*, *operation*, **dropSiteStatus**, and **dropAction** fields are initialized as for the **XmNdropStartCallback** procedure.

17.7.4 XmNdragDropFinishCallback

The *XmdragDropFinishCallback* routine is performed when the complete drag and drop transaction has finished. This routine is called immediately after the initiator’s **XmNdropFinishCallback** has finished. The initiator frees any remaining structures it has allocated during the drag.

The following sample code from **DNDDemo.c** removes the icons when the drop is complete:

```
static void
ColorDragDropFinishCB(Widget w, XtPointer client, XtPointer call)
{
    Widget  sourceIcon;
    Widget  stateIcon = (Widget) client;
    Arg     args[1];

    XtSetArg(args[0], XmNsourceCursorIcon, &sourceIcon);
    XtGetValues(w, args, 1);

    XtDestroyWidget(sourceIcon);
    XtDestroyWidget(stateIcon);
}
```

Chapter 18

MWM and the ICCCM

The X Consortium Standard *Inter-Client Communication Conventions Manual* (ICCCM) defines standards by which X clients should communicate with each other. The Motif toolkit and MWM comply with ICCCM. Applications may define private protocols for communicating with other applications that share those protocols. If they do so, they should also conform to ICCCM standards.

18.1 Window Managers, ICCCM, and Shells

ICCCM defines protocols for communication between clients and window managers. Most of the communication takes place through properties on an application's top-level windows. The window manager can also generate events that are available to the application.

In Motif and Xt, shells handle most communication between an application and a window manager. An application seldom has to deal directly with properties or events. The application can usually specify properties by setting resources of a shell. Shells also select for and handle most events from the window manager.

This section discusses the relations between some shell resources, properties, and events concerned with communication between an application and any window manager. The following section discusses resources, properties, and events that apply to MWM in particular.

18.1.1 Application Startup

When a top-level window is mapped, the window manager may search the resource database for information about the window. The resource name and class come from the WM_CLASS property for the window. This property contains two consecutive strings that identify the instance and class names.

Xt sets the WM_CLASS property when a shell that is a subclass of **WMShell** is realized. The instance name is the name of the shell. For an ApplicationShell, this is generally the name of the application passed to **XtDisplayInitialize**. The class name is the application class from the highest-level widget in the hierarchy. For an ApplicationShell, this is generally the application class passed to **XtDisplayInitialize**. If the root widget is not an ApplicationShell, the class name is the widget's class name.

Most window managers display a name for a top-level window, often in a title bar. The window name comes from the WM_NAME property. This property is a string whose encoding is identified by the type of the property.

A Motif application specifies a window name using the WMShell resources **XmNtitle** and **XmNtitleEncoding**. If the shell is a TopLevelShell subclass and the **XmNiconName** resource is not NULL, the value of that resource is the default for **XmNtitle**. Otherwise, the default title is the name of the shell. For a dialog, an application can supply a title as the value of the BulletinBoard resource **XmNdialogTitle**.

XmNtitleEncoding is an atom representing the encoding of the name. The default title encoding depends on whether or not a language procedure has been set. If no language procedure has been set, the default is STRING. If a language procedure has been set, the title is assumed to be in the encoding of the locale and is passed to **XmbTextListToTextProperty** with an encoding style of **XStdICCTextStyle**. The returned property is used as the WM_NAME property. If the title is fully convertible to type STRING, the encoding is STRING; otherwise, the encoding is COMPOUND_TEXT.

18.1.2 Window Configuration

A window manager can assign any position and size to a window. The user and application can supply preferred positions and sizes, but the window manager is free to use or ignore these as it wishes.

The user generally specifies position and size using the –**geometry** option when invoking the command that starts the application. In Motif, the value specified for –**geometry** becomes the value of the Shell **XmNgeometry** resource. An application should never set this resource itself; it should reserve it for the user. An application specifies size and position by supplying values for the Core resources **XmNx**, **XmNy**, **XmNheight**, **XmNwidth**, and **XmNborderWidth**. When an x, y, width, or height value is specified for both **XmNgeometry** and one of the specific geometry resources, the value from **XmNgeometry** takes precedence.

The MWM **positionIsFrame** resource determines whether MWM interprets x and y values as referring to the upper left corner of the client window itself or the upper left corner of the frame that MWM puts around the client window. By default x and y values refer to the frame.

When a top-level window is mapped, MWM uses the following order of precedence in determining size and position:

- If the user specifies position and size using the –**geometry** option, MWM uses those values.
- If the MWM **interactivePlacement** resource is True, MWM waits for the user to select a position using a button press for the upper left corner of the window. If the user drags the pointer down and to the right with the mouse button pressed, the user can then determine the size of the window by releasing the mouse button. If the user does not determine a size in this way, MWM uses the window's **XmNwidth** and **XmNheight**.
- If the MWM **usePPosition** resource is True, or if **usePPosition** is *nonzero* and the window's **XmNx** or **XmNy** is nonzero, MWM uses the window's **XmNx** and **XmNy** to position the window. MWM uses the window's **XmNwidth** and **XmNheight** for the window's size. If the MWM **positionOnScreen** resource is

True and if the window would be completely off the screen, MWM alters the window position so that at least part of the window is on the screen.

- If the MWM **clientAutoPlace** resource is True, MWM positions the window with its top left corner offset horizontally and vertically from the last client mapped. MWM uses the window's **XmNwidth** and **XmNheight** for the window's size.
- MWM positions the window in the upper left corner of the screen and uses the window's **XmNwidth** and **XmNheight** for the window's size.

Before a window is mapped, the application communicates additional position and size information to the window manager through the WM_NORMAL_HINTS property on the window. This property is of type WM_SIZE_HINTS and contains a number of fields derived from WMShell resources:

XmNminHeight, **XmNminWidth**
Specifies the minimum height and width that the application wants the widget's window to have. If an initial value is supplied for one of these resources but not for the other, the value of the unspecified resource is set to 1 when the widget is realized. If no value is specified for either resource, MWM uses the values from **XmNbaseHeight** and **XmNbaseWidth** if specified. Otherwise, MWM uses a minimum height and width of at least 1.

XmNmaxHeight, **XmNmaxWidth**
Specifies the maximum height and width that the application wants the widget's window to have. If an initial value is supplied for one of these resources but not for the other, the value of the unspecified resource is set to 32767 when the widget is realized. If the MWM resource **maximumClientSize** is specified, MWM uses that value to determine the maximum window size. Otherwise, MWM uses the maximum height and width from the WM_NORMAL_HINTS property, except that the window size may not exceed the height and width specified by the MWM **maximumMaximumSize** resource.

XmNbaseHeight, **XmNbaseWidth**
Specifies the base for a progression of preferred heights and widths for the window manager to use in sizing the widget. The preferred heights are **XmNbaseHeight** plus integral multiples of **XmNheightInc**, with a minimum of **XmNminHeight** and a maximum of **XmNmaxHeight**. The preferred widths are **XmNbaseWidth** plus integral multiples of **XmNwidthInc**, with a minimum of **XmNminWidth** and a maximum of

XmNmaxWidth. If an initial value is supplied for one of these resources but not for the other, the value of the unspecified resource is set to 0 when the widget is realized. If no value is specified for either resource, MWM uses the values from **XmNminHeight** and **XmNminWidth** if specified. Otherwise, MWM uses a base height and width of at least 1.

XmNheightInc, **XmNwidthInc**
Specifies the increment for a progression of preferred heights and widths for the window manager to use in sizing the widget. The preferred heights are **XmNbaseHeight** plus integral multiples of **XmNheightInc**, with a minimum of **XmNminHeight** and a maximum of **XmNmaxHeight**. The preferred widths are **XmNbaseWidth** plus integral multiples of **XmNwidthInc**, with a minimum of **XmNminWidth** and a maximum of **XmNmaxWidth**. If an initial value is supplied for one of these resources but not for the other, the value of the unspecified resource is set to 1 when the widget is realized. If no value is specified for either resource, MWM uses an increment of 1.

XmNminAspectX, **XmNminAspectY**
Specifies the numerator and denominator of the minimum aspect ratio (X/Y) that the application wants the widget's window to have. If no value is specified for either resource, MWM imposes no minimum aspect ratio.

XmNmaxAspectX, **XmNmaxAspectY**
Specifies the numerator and denominator of the maximum aspect ratio (X/Y) that the application wants the widget's window to have. If no value is specified for either resource, MWM imposes no maximum aspect ratio.

XmNwinGravity
Specifies the window gravity for use by the window manager in positioning the widget. If no initial value is specified, the value is set when the widget is realized. If **XmNgeometry** is not NULL, **XmNwinGravity** is set to the window gravity returned by **XWMGeometry**. Otherwise, **XmNwinGravity** is set to **NorthWestGravity**.

After a window is mapped, an application can request changes to window size or position by calling **XtSetValues** for one or more of the Core geometry resources. A user can generally employ window manager facilities to move or resize a top-level window.

Calling **XtSetValues** for a geometry resource generates a geometry request that may propagate up the widget hierarchy to the shell. This may cause the shell to make its own geometry request, and this invokes the shell's **root_geometry_manager** procedure. This procedure uses **XConfigureWindow** to ask the window manager to change the window's size or position.

If a window manager responds to a configuration request by denying it or by moving the window without resizing it, the window manager sends a synthetic **ConfigureNotify** event. If the window is resized, the window receives a real **ConfigureNotify** event.

These events may be handled by either the **root_geometry_manager** procedure or a Shell event handler. If the VendorShell resource **XmNuseAsyncGeometry** is True, the **root_geometry_manager** procedure does not wait for the window manager to respond to the configuration request, but instead returns **XtGeometryYes**. If the WMShell resource **XmNwaitForWm** is True and if the window manager grants the configuration request within the **XmNwmTimeout** interval, the **root_geometry_manager** procedure updates the widget's geometry resources and returns **XtGeometryYes**. Otherwise, the **root_geometry_manager** procedure returns **XtGeometryNo** and relies on the event handler to reconfigure the widget when it receives a subsequent **ConfigureNotify** event.

The shell's **ConfigureNotify** event handler is invoked when the user reconfigures a top-level window or when the application reconfigures a window and this reconfiguration is not handled by the **root_geometry_manager** procedure. The event handler updates the shell's core geometry fields with the values allowed by the window manager. If the size of the shell changes, the event handler calls the shell's *resize* procedure. This procedure calls **XtResizeWidget** to change the height, width, and border width of the child to be the same as those of the shell.

18.1.3 Icons

An application uses several properties to communicate with the window manager about icons associated with top-level windows. A Motif application can use resources of several Shell subclasses to specify values for these properties.

When a window is first mapped, it can appear in either its normal state or iconic state. An application uses a field in the WM_HINTS property to tell the window manager which initial state it prefers. A Motif application specifies the initial state by setting the WMShell resource **XmNinitialState** or the TopLevelShell resource **XmNiconic**. **XmNiconic** takes precedence over **XmNinitialState**. After a window is realized, an application can use **XtSetValues** for **XmNiconic** to either iconify or deiconify the window.

An application can supply a name, a bitmap, or a window for the window manager to use as an icon. When a top-level window is in iconic state, the window manager usually displays the icon window if one is supplied, or else the icon pixmap if one is supplied, or else the icon name. MWM uses the **iconDecoration** resource in determining what aspects of an icon to display.

The icon name comes from the WM_ICON_NAME property. Like WM_NAME, this property is a string whose encoding is identified by the type of the property.

A Motif application specifies an icon name using the TopLevelShell resources **XmNiconName** and **XmNiconNameEncoding**. The default icon name is the name of the shell. **XmNiconNameEncoding** is an atom representing the encoding of the name. The default encoding depends on whether or not a language procedure has been set. If no language procedure has been set, the default is STRING. If a language procedure has been set, the icon name is assumed to be in the encoding of the locale and is passed to **XmbTextListToTextProperty** with an encoding style of **XStdICCTextStyle**. The returned property is used as the WM_ICON_NAME property. If the icon name is fully convertible to type STRING, the encoding is STRING; otherwise, the encoding is COMPOUND_TEXT.

An application uses fields in the WM_HINTS property to supply an icon bitmap and an optional mask for displaying the bitmap in a nonrectangular shape. A Motif application specifies an icon bitmap as the value of the WMShell resource **XmNiconPixmap**, and it specifies the mask as the value of the WMShell resource **XmNiconMask**.

An application uses a field in the WM_HINTS property to supply an icon window. A Motif application specifies an icon window as the value of the WMShell resource **XmNiconWindow**. The icon window must be an InputOutput child of the root window. It must also use the root visual and the default colormap of the screen. The application must not map, unmap, or configure this window. It must, however, select for Expose events on the window and redisplay the contents when it receives these events.

The window manager may specify preferred maximum and minimum sizes and size increments for icon bitmaps and windows. To do this it puts a WM_ICON_SIZE property on the root window. MWM uses the **iconImageMaximum** and **iconImageMinimum** resources, with increments of 1, in setting this property. Before an application specifies an icon bitmap or window, it should use the Xlib routine **XGetIconSizes** to check these constraints and then supply a bitmap or window that is of one of the preferred sizes.

An application can use two fields of the WM_HINTS property to supply preferred x and y root coordinates for the icon location. A Motif application specifies these coordinates as the values of the WMShell resources **XmNiconX** and **XmNiconY**. The window manager may ignore these values. MWM uses the **useIconBox**, **iconPlacement**, and **iconPlacementMargin** resources in determining where to place icons.

18.1.4 Window Groups

An application can use a field of the WM_HINTS property to supply the window ID of a window to serve as the "leader" for a group of windows. The window manager may treat all windows in this group as a whole for certain purposes, such as showing a single icon when the entire group is iconified.

A Motif application specifies a window group leader as the value of the WMShell resource **XmNwindowGroup**. For VendorShell and its subclasses, if the shell has a parent, Motif sets the **XmNwindowGroup** to the parent’s window at the time that the shell and its parent are both realized. Otherwise, the default value is **XtUnspecifiedWindowGroup**, which means that no window group is set.

18.1.5 Menus and Dialogs

A window manager may treat dialogs differently from other top-level windows, and it must not interfere with menus at all.

An application tells a window manager not to decorate or otherwise interfere with a window by setting the **override_redirect** attribute of the window to True. A Motif application does this by setting the **Shell** resource **XmNoverrideRedirect** to True, or by using an **OverrideShell**, which has a default value of True for this resource. **XmMenuShell** is a subclass of **OverrideShell**, and MenuShells are the only widgets that should have a value of True for **XmNoverrideRedirect**. An application normally does not supply a value other than the default for this resource.

An application tells a window manager to treat a window as transient or secondary by setting the window's WM_TRANSIENT_FOR property. This property contains the window ID of another top-level window, usually the window from which the transient window was popped up. A Motif application generally specifies this property by creating a DialogShell, a subclass of TransientShell, which has an **XmNtransientFor** resource. The value is a widget, and the default is set to the shell's parent at the time that both the shell and its parent are realized. The window of the **XmNtransientFor** widget is used for the WM_TRANSIENT_FOR property. For a shell that is not a subclass of TransientShell, an application can set the WMShell **XmNtransient** resource to True. The **XmNwindowGroup** is then used for the WM_TRANSIENT_FOR property. An application normally does not supply a value other than the default for **XmNtransient** or **XmNtransientFor**.

MWM treats transient windows differently from other top-level windows. By default it keeps transient windows stacked on top of their primary windows and does not allow transient windows to be iconified separately from their primary windows. The MWM **transientDecoration** and **transientFunctions** resources determine which decorations and functions apply to transient windows. An application can further specify these decorations and functions by using the VendorShell **XmNmwmDecorations** and **XmNmwmFunctions** resources, explained in Section 18.2.

18.1.6 Input Focus

ICCCM recognizes four models for the relationship between clients and window managers in setting input focus:

No input	The client does not expect keyboard input and does not want the window manager to set focus to any of its windows.
Passive input	The client expects keyboard input and wants the window manager to set focus to its top-level window. It does not set focus itself.
Locally active input	The client expects keyboard input and wants the window manager to set focus to its top-level window. It may also set focus to one of its subwindows when one of its windows already has the focus. It does not set focus itself when the current focus is in a window that the client does not own.

Globally active input	The client expects keyboard input but does not want the window manager to set focus to any of its windows. Instead, it sets focus itself, even when the current focus is in a window that the client does not own.

An application tells the window which model it prefers by using two properties:

- If the *input* field of the WM_HINTS property is True, the application wants the window manager to set focus to its top-level window. If this field is False, the application does not want the window manager to set focus.
- If the WM_PROTOCOLS property contains a WM_TAKE_FOCUS atom, the application sometimes sets focus itself. If the WM_PROTOCOLS property does not contain a WM_TAKE_FOCUS atom, the application does not set focus itself.

These combinations are summarized in the following table:

Table 18–1. Input Models

Input Model	**Input field**	WM_TAKE_FOCUS
No input	False	Absent
Passive	True	Absent
Locally active	True	Present
Globally active	False	Present

A window manager generally does not set input focus to a window when the WM_HINTS *input* field is False. A window with a WM_TAKE_FOCUS protocol may receive a ClientMessage when the window manager wants the window to accept keyboard focus. The window may respond by setting the input focus or by ignoring the message.

A Motif application can set the *input* field of the WM_HINTS property by specifying a value for the WMShell resource **XmNinput**. The application can install the WM_TAKE_FOCUS atom on the WM_PROTOCOLS property by calling **XmAddWMProtocols** or **XmAddWMProtocolCallback**, explained in Section 18.3.

A Motif application normally should avoid setting input focus itself. The application can control the location of focus within its subwindows by using the VendorShell resource **XmNkeyboardFocusPolicy**, the Gadget, Primitive, and Manager resource **XmNtraversalOn**, and the **XmProcessTraversal** routine. If the application wants a widget to receive no input at all, it can use **XtSetSensitive** to make the widget insensitive. If the application needs to set focus directly, it should usually use **XtSetKeyboardFocus** and avoid using **XSetInputFocus**. For more information, see Chapter 13.

A number of MWM resources influence keyboard focus. When **keyboardFocusPolicy** is "explicit" (the default), the user must press Btn1 on a window or its decoration to give it focus. When **keyboardFocusPolicy** is "pointer", the window that contains the pointer has the focus. With an explicit policy, other resources determine whether a window has focus when it is first mapped (**startupKeyFocus**), deiconified (**deiconifyKeyFocus**), or raised (**raiseKeyFocus**). When **autoKeyFocus** is True, and the window with focus is iconified or withdrawn, focus passes to the window that last had focus. When **enforceKeyFocus** is True, MWM sets focus to globally active windows.

18.1.7 Colormaps

An application can create and set colormaps for its windows, but only the window manager should install colormaps. Each window manager has a colormap focus policy that determines which top-level window has the colormap focus at a given time. When a window has colormap focus, the window manager installs one or more colormaps associated with that window.

If all windows in an application use the same colormap, the application need take no special action to tell the window manager to use that colormap. The window manager keeps track of the colormap attribute for each top-level window and installs that colormap when the window has colormap focus.

If an application uses different colormaps for some windows in its hierarchy, it must tell the window manager about those colormaps by setting a WM_COLORMAP_WINDOWS property on the top-level window. This property is a list of windows whose colormaps the window manager should install when the top-level window has colormap focus. The list should be in order of priority, with the windows whose colormaps the application would most like to have installed listed first. The application can use **XSetWMColormapWindows** to set this property.

On many servers, only one hardware colormap can be installed at a time. This may cause colors in windows that use different colormaps to be displayed incorrectly when their own colormaps are not installed. To reduce contention for colormaps, applications should use the facilities for standard colormaps described in *Xlib—C Language X Interface*.

The MWM **colormapFocusPolicy** resource determines the colormap focus policy. When the value is "keyboard", the window with keyboard focus has the colormap focus. When the value is "pointer", the window under the pointer has the colormap focus, regardless of whether that window also has keyboard focus. When the value is "explicit", the colormap focus changes only when the user invokes the **f.focus_color** function.

When a window with colormap focus has a WM_COLORMAP_WINDOWS property, the user can install the next and previous colormaps on the list by invoking the **f.next_cmap** and **f.prev_cmap** functions.

18.1.8 Application Shutdown and Restart

An application may run under a session manager with facilities for saving and restoring the state of the application. An application communicates with a session manager by placing WM_COMMAND and WM_CLIENT_MACHINE properties on its top-level windows. WM_COMMAND contains a string that would restart the client in its current state.

A Motif application should have only one non-NULL WM_COMMAND property for each logical application (that is, for each ApplicationShell hierarchy). Xt sets the WM_COMMAND property for an ApplicationShell when the shell is realized, using the command that started the application. Note that if an application is using an unrealized ApplicationShell with multiple TopLevelShell popup children, Xt will not place a WM_COMMAND property on any window, and the application must put this property on some (possibly unmapped) window in the application.

WM_CLIENT_MACHINE contains a string that represents the name of the host on which the application is running. Xt sets the WM_CLIENT_MACHINE for a WMShell or subclass when the shell is realized.

A session manager can inform an application when a top-level window is about to be deleted or when the application should try to save its state. An application expresses interest in these notifications by adding a WM_DELETE_WINDOW atom or a WM_SAVE_YOURSELF atom to the WM_PROTOCOLS property.

If a WM_DELETE_WINDOW protocol exists, the session manager sends a ClientMessage when it wants to delete a top-level window. The application may ask for user confirmation and may decide to comply or not comply with the request. If it decides to comply, the application can either unmap or destroy the window.

If a WM_SAVE_YOURSELF protocol exists, the session manager sends a ClientMessage when it wants the application to save its current state in such a way that it could be restored. The application should do whatever is necessary to save its internal state and then update the non-NULL WM_COMMAND property with a command that will restart the application in its current state. Finally, the application updates the WM_COMMAND property on the window that has the WM_SAVE_YOURSELF protocol if it has not already done so. This informs the session manager that the application has finished saving its state.

Motif installs a WM_DELETE_WINDOW protocol for VendorShell and its subclasses. It also installs a procedure to be called after any application-supplied WM_DELETE_WINDOW handlers are invoked. This procedure destroys the widget, unmaps the window, or does nothing, depending on the value of the VendorShell resource **XmNdeleteResponse**. If the procedure destroys an ApplicationShell, it then exits the application.

An application can add its own WM_DELETE_WINDOW and WM_SAVE_YOURSELF protocols by using **XmAddWMProtocols** or **XmAddWMProtocolCallback**.

When the user invokes the **f.kill** command, MWM sends a ClientMessage if an application has a WM_DELETE_WINDOW protocol and a separate ClientMessage if an application has a WM_SAVE_YOURSELF protocol. If the application has no WM_DELETE_WINDOW protocol, the **f.kill** command kills the client. In this case, if a WM_SAVE_YOURSELF protocol exists, MWM sends the ClientMessage and then waits for the time specified by the **quitTimeout** resource before killing the client.

18.2 MWM Properties and Resources

In addition to the properties and protocols described in ICCCM, Motif uses properties and protocols of its own. A Motif application usually specifies these properties using VendorShell and BulletinBoard resources.

18.2.1 Decorations

A Motif application expresses preferences for MWM window decorations by supplying a value for the VendorShell resource **XmNmwmDecorations**. The value is the bitwise inclusive OR of one or more flag bit constants, each of which indicates a preference for or against a particular decoration. If a value has been supplied for this resource, MWM displays only those decorations specified by both **XmNmwmDecorations** and the MWM **clientDecoration** resource (for primary windows) or specified by both **XmNmwmDecorations** and the MWM **transientDecoration** resource (for transient windows). If no value has been supplied for **XmNmwmDecorations**, MWM displays the decorations specified by the **clientDecoration** or **transientDecoration** resource.

18.2.2 Functions

A Motif application expresses preferences for MWM window functions by supplying a value for the VendorShell resource **XmNmwmFunctions**. The value is the bitwise inclusive OR of one or more flag bit constants, each of which indicates a preference for or against a particular function. If a value has been supplied for this resource, MWM displays only those functions specified by both **XmNmwmFunctions** and the MWM **clientFunctions** resource (for primary windows) or specified by both **XmNmwmFunctions** and the MWM **transientFunctions** resource (for transient windows). If no value has been supplied for **XmNmwmFunctions**, MWM displays the functions specified by the **clientFunctions** or **transientFunctions** resource.

BulletinBoard may change the initial value of **XmNmwmFunctions** if its parent is a subclass of VendorShell. The BulletinBoard resource **XmNnoResize** determines whether the decorations of the VendorShell parent include resize controls.

18.2.3 Input Mode

An application can inform MWM that it should impose constraints on which windows can obtain input. A Motif application does this by supplying a value for the VendorShell resource **XmNmwmInputMode**. For a BulletinBoard whose parent is a DialogShell, the application can set **XmNmwmInputMode** indirectly by specifying a value for the BulletinBoard resource **XmNdialogStyle**.

The possible modes are as follows:

Modeless	Input goes to any window.
Primary application modal	Input does not go to ancestors of this window or their descendants.
Full application modal	Input goes to this window or its descendants and to other applications but not to other windows in this application.
System modal	Input goes only to this window or its descendants.

18.2.4 Window Menu

An application can supply items for MWM to add to the end of the window menu for a window by specifying a value for the _MOTIF_WM_MENU property. A Motif application does this by supplying a value for the VendorShell resource **XmNmwmMenu**. The window menu itself is the value of the MWM **windowMenu** resource.

18.2.5 MWM Messages

An application can specify a message for MWM to send the application when the user invokes the **f.send_msg** function. The application places a _MOTIF_WM_MESSAGES atom on the WM_PROTOCOLS property for the window. The application also places an atom on the _MOTIF_WM_MESSAGES property. When the **f.send_msg** function is invoked with this atom as the argument, MWM sends the application a ClientMessage. The application can use **XmAddWMProtocols** to place a _MOTIF_WM_MESSAGES atom on the WM_PROTOCOLS property, and it can use **XmAddProtocolCallback** to place an atom on the _MOTIF_WM_MESSAGES property and associate it with a routine to be called when MWM sends the ClientMessage.

18.2.6 MWM Information

MWM maintains a _MOTIF_WM_INFO property on the root window of each screen it manages. This property is available for applications to inspect but not to change. The **XmIsMotifWMRunning** routine examines this property when determining whether or not MWM is running.

18.3 Atom and Protocol Management

Xlib provides two atom utility routines. **XInternAtom** returns an existing atom or (if the third argument is False) creates and returns an atom that matches the given string. **XGetAtomName** returns the string that matches the given atom.

Motif has a number of routines to help an application install protocol atoms and handle ClientMessages sent when the protocols are invoked. These routines maintain an internal registry of properties, protocol atoms associated with the properties, and callback routines associated with the protocol atoms. The application can use these routines with shells that are subclasses of VendorShell.

XmAddProtocols associates one or more protocol atoms with a property for a given shell. If the shell is realized, it adds those protocols to the property for the shell’s window. If the shell is not realized, it arranges for the protocols to be added to the property and for a ClientMessage event handler to be added at the time the shell is realized. **XmAddWMProtocols** is a specialized version that adds protocols for the WM_PROTOCOLS property.

XmAddProtocolCallback adds a callback routine to a callback list associated with a protocol. It calls **XmAddProtocols** if the protocol has not yet been registered. When the protocol manager’s ClientMessage event handler receives a ClientMessage for the protocol, it invokes the procedures on the associated callback list. The first argument to each callback procedure is the shell associated with the protocol. The second argument is the client data, if any, specified in the call to **XmAddProtocolCallback**. The third argument is a pointer to an **XmAnyCallbackStruct** structure whose *reason* member is **XmCR_PROTOCOLS** and whose *event* member is a pointer to the ClientMessage event. In the ClientMessage event, the **message_type** member is the property that contains the protocol, the *format* member is 32, and the **data.l[0]** member is the protocol atom. **XmAddWMProtocolCallback** is a specialized version of **XmAddProtocolCallback** that adds a callback for a protocol on the WM_PROTOCOLS property.

An application can also use **XmSetProtocolHooks** to specify a routine to be called before or after a callback list is invoked for a protocol. **XmSetWMProtocolHooks** is a specialized version that adds prehooks and posthooks for a protocol on the WM_PROTOCOLS property.

Once an application has registered a protocol and optional callback routines, it can make the protocol active or inactive. A protocol is active if it has been added to the associated property for the window. A protocol is inactive if it has been removed from the associated property. **XmActivateProtocol** makes a registered protocol active, and **XmDeactivateProtocol** makes a protocol inactive. **XmActivateWMProtocol** and **XmDeactivateWMProtocol** are specialized versions that activate or inactivate a protocol on the WM_PROTOCOLS property.

XmRemoveProtocolCallback removes a callback routine from the callback list associated with the protocol. **XmRemoveProtocols** removes one or more protocols and all callbacks associated with those protocols from the internal registry. If the shell is realized, it removes those protocols from the associated property. **XmRemoveWMProtocolCallback** and **XmRemoveWMProtocols** are specialized versions that remove callbacks or protocols for the WM_PROTOCOLS property.

Chapter 19

Printing

This chapter describes how application developers can use the X Print Service and printing-related APIs to achieve printing from their applications.

The X printing architecture is fundamentally concerned with reusing most of the X Protocol and X library APIs with an X Server that generates such printer languages as PostScript and PCL (refer to the specification, *X Printing Architecture*). Correspondingly, the model chosen for widget printing is to re-use the code that renders the widgets on the video side (through their **expose** method) for the X Print Server side.

An application therefore needs only to create new instances of widgets using a **Shell** widget created on the X Print Server and to set resources according to what it is to be printed on paper. In addition, depending on the programming model chosen for rendering, the application may instruct the widgets to render themselves manually or use callbacks to change the state of the widget content between pages.

As a result of that design choice, there is no clear boundary between printing only the content of the widgets and printing high-resolution screen dumps of the widget's visual representation. The separation between current content and current visual is left to the programmer, and so widget printing becomes mostly a matter of the programmer deciding which widget resources need to be copied from the video widget instance to the print widget instance. A section in this chapter is intended to help the programmer answer this question for the Motif **XmText** and **XmLabel** widget classes.

As far as rendering the widget on the print side is concerned, two models of programming are supported:

- Synchronous, where the application directs the widget to print itself (using the **XmRedisplayWidget** API)
- Asynchronous, where all drawing/printing happens as a result of **Exposure** and other **Print** events being generated by the X Print Server and automatically dispatched through the **XmPrintShell** callback mechanism (**XmNpageSetUpCallback**).

In addition, a convenience function to set up the X Print Server connection and a top-level **Print** shell widget are provided, as well as a convenience function supporting a print-to-file service. Both APIs are also usable from Motif/CDE applications that do not print using widgets but print by means of direct rendering via **Xlib**.

19.1 The Printing APIs

The printing APIs are as follows:

- **XmPrintSetup**
- **XmPrintShell**
- **XmPrintPopupPDM**
- **XmGetScaledPixMap**
- **XmRedisplayWidget**
- **XmPrintToFile**

This section summarizes the APIs and gives examples of their use.

XmPrintSetup registers an X Print Server connection with Xt, sets resources appropriately, and creates an **XmPrintShell** that it returns to the caller. In this example, the print shell is created in the **OKcallback** of a **DtPrintSetupBox**:

```
void OKCallback(Widget w, XtPointer client_data, XtPointer call_data)
{
    DtPrintSetupCallbackStruct *pbs = call_data;
    static Widget shell;
    static int num;          /* app data to print */

    if (!shell) {
      shell =
          XmPrintSetup(widget,
                       XpGetScreenOfContext(pbs->print_display,
                                            pbs->print_context)
                       "Print", NULL, 0);
      XtAddCallback(p->print_shell, XmNpageSetupCallback,
                    PageSetupCB, &num);
      XtAddCallback(p->print_shell, XmNpdmNotificationCallback,
                    PdmNotifyCB, &num);
    }

    num = 0;

    /* check print-to-file & print */
}
```

XmPrintShell encapsulates some of the X Print Service functionalities and provides the framework for the asynchronous printing programming model. Here is an example of a **XmNpageSetupCallback**:

```
void PageSetupCB(Widget w, XtPointer client_data, XtPointer
                 call_data)
{
   Widget print_shell = widget;
   XmPrintShellCallbackStruct* pr_cbs = call_data;
   int *num = client_data;
   static Widget form, text;
   char buf[10];

   /* don't do anything after last page is printed *
   if (pr_cbs->last_page) return;

   /* create the widgets once */
   if (!form) {
      form = XmCreateForm("form", print_shell, NULL, 0);
      text = XmCreateTextField("text", form, NULL, 0);
   }

   /* setup app's data in print text widget */
   sprintf(buf, "%d", *num);
   XmTextSetString(text, buf);

   num++
   if (num > 9) pr_cbs->last_page = True;
}
```

XmPrintPopupPDM sends a notification to start a print dialog manager on behalf of the application. An example:

```
if (XmPrintPopupPDM(print_shell, top_level) !=
     XmPDM_NOTIFY_SUCCESS)
 {
     /* let user know of problem */
 }
```

XmGetScaledPixMap retrieves a **Pixmap** from an *XBM* or *XPM* file and potentially scales it to match the printer resolution. Again, this example is from a **XmNpageSetupCallback**:

```
int *month = client_data;
static Widget form, label;
Pixmap pixmap;
unsigned long fg, bg;
int depth;

/* create the widgets once */
if (!form) {
   form = XmCreateForm("form", print_shell, NULL, 0);
   label1 = XmCreateLabel("label", form, NULL, 0);
}

XtVaGetValues(label, XmNforeground, &fg,
              XmNbackground, &bg, XmNdepth, &depth);
```

```
/* since label is a child of a XmPrintShell, and we have
   specified 0 as the scaling factor, it will be scaled
   appropriately.  Also, the cache will take care of calling
   this more than once */
pixmap = XmGetScaledPixmap(label,
                           ((*month) ? "month.xpm": "week.xpm"),
                           fg, bg, depth, 0);
```

XmRedisplayWidget calls the **expose** method of a widget in order to draw its content; for example:

```
XmTextScroll(text, lines_per_page);
XmRedisplayWidget(text);
```

XmPrintToFile retrieves the data being sent by a Print Server and prints it to a file on the client side. What follows is the remaining part of the **OKCallback** above:

```
int save_data = XPSpool;

if (pbs->destination == DtPRINT_TO_FILE)
save_data = XPGetData;

/* start job must precede XpGetDocumentData in XmPrintToFile */
XpStartJob(XtDisplay(print_shell), save_data);

/* setup print to file */
if (pbs->destination == DtPRINT_TO_FILE)
{
   if (!XmPrintToFile(XtDisplay(print_shell),
                      pbs->dest_info, FinishPrintToFile, NULL))
      {
        /* output some kind of error message */
       XpCancelJob(XtDisplay(print_shell), True);

       /* go back to the event loop as if we had never printed */
          return;
      }
}
```

19.2 Motif Widget Printing

In this section, we examine the Motif widgets (mainly, **XmText** and **XmLabel**) and give hints about which widget resources need to be turned on or off in the printing case. This section also gives examples of sample code that copies data from a video widget to a corresponding print widget.

The assumption we make in this discussion is that we want to print the content of the widgets, rather than the visual representation of the widgets; in other words, our goal is not to print high-resolution screen dumps. Although it's impossible to define what constitutes the content of an arbitrary widget class (that is a sufficiently difficult task for the widgets we know), we can start by pointing out the resources that are important visually in Xt/Motif, in order to help the programmer set or unset them appropriately for printing.

19.2.1 Purely Visual Resources

At the **Core** class level, there are **XmNbackground**, which the application will probably want to set to white for paper output, and **XmNborderWidth**, which should be **0** (its default value).

For **XmPrimitive**, the **XmNshadowThickness** and **XmNhighlightThickness** resources are probably good candidates for **0** value as well, unless some three-dimensional effect is intended on the paper. **XmNforeground** should probably be black.

For the **XmText** and **XmTextField** widgets, one obvious setting is to turn off the blinking of the I-beam cursor. This happens automatically (refer to default dynamic and *XmNcursorPositionVisible* == False) if the widget is rooted to a **XmPrintShell**.

The print case almost certainly shouldn't print the scrollbars in a **ScrolledText** instance. The **XmNscrollHorizontal** and **XmNscrollVertical** resources can be set to **False** to suppress the printing of scrollbars. Note that, as a rule, it is better to avoid **ScrolledText** altogether in the case of printing.

For the **Text** widget, **XmMwordWrap** is a resource that deserves some attention; it's up to the programmer to decide what the application should do with it. If a wordwrap-on video text buffer is transferred to a print text buffer using (as is likely to be the case) a different character size, font, resolution, and so on (with wordwrap on as well), the number of pages will be different in the print and video cases. That's probably be fine since the number of pages on the video side should not be treated as content to be printed out: they are virtual pages, after all. If the video text has wordwrap off, the sequence of lines on the print side will be the same, and it's up to the application or its user to guarantee that this layout is appropriate for the print font, paper size, and so on, because no new formatting is done. The application can decide to take a wordwrap-on video text buffer and generate explicit newlines before passing it to the print text, so that the same layout is preserved. However, the application does so at the risk of allowing the content to run off the edge of the paper).

Finally, the **Text** widget also carries internal margins (**XmNmarginWidth** and **XmNmarginHeight**) that the print code might want to reset before printing.

For the **XmLabel** widget, the important visual resources are margins and alignment, since shadows and highlights are covered by **XmPrimitive**. **XmLabel** carries six margins resources (**XmNmarginWidth**, **XmNmarginHeight**, **XmNmarginTop**, **XmNmarginBottom**, **XmNmarginRight**, **XmNmarginLeft**) and the **XmNaligment** resource. These resources are mostly useful when the widget holds a border, highlight, or shadow of some kind. If borders are not to be transferred, there is no real need to transfer a margin either.

19.2.2 Content Resources

In connection with defining the content for **XmText** and **XmLabel**, the resource settings that the programmer would probably want to transfer to the print widget are the following: **XmNvalue** and **XmNvalueWcs** for the text widgets (or by using **XmTextGetString**/**XmTextSetString**). The **XmText** resource **XmNtopCharacter** may be relevant if the current page only is to be printed. **XmTextSetHighlight** can be used to highlight some text in a text widget.

For **XmLabel**, **XmNlabelType** (*PIXMAP* or *STRING*), and **XmNlabelString** or **XmNlabelPixmap** are important resources.

In both the label and text cases, the setting of **XmNstringDirection** is probably inherited from the environment, so it's probably best to leave it alone.

19.2.3 Examples

In the following examples, a simple widget initialize method wants to know whether it is creating a widget for printing or video.

```
Widget CopyText(print_parent, video_text...)
/*-------------*/
{
    /* get the content out of the video text */
    buffer = XmTextGetString (video_text);
     /* now create the print instance */
    ptext = XtVaCreateWidget(print_parent, ...,
                              XmNvalue, buffer, NULL);
    XtFree(buffer);
    return ptext;

}

Widget CopyLabel(print_parent, video_label...)
/*-------------*/
{
    /* get the string content out of the video label */
    XtVaGetValues(video_label, XmNlabelString, &buffer, NULL);
     /* now create the print instance */
    plabel = XtVaCreateWidget(print_parent, ...,
                               XmNlabelString, buffer, NULL);
    XtFree(buffer);
    return plabel;

}
```

Appendix A

The Motif Clipboard

Motif provides a set of routines for dealing with the CLIPBOARD selection. Although these routines are not obsolete, applications should use UTM routines, instead of these clipboard routines, whenever possible.

The Motif clipboard interface allows an application to assert ownership of the selection and request conversion of the selection. The interface stores the data in the selection and other information about the selection on the server. The owner can place the selection value in these server data structures either at the time it asserts ownership or at the time a client requests conversion.

By copying the selection value at the time it asserts ownership, an application can simplify conversion and make the data available for retrieval even if the owner is killed. By copying the selection value when a client requests it, an application can avoid converting data that no client may request. However, in this case, the application may need to make a copy of the data to be transferred. With either copying mechanism, the data is stored in the Motif clipboard's server data structures the first time a client requests the data.

A.1 Copying Data to the Clipboard

To assert ownership and copy data to the clipboard, an application takes these steps:

1. It calls **XmClipboardStartCopy** to begin the interaction.
2. It makes one or more calls to **XmClipboardCopy** to place data on the clipboard.
3. It terminates the interaction by calling **XmClipboardEndCopy** or **XmClipboardCancelCopy**.

An application begins an interaction to copy data to the clipboard by calling **XmClipboardStartCopy**. The application passes the following: a display pointer and timestamp; the ID of a window in the application; a compound string that could be used to label the data; and, if the application intends to delay copying the data until it is requested, a widget ID and a function to be called to convert the data. **XmClipboardStartCopy** returns in one of the arguments a data ID that the application must later pass to **XmClipboardEndCopy** or **XmClipboardCancelCopy**. The application must also pass the same window ID to subsequent clipboard calls in this sequence that it uses in the call to **XmClipboardStartCopy**.

After calling **XmClipboardStartCopy**, the application makes one or more calls to **XmClipboardCopy** to place data on the clipboard. Each call associates the data with a single target (called a format in the clipboard interface). The application can associate the same data or different data with more than one target, but it must do so by making separate calls to **XmClipboardCopy**.

If the application passes a NULL data buffer to **XmClipboardCopy**, it asserts that it intends to transfer the actual data for that target when a client requests it. Otherwise, **XmClipboardCopy** transfers data to be stored on the clipboard by **XmClipboardEndCopy**. If the application makes more than one call to **XmClipboardCopy** for the same target, the data is appended to the previously transferred data for that target.

XmClipboardCopy returns in one of its arguments a data ID that identifies the data and target specified in this call. An application that provides actual data at the time a client requests it uses this ID in its conversion routine to identify the data and target to be converted. Such an application must store a mapping of the data ID to the data and target after **XmClipboardCopy** returns.

The application terminates the interaction by calling either **XmClipboardEndCopy** or **XmClipboardCancelCopy**. **XmClipboardEndCopy** stores in the server data structures the data transferred by the calls to **XmClipboardCopy** during this interaction sequence. It also asserts ownership of the CLIPBOARD selection. If the application calls **XmClipboardCancelCopy** instead of **XmClipboardEndCopy**, the interaction is terminated without storing any of the transferred data or asserting ownership of the selection.

If a client later requests data that the owner has declared it would provide at the time of the request, the clipboard interface invokes the conversion routine that the owner registered in the call to **XmClipboardStartCopy**. This routine receives the following as arguments: the widget ID passed to **XmClipboardStartCopy**; the data ID for this data and target returned by **XmClipboardCopy**; a private ID the application may have supplied in the call to **XmClipboardCopy**; and a reason for invoking the routine.

The conversion routine is responsible for converting the data to the requested target. In order to do this it must consult the mapping it established between the data ID or the private ID and the data and target when it called **XmClipboardCopy**. Once the conversion routine has determined the proper target, it copies the data to the clipboard. To do this it calls **XmClipboardCopyByName**, using the data ID passed to the conversion routine. The application can call **XmClipboardCopyByName** more than once, if necessary, to convert all the data for this target.

Once an application has copied data to the clipboard in this way, it no longer asserts that it will convert the same data to the same target in the future. It can remove the data ID from its mapping of data IDs to data and targets, and it can free any data it has associated with this ID if it is not needed for any other purpose.

The clipboard interface calls the conversion routine when a data item intended for later conversion has been removed from the clipboard and is no longer needed. For example, another application may have copied new data to the clipboard. In this case, the conversion routine can remove the data ID from its mapping of data IDs to data and targets, and it can free any data it has associated with this ID if it is not needed for any other purpose. If the conversion routine is being called because an item has been removed from the clipboard, the *reason* argument to the conversion routine is **XmCR_CLIPBOARD_DATA_DELETE**. If the conversion routine is being called because a client has requested data conversion, the *reason* argument is **XmCR_CLIPBOARD_DATA_REQUEST**.

An application can use **XmClipboardWithdrawFormat** to rescind its assertion that it will convert data to a particular target on request.

XmClipboardUndoCopy removes the last item placed on the clipboard by an application using the same *display* and *window* arguments. This function also restores to the clipboard the item that was on the clipboard before the cancelled copy was done. If the application calls **XmClipboardUndoCopy** a second time, the function restores to the clipboard the item that was removed by the first call to **XmClipboardUndoCopy**.

A.2 Retrieving Data from the Clipboard

To retrieve data from the clipboard, an application takes these steps:

1. It calls **XmClipboardStartRetrieve** to begin the interaction.
2. It makes one or more calls to **XmClipboardRetrieve** to retrieve data from the clipboard.
3. It terminates the interaction by calling **XmClipboardEndRetrieve**.

An application begins an interaction to retrieve data from the clipboard by calling **XmClipboardStartRetrieve**. The application passes a display pointer, a timestamp, and the ID of a window in the application. The application must pass the same window ID to subsequent clipboard calls in this sequence that it uses in the call to **XmClipboardStartRetrieve**. **XmClipboardStartRetrieve** locks the clipboard.

After calling **XmClipboardStartRetrieve**, the application makes one or more calls to **XmClipboardRetrieve** to retrieve data from the clipboard, converted to a given target. The application passes **XmClipboardRetrieve** a buffer to receive the data. If this buffer is not large enough to contain all the data for the given target, **XmClipboardRetrieve** returns *XmClipboardTruncate*. The application can make repeated calls to **XmClipboardRetrieve** to retrieve the remainder of the data. The function **XmClipboardInquireLength** returns the length of the data on the clipboard for the given target. This allows the application to allocate a buffer of the correct size.

XmClipboardEndRetrieve unlocks the clipboard and ends the interaction.

A.3 Utility Routines

The Motif clipboard interface has routines to lock and unlock the clipboard, to make inquiries about its contents, and to register new targets.

XmClipboardLock prevents another application from gaining access to the Motif clipboard. **XmClipboardUnlock** allows other applications to gain access. The clipboard interface automatically locks the clipboard during calls to **XmClipboardStartRetrieve** and **XmClipboardEndRetrieve**. At other times, an application can use **XmClipboardLock** and **XmClipboardUnlock** to lock the clipboard explicitly.

The clipboard interface includes four routines for making inquiries about the clipboard contents:

- **XmClipboardInquireCount** returns the number of targets for which data exists on the clipboard.
- **XmClipboardInquireFormat** returns the name of the target for a given index of targets on the clipboard. An application could retrieve the names of all the targets associated with data on the clipboard by first calling **XmClipboardInquireCount** to find out how many such targets exist and then calling **XmClipboardInquireFormat** with indices from 1 to the number of targets, inclusive. Note that the first index for **XmClipboardInquireFormat** is 1, not 0.
- **XmClipboardInquireLength** returns the number of bytes of data associated with a given target on the clipboard.
- **XmClipboardInquirePendingItems** returns a list of pairs of data ID and private ID for a given target if that target exists on the clipboard and if the owner has asserted that it will supply the actual data on request (but has not yet done so).

An application that makes more than one call to an inquiry function at a time should use **XmClipboardLock** and **XmClipboardUnlock** to lock the clipboard for the duration of the interaction.

XmClipboardRegisterFormat registers a new target with the clipboard interface. The application supplies the length of the data in bits along with the name of the target so that the correct byte order will be maintained when transferring data across platforms.

All targets defined in ICCCM are preregistered; the application does not have to call **XmClipboardRegisterFormat** for these.

Appendix B
Data Transfer in the Motif Toolkit

This appendix summarizes the data transfer support in the Motif toolkit.

Table B–1. Data Transfer Mechanisms in the Standard Widget Set

Widget	**Primary**	**Secondary**	**Clipboard**	**Drag and Drop**
XmCascadeButton	Yes (Source only)	No	No	Yes (Drag only)
XmCascadeButtonGadget	Yes (Source only)	No	No	Yes (Drag only)
XmDrawnButton	Yes (Source only)	No	No	Yes (Drag only)
XmContainer	Yes	No	Yes	Yes
XmLabel	No	No	No	Yes (Drag only)
XmLabelGadget	No	No	No	Yes (Drag only)
XmList	Yes (Source only)	No	Yes (source only)	Yes (Drag only)
XmPushButton	Yes (Source only)	No	No	Yes (Drag only)
XmPushButtonGadget	Yes (Source only)	No	No	Yes (Drag only)
XmScale	No	No	No	Yes (Drag only)

XmText	Yes	Yes	Yes	Yes
XmTextField	Yes	Yes	Yes	Yes
XmToggleButton	Yes (Source only)	No	No	Yes (Drag only)
XmToggleButtonGadget	Yes (Source only)	No	No	Yes (Drag only)

The following widgets provide an **XmNconvertCallback** resource. Therefore, your application can supply an **XmNconvertCallback** procedure on any of the following widgets:

- **XmPrimitive** and all its subclasses
- **XmContainer**
- **XmDrawingArea**
- **XmScale**

The following widgets provide an **XmNdestinationCallback** resource. Therefore, your application can supply an **XmNdestinationCallback** procedure on any of the following widgets:

- **XmContainer**
- **XmDrawingArea**
- **XmList**
- **XmText**
- **XmTextField**

Index