MOTIF 2.1—Widget Writer’s Guide

Desktop Product Documentation

The Open Group

Desktop Product Documentation:

Motif 2.1—Widget Writer's Guide

ISBN 1-85912-129-2

Document Number M216

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—Widget Writer's Guide* describes how to create Motif widgets. Widgets are objects from which applications programmers build Motif applications. The guide explains (primarily through examples) how to write an Intrinsics-based widget that conforms to Motif specifications.

Audience

This guide is aimed at experienced programmers who wish to write their own Motif widgets.

The ideal reader will have as prerequisite some experience writing Intrinsics-based widgets. Failing that, the reader must have at least a general understanding of the X Window System, plus a good grasp of Xlib and Xt. The reader should also understand Motif programming topics such as resources, actions, and keyboard traversal. Finally, the more the reader understands about object-oriented programming, the better.

This guide assumes that the reader is familiar with the American National Standards Institute (ANSI) C programming language or with the C++ programming language.

Applicability

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

Purpose

After reading this guide, you should know how to write a subclass of either a Primitive or Manager Motif widget. In addition, you will also know how to make your new widget(s) accessible to Motif application programmers.

This guide does not explain the Intrinsics.

Organization

This guide is organized into two parts. Part 1, covering Chapters 1 through 15, is a programmer's guide. The chapters are arranged as follows:

- Chapter 1 explains the software and prerequisite knowledge you will need in order to write a Motif widget.
- Chapter 2 takes you step by step through the mechanics of creating a Motif widget in the C language. This chapter also includes information on writing multithread-safe widgets.
- Chapter 3 details how to write the class record for a Motif primitive widget.
- Chapter 4 details how to write the class record for a Motif manager widget.
- Chapter 5 explains how to use traits.
- Chapter 6 describes how to create resources and synthetic resources.
- Chapter 7 explains how to set up translations and actions and explains several important action routines of Primitive and Manager.
- Chapter 8 explains how to use the Xme widget-writing routines.
- Chapter 9 describes how to handle textual data.
- Chapter 10 describes the Uniform Transfer Method (UTM) for transferring data between widgets.
- Chapter 11 explains how to write a Motif button widget.
- Chapter 12 describes geometry management.
- Chapter 13 details how to write a Motif widget in C++.
- Chapter 14 explains how to extend UIL to interpret your new Motif widgets.
- Chapter 15 describes miscellaneous features of Motif widget writing such as internationalization.
- Chapter 16 discusses widget printing.

Part 2 of the guide includes Chapters 17 through 19. This part consists of pure reference documentation categorized as follows:

- Chapter 17 details all the Xme widget-writer routines.

- Chapter 18 details all Motif traits and trait methods.
- Chapter 19 details all the Exm demonstration widgets that we are providing to serve as good examples of small Motif widgets.

Note: The code examples that appear throughout this manual are not guaranteed to be multithread-safe.

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 Guide
Document Number M213 ISBN 1-85912-134-9

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

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 2 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

In this guide, mouse buttons are described using a *virtual button* mechanism to better describe behavior independent from the number of buttons on the mouse. This guide assumes a 3-button mouse. On a 3-button mouse, the leftmost mouse button is usually defined as **BSelect**, the middle mouse button is usually defined as **BTransfer**, and the rightmost mouse button is usually defined as **BMenu**. For details about how virtual 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

Part 1

Widget Writer’s Guide

Chapter 1

What Is Involved in Writing a Motif Widget

This chapter helps answer the following questions:

- Should you create your own Motif widget?
- What do you need to know in order to create Motif widgets?
- What software do you need?
- What kind of Motif widgets can you write?
- How much work is involved in writing a Motif widget?
- What kinds of design considerations should I make?

1.1 Pros and Cons of Writing a Motif Widget

Why would you want to write your own Motif widget? After all, Motif Release 2.0 already provides dozens of widgets. Using the standard widget set has some big advantages. The biggest advantage is, of course, that you do not have to do the work. Someone has already gone through the painstaking process of developing the widgets and testing them. The existing Motif widget set is used by thousands of programmers and millions of computer users. This wide use ensures that software defects (bugs) will be ferreted out and reported, and that the standard Motif widgets will grow ever more mature and defect free. Best of all, when a bug is found in one of the Motif widgets, you do not have to fix it.

So, what possible advantage would there be in writing your own widget? Broadly speaking, the biggest advantage is customization. The standard Motif widget set is meant to provide general-purpose solutions. However, general-purpose solutions may not always be sufficient. For example, you may need to write widgets that

- Provide flexible table layout
- Display complex graphs
- Display new kinds of scales, such as circular scales
- Generate form layouts that are appropriate for certain applications

Custom-built widgets can answer different needs. The most obvious need is to provide a widget for a specific application. However, custom-built widgets can also benefit a related group of applications. For example, a geologic consortium could produce a set of widgets that are useful for geologic applications. Doing so would give geologists a more consistent interface across geologic applications and would give geologic programmers a common set of widgets to work from.

Another advantage of writing your own widgets is that you can easily subclass them. In fact, you can gradually build an entire widget tree to meet your needs.

1.2 Prerequisite Knowledge

Before attempting to write a Motif widget, you should know

- A lot about the Intrinsics (Xt).
- At least a little about Xlib. For example, you should know something about Graphics Contexts (GCs).

- A lot about writing Motif applications so that you understand what it is that Motif applications programmers expect from Motif widgets.
- A lot about Motif style as described in the *CDE 2.1/Motif 2.1—Style Guide and Glossary*. All the Motif widgets you write should be style-guide compliant.

1.3 The Software You Will Need

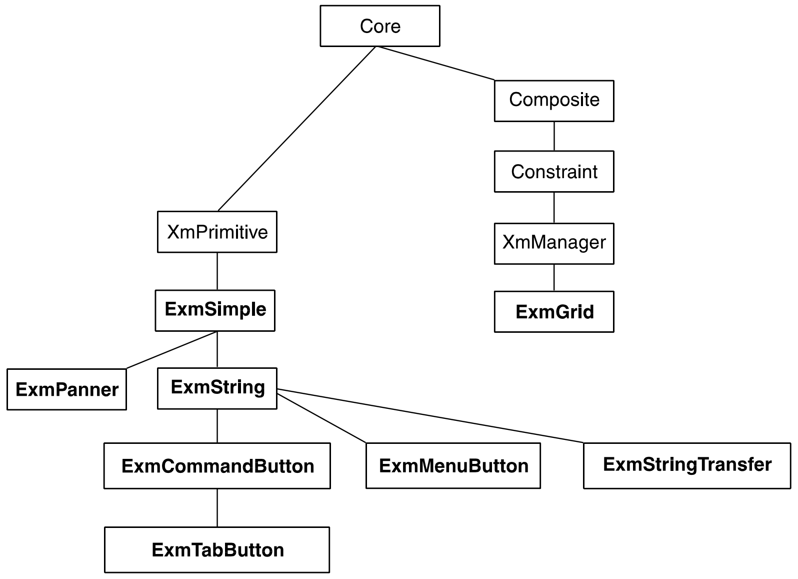
In order to write your Motif widget, you do not need the source files for the standard Motif widgets. In other words, you can develop Motif widgets even if you do not have a Motif source license. (Of course, if you do have a Motif source license, you will probably find it somewhat easier to develop widgets.)

You must have the following software in order to create a Motif widget:

- A C or C++ compiler. If you are programming in the C language, an ANSI C compiler is required.
- The Motif applications programmer's development software. (This comprises all the header files necessary for developing Motif applications, plus the **libXm.a** library.)

Although not a requirement, we strongly recommend that you have access to the Exm Motif demonstration widget set. The source code for this widget set is stored online in the **demos/lib/Exm** directory that accompanies Motif. This widget set illustrates how to code many commonly used Motif widget features. Figure 1-1 shows the class hierarchy of the Exm demonstration widget set.

Figure 1–1. Hierarchy of Exm Demonstration Widget Set



If you are writing a Motif widget that is to be accessible from a User Interface Language (UIL) application, then you will also need to have the following software:

- A UIL compiler.
- The **wml/tools** directory.
- Motif Resource Manager (MRM) software. This comprises several header files plus a runtime library named **libMrm.a**.

You can develop Motif widgets from any operating system that supports Motif. Ideally, the widget source code you write should compile without modification on any operating system that supports Motif.

1.4 What Kinds of Widgets Can You Create?

This guide tells you how to subclass a Motif widget from **XmPrimitive** or **XmManager**. This guide does not explain how to write a gadget or a shell widget.

Furthermore, this guide does not explain how to subclass directly from the Athena widget set or from any other widget set.

Subclassing from **XmPrimitive** or **XmManager** will allow your widget to call existing methods that provide much of the appropriate Motif look and feel.

1.5 How Much Work Is Involved?

If you have made it this far, then you presumably have the prerequisite knowledge and software. Now the question is whether you also have the determination to complete writing a Motif widget.

For very simple widgets, you should expect to write at least 800 to 1000 lines of source code. Complex widgets may easily require 10,000 to 20,000 lines of source code. The amount of code is related not only to the complexity of the widget but also to the availability of suitable code in superclasses. That is, if your widget's superclass already contains a suitable method, then your widget can simply inherit that method.

We do not recommend that you write all this code "from scratch." Rather, we suggest copying large chunks of your widget from existing widgets, such as the Exm demonstration widgets we provide.

1.6 Defining the Goal

The primary goal for Motif widget writers is to create a self-contained object that satisfies the *CDE 2.1/Motif 2.1—Style Guide and Glossary* conventions and interacts appropriately with other Motif widgets. Another important goal is to write widget code that conforms to the widget coding guidelines presented in Chapter 2.

By "self-contained object," we are suggesting that you follow good object-oriented design rules. For example, any widget that you create should itself be subclassable. Design for reuse.

Neither Motif nor the Intrinsics will be able to verify that you have followed the dictates of the *CDE 2.1/Motif 2.1—Style Guide and Glossary*. However, Motif users will know when you have not followed the conventions. If you are not yet familiar with the *CDE 2.1/Motif 2.1—Style Guide and Glossary*, you should become so. In order to give your new widgets that all-important Motif look and feel, your widgets should do the following:

- Inherit (or envelop) methods from **XmPrimitive** and **XmManager**, whenever possible.
- Use the internal Motif widget writer functions (the **Xme** functions), whenever possible. The standard Motif widget set uses the **Xme** functions extensively. For example, the **XmeDrawShadows** function draws a Motif style shadow inside a widget. Therefore, there is no good reason to invent your own shadow drawing routine when you can call the existing one.
- Specify translations and actions consistent with the *CDE 2.1/Motif 2.1—Style Guide and Glossary*. In other words, respond to events as similar Motif widgets would.

Chapter 2

A Motif Widget Writing Tutorial

This chapter walks you through the steps in creating a Motif widget. It focuses on a demonstration widget named **ExmSimple**. While not an officially released widget, the **ExmSimple** widget does follow the same guidelines used to write standard Motif widgets. Furthermore, this example widget provides an excellent opportunity for you to study the inner workings of widgets. You can find the source code **ExmSimple** and the other sample C widgets in the **demos/widgets/Exm/lib** directory.

In this chapter, we focus more on the big picture of writing widgets than on the fine details of perfecting them. (Later chapters provide these details.) For now, this chapter focuses on producing a working widget, albeit a rather simple one.

This chapter also includes general information on writing multithread-safe widgets.

2.1 Namespace

You will undoubtedly create many names while coding a new widget. For example, you will need to create names for new resources, new macros, new variables, new methods, and the new widgets themselves. We recommend that all these names begin with the same prefix.

The chosen prefix should be a short name, generally only two or three characters long. Furthermore, the chosen prefix should help identify the organization or the product from which the widgets came. For instance, all the example widgets accompanying this guide are tagged with the prefix *Exm* (Example Motif).

You should avoid the following prefixes:

- *X*
- *Xt*
- *_Xt*
- *xm*
- **Xm**
- *Xme*
- **XmQT**
- *_Xm*
- *Exm*

Xlib reserves the *X* prefix. The X Toolkit reserves the *Xt* and *_Xt* prefixes. Motif reserves the other six prefixes.

2.2 Recommended Files for Each Widget

We recommend that you implement each Motif widget in three files: a public header file, a private header file, and a source code file. If you call the new widget *ExmMyName*, Motif recommends that you call the public header file **MyName.h**, the private header file **MyNameP.h**, and the source code file **MyName.c**.

Some operating systems prohibit long filenames. If you are concerned with portability, abbreviate the filenames as described by the Intrinsics documentation.

2.3 The Widget Public Header File

The purpose of the widget public header file is to define the Application Programming Interface (API) for the widget. That is, the widget public header file defines the mechanism to instantiate a widget and the methods to access or modify the widget's public data.

All Intrinsics-based widgets provide a widget public header file. Motif adds few recommendations for widget public header files beyond those imposed by the Intrinsics. The purpose of this section is to review the important features of Intrinsics-based public header files and fill in the details relevant to Motif. Follow these steps to create a widget public header file:

1. Ensure that the file is included only once.
2. Include the appropriate header files.
3. Allow for use by a C++ application. (This is an optional step, but is recommended.)
4. Specify the names for the widget class and instance types.
5. Define the string equivalents of any new resource names used by this widget.
6. Define the application programmer's interface to the widget, including any possible convenience functions for this widget.

The following subsections detail these steps by examining the public header file for the **ExmSimple** widget. This file is stored online in the **demos/widgets/Exm/lib** directory at pathname **Simple.h**.

2.3.1 Step 1: Ensure That the File Is Included Only Once

You must encase the contents of the header file inside a conditional compilation directive like the following:

```
#ifndef _ExmSimple_h
#define _ExmSimple_h
  ...
#endif /* _ExmSimple_h */
```

These lines prevent compiler errors by ensuring that the code inside the file will be included only once per compilation.

2.3.2 Step 2: Include the Appropriate Header Files

You must include the public header file for your widget’s superclass. The superclass of **ExmSimple** widget is **XmPrimitive**. The public header file of the **XmPrimitive** widget is **Xm/Primitive.h**. Therefore, **Simple.h** includes the following declaration:

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

Now consider the **ExmString** demonstration widget. The superclass of the**ExmString** widget is **ExmSimple** widget. Therefore, the **ExmString** public header file must include the following declaration:

```
#include <Exm/Simple.h>
```

2.3.3 Step 3: Allow for C++ Compilation

You should encase the remainder of the public header file inside the following pair of conditional compilation directives:

```
#ifdef __cplusplus
extern "C" {
#endif
 ...

#ifdef __cplusplus
}  /* Close scope of ’extern "C"’ declaration which encloses file. */
#endif
```

The preceding code prevents link-time errors when C++ applications use this widget.

When thinking up variable names, try to avoid names that are C++ keywords, such as *class* and

new. Using such keywords as variable names could prevent C++ compilation.

2.3.4 Step 4: Specify Widget Class Names

You must create externally accessible names for the widget and widget class.

For example, the following provides the definition of widget and widget class for the **ExmSimple** widget:

```
externalref WidgetClass exmSimpleWidgetClass;
typedef struct _ExmSimpleClassRec *ExmSimpleWidgetClass;
typedef struct _ExmSimpleRec      *ExmSimpleWidget;
```

The *externalref* macro encapsulates all system dependencies regarding external data references. Therefore, for portability, you should use the **externalref** macro instead of the *extern* keyword in order to make variables externally accessible. You should use *extern* to make functions externally accessible.

2.3.5 Step 5: Define String Equivalents of New Resource Names

You must define string equivalents for every new resource name created by your widget. However, if your widget is using a resource name already used in the Motif toolkit, you should not define a string equivalent for it. You define the string equivalents with the **#define** preprocessor directive.

For example, the **ExmSimple** widget defines a new resource named **ExmNsimpleShape**. Since this resource name is not defined in the standard Motif widget set, the widget public header file defines the following two string equivalents:

```
#define ExmNsimpleShape "simpleShape"
#define ExmCSimpleShape "SimpleShape"
```

The preceding definitions associate a literal string with the new resource name.

In addition to **ExmNsimpleShape**, the **ExmSimple** widget specifies two other resources: **XmNmarginHeight** and **XmNmarginWidth**. However, the widget public header file does not need to define string equivalents for these resources because these resource names are already used in the standard Motif widget set (for example, by **XmLabel**). If you want to determine whether a particular resource name has already been defined by Motif, look in the **Xm/XmStrDefs.h** file.

You must also define a string equivalent for each new representation type used by your widget. (See Chapter 6 for details on representation types.) For example, the **ExmSimple** widget creates a representation type named **ExmRSimpleShape**. Therefore, the widget public header file for **ExmSimple** defines this string equivalent as follows:

```
#define ExmRSimpleShape "ExmSimpleShape"
```

In addition, since the **ExmRSimpleShape** representation type is an enumerated data type, the widget public header file must also specify all its enumerated constants as follows:

```
enum { ExmSHAPE_OVAL=0, ExmSHAPE_RECTANGLE=1 };
```

You should use Motif’s representation type facility to register new enumerated types. (See Chapter 6 for details.)

2.3.6 Step 6: Specify the API for This Widget

Your widget public header file should define the API for the widget. For example, the following code establishes the API for the **ExmSimple** widget:

```
extern Widget ExmCreateSimple(
                              Widget    parent,
                              String    name,
                              Arg      *arglist,
                              Cardinal  argCount
                             );
```

If your widget contains additional convenience functions, then this is the place to declare them.

2.4 The Widget Private Header File

The widget private header file serves the same purpose in Motif that it does for all Intrinsics-based widgets. Follow these steps to create a private header file that is compatible with other Motif widgets:

1. Ensure that the file is included only once.
2. Specify the appropriate header files.
3. Allow for use by a C++ application. (This is an optional step, but is recommended.)
4. Define inheritance macros for methods that your widget wishes to export to subclasses. Define new data types to support these macros.
5. Declare the widget class part.
6. Declare the full widget class record.
7. Declare the widget instance part.
8. Declare the full widget instance record.
9. Declare constraint structures if you are writing a manager widget.
10. Define the API of any private functions used by other classes.

The following subsections detail these steps by examining the private header file for the **ExmSimple** widget. This file is stored online in the **demos/widgets/Exm/lib** directory at pathname **SimpleP.h**.

2.4.1 Step 1: Ensure That the File is Included Only Once

You must encase the contents of the file inside a conditional compilation directive, as follows:

```
#ifndef _ExmSimpleP_h
#define _ExmSimpleP_h
 ...
#endif /* _ExmSimpleP_h */
```

These lines prevent compiler errors by ensuring that the code inside the file will only be included once per compilation.

2.4.2 Step 2: Include the Appropriate Header Files

You must include the following two header files:

- The widget’s public header file
- The private header file of your widget’s immediate superclass

For example, the **ExmSimple** widget derives from the **XmPrimitive** class. Therefore, the widget private header file of **ExmSimple** includes the following two files:

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

2.4.3 Step 3: Allow for C++ Compilation

You should encase the remaining code of the file inside the following pair of conditional compilation directives:

```
#ifdef __cplusplus
extern "C" {
#endif
 ...

#ifdef __cplusplus
}  /* Close scope of ’extern "C"’ declaration which encloses file. */
#endif
```

If the application program using your widget is written in C++, then allowing for C++ compilation is mandatory. Otherwise, allowing for C++ compilation is merely a good idea.

2.4.4 Step 4: Define Inheritable Methods

You must define inheritance class method macros. Once defined, your widget’s subclasses can inherit your widget’s methods simply by specifying the inheritance macro in the appropriate field of the class record. For example, the **ExmSimple** widget creates the following eight inheritance macros:

```
#define ExmInheritDrawVisual     ((XtWidgetProc) _XtInherit)
#define ExmInheritDrawShadow     ((XtWidgetProc) _XtInherit)
#define ExmInheritCreateGC       ((XtWidgetProc) _XtInherit)
#define ExmInheritDestroyGC      ((XtWidgetProc) _XtInherit)
#define ExmInheritSelectGC       ((ExmSelectGCProc) _XtInherit)
#define ExmInheritCalcVisualSize ((XtWidgetProc) _XtInherit)
#define ExmInheritCalcWidgetSize ((XtWidgetProc) _XtInherit)
#define ExmInheritReconfigure    ((ExmReconfigureProc) _XtInherit)
```

If an inheritance macro requires a new data type definition, you must provide it here. For example, **ExmInheritSelectGC** and **ExmInheritReconfigure** both require new data type definitions, as follows:

```
typedef GC   (*ExmSelectGCProc)(
                        Widget);
typedef void (*ExmReconfigureProc)(
                        WidgetClass,
                        Widget,
                        Widget) ;
```

The inheritance macros should follow Motif naming conventions. The inheritance macro names should begin with the widget set prefix (in this case, *Exm*) followed by the word *Inherit*. Similarly, the new data type definitions should begin with the widget set prefix and end with *Proc*.

2.4.5 Step 5: Define the Widget Class Part

The widget class part defines the inheritable methods and data members of the widget. For example, following is the widget class part of the **ExmSimple** widget:

```
typedef struct _ExmSimpleClassPart
{
        XtWidgetProc                  draw_visual;
           XtWidgetProc               draw_shadow;
           XtWidgetProc               create_gc;
           XtWidgetProc               destroy_gc;
           ExmSelectGCProc                 select_gc;
        XtWidgetProc                  calc_visual_size;
           XtWidgetProc               calc_widget_size;
```

```
   ExmReconfigureProc      reconfigure;
        XtPointer                         extension;
} ExmSimpleClassPart;
```

Motif strongly recommends specifying an *extension* field as the last field of every widget class part. Providing an *extension* field helps implement binary compatibility if you add methods to the widget in future releases. The *extension* field should have the **XtPointer** data type.

2.4.6 Step 6: Declare the Full Class Record

You must define the widget’s full class record. This structure specifies the class parts of all widget classes in the current widget’s hierarchy. The first field in the full class record specifies the class part of the top widget in the hierarchy (**Core**). The next field specifies the class part of the widget below the top widget, and so on. The final field specifies the class part of the current widget.

For example, the **ExmSimple** widget derives from **Core** and **XmPrimitive**. Therefore, the class record structure of **ExmSimple** is as follows:

```
typedef struct _ExmSimpleClassRec
{
        CoreClassPart           core_class;
        XmPrimitiveClassPart    primitive_class;
        ExmSimpleClassPart      simple_class;
} ExmSimpleClassRec;
```

There are a few conventions that you should follow in the full class record:

- The tag name (for example, *_ExmSimpleClassRec*) should begin with an underscore.
- The data type name (for example, **ExmSimpleClassRec**) should have the same name as the tag except for the leading underscore.

After defining the data type, you must now declare an *externalref* variable having this data type; for example:

```
externalref ExmSimpleClassRec exmSimpleClassRec;
```

By convention, the variable has the same name as the data type except that the first character of the variable is always lowercase.

All manager widgets derive from **Core**, **Composite**, **Constraint**, and **XmManager**. Therefore, the class record structure of the sample manager widget **ExmGrid** is the following:

```
typedef struct _ExmGridClassRec
{
    CoreClassPart       core_class;
    CompositeClassPart  composite_class;
    ConstraintClassPart constraint_class;
    XmManagerClassPart  manager_class;
    ExmGridClassPart    grid_class;
} ExmGridClassRec;
externalref ExmGridClassRec exmGridClassRec;
```

2.4.7 Step 7: Define the Widget Instance Part

Your private header file typically declares a widget instance record. The widget instance record describes the inheritable data members of your widget. The inheritable data members consist of your widget's resources plus those variables that subclasses may need access to. For example, following is the widget instance record for the **ExmSimple** widget:

```
typedef struct _ExmSimplePart
{
    unsigned char           simple_shape;
    Dimension               margin_height;
    Dimension               margin_width;
    GC                      normal_gc;
    GC                      insensitive_gc;
    Dimension               pref_width;
    Dimension               pref_height;
    Boolean                 need_to_compute_width;
    Boolean                 need_to_compute_height;
    XRectangle              visual;
    Boolean                 need_to_reconfigure;
    Pixel                   saved_foreground;
} ExmSimplePart;
```

By convention, the top fields in the widget instance part are declarations of the widget's resources. The **ExmSimple** widget declares three new resources, so the widget instance part declares them in the top three fields.

The fields below the resource fields declare variables that are accessible to subclasses of **ExmSimple**. For example, the **ExmString** widget is a subclass of **ExmSimple**, so any function in **ExmString** can access any of these variables. Conversely, a widget that is not a subclass of **ExmSimple** should not access any of these variables.

2.4.8 Step 8: Declare the Full Widget Instance Record

You must declare a full instance record for your widget private header file. The full instance record of **ExmSimple** appears as follows:

```
typedef struct _ExmSimpleRec
{
    CorePart            core;
    XmPrimitivePart     primitive;
    ExmSimplePart       simple;
} ExmSimpleRec;
```

All Motif manager widgets derive from **Core**, **Composite**, **Constraint**, and **XmManager**. Therefore, the full instance record of the *ExmGrid* sample manager widget is as follows:

```
typedef struct _ExmGridRec
{
    CorePart            core;
    CompositePart       composite;
    ConstraintPart      constraint;
    XmManagerPart       manager;
    ExmGridPart         grid;
} ExmGridRec;
```

2.4.9 Step 9 (Optional): Declare Constraints Structures

Note: If you are writing a primitive widget, you can skip this section.

If you are writing a manager widget, you should always define the following two structures:

- A constraint part structure
- A full constraint structure

You should define these structures even if the widget does not currently support any constraint resources. Declaring these constraint structures now will prevent source compatibility problems if a future subclass wants to provide constraint resources.

The following subsections detail the two constraint structures.

2.4.9.1 The Constraint Part Structure

The constraint part structure defines the constraints themselves, one constraint per field. For example, the *ExmGrid* widget defines two constraints, so its constraint part structure is as follows:

```
typedef struct _ExmGridConstraintPart
{
        Dimension      grid_margin_width_within_cell;
        Dimension      grid_margin_height_within_cell;
} ExmGridConstraintPart, * ExmGridConstraint;
```

You should follow these conventions:

- The tag name (for example, *_ExmGridConstraintPart*) should begin with an underscore.
- The first data type name (for example, *ExmGridConstraintPart*) should have the same name as the tag except without the leading underscore.
- The second data type name (for example, *ExmGridConstraint*) should be a pointer having the same name as the first data type except for the *Part* suffix.

2.4.9.2 The Full Constraint Structure

The full constraint structure specifies the widgets in the current widget hierarchy that define constraints. The last field in the structure specifies the current widget (assuming it defines constraints). For example, the *ExmGrid* widget defines the following full constraint structure:

```
typedef struct _ExmGridConstraintRec
{
        XmManagerConstraintPart manager;
        ExmGridConstraintPart   grid;
} ExmGridConstraintRec, *ExmGridConstraintPtr;
```

You may be wondering why we specified **XmManagerConstraintPart** as one of the fields. After all, the **XmManager** widget does not currently have any constraints. However, in order to allow for the future possibility of constraints in **XmManager**, the **XmManager** widget defines a constraint part structure. The first field of your full constraint structure should always contain **XmManagerConstraintPart**.

2.4.9.3 The Constraint Access Macro

If you are writing a manager widget that provides constraints, your private header file should define a constraint access macro. The constraint access macro should be named *widget_name***CPart**. For example, the *ExmGrid* widget defines a macro named **ExmGridCPart**. Here is its definition:

```
#define ExmGridCPart(w) \
  (&((ExmGridConstraintPtr) (w)->core.constraints)->grid)
```

2.4.10 Step 10 (Optional): Declaring Private Functions

In the course of writing your own widgets, you can create convenience functions that could help other widget writers (or perhaps yourself when writing a subsequent widget). Prototypes for these convenience functions must appear in the widget private header file.

Think about it this way. If you want a function to be accessible to application programmers, define it in the widget public header file. If you want a function to be inaccessible to application programmers, define it in the widget private header file.

2.5 The Widget Source Code File

The most time consuming portion of writing a widget is, of course, writing the widget source code file itself. The steps for creating a widget source code file are as follows:

1. Include the appropriate header files.
2. Define any macros that your widget may need. However, if a subclass might need a macro, define it in the widget private header file instead of the widget source code file.
3. Declare all the widget methods as static functions.
4. Define the widget's translations string and its actions table.
5. Declare the widget's resources.
6. Declare the widget class record.
7. Provide external definitions of the widget class record and the widget class.
8. Declare any static variables needed throughout the widget.
9. Declare any trait record variables.
10. Provide the code for all the widget methods.

The **Simple.c** file is an example of a widget source code file. The following subsections examine selected portions of this file.

2.5.1 Step 1: Include the Appropriate Header Files

The typical widget source code file begins by including several header files. The order of inclusion is important. You should include

- Any operating system or C header files first
- Any Xlib or Xt header files next

- Your widget's private header file (required)
- Any Motif header files

For example, the widget source code file for the *ExmSimple* widget does not specify any operating system, C, Xlib, or Xt header files. (Actually, a lot of these header files will be automatically included by Motif header files.) **Simple.c** does require its private header file (**SimpleP.h**). **Simple.c** also needs a bunch of other Motif header files in order for Xme calls and traits to work properly. The entire collection of header files included in **Simple.c** is as follows:

```
#include <Exm/SimpleP.h>    /* widget private header file for ExmSimple */
#include <Xm/DrawP.h>       /* for Xme drawing functions */
#include <Xm/RepType.h>     /* for representation type facility */
#include <Xm/Screen.h>      /* for screen information */
#include <Xm/TraitP.h>      /* for installing traits */
#include <Xm/CareVisualT.h> /* for XmQTcareParentVisual trait */
#include <Xm/ContItemT.h>   /* for XmQTcontainerItem trait */
#include <Xm/ContainerT.h>  /* for XmQTcontainer trait */
```

Table 2-1 lists some of the files that are more commonly used by widget writers.

Table 2–1. Helpful Header Files For Motif Widget Writers

File	Contains:
DrawP.h	Definitions for Motif-style internal drawing functions
RepTypeP.h	Definitions for representation type facility
TraitP.h	Definitions for trait installation and access routines
XmP.h	Widget private header file for **XmPrimitive** widget; definitions for many Motif macros and constants; definitions to support **XmPartOffset** binary compatibility; definitions of many data types used by Motif widgets; definitions of many enumerated constants used by Motif widgets
Xm.h	Typedefs for callback structures and representation types; this file includes **XmP.h**
XmStrDefs.h	String definitions for all Motif XmN, XmC, and XmR data types; this file includes **Xm.h**

2.5.2 Step 2 (Optional): Define Your Macros

After specifying header files, you can optionally define any macros useful to your widget. Motif makes the following stylistic suggestions regarding your macros:

- Try to avoid duplicating macros that Motif already defines. The Motif header file **Xm/XmP.h** defines several macros that your widget can access. The wise widget writer always looks at this file prior to creating a new macro. Some of these macros (such as **XtWidth**) are required for any Intrinsics-based widget set. Other macros (like **XmLOOK_AT_BACKGROUND**) are specific to Motif. All Motif-specific macros begin with the **Xm** prefix.
- Format the code for long macros as you would format the code for a function. In other words, make the macro easy to understand.
- Comment your macros.
- Place all macros in this section, even if the macro is only used in one function. (In other words, do not define local macros.)

2.5.3 Step 3: Declare All static Functions

You should declare all **static** functions.

We recommend that you use ANSI C style function prototypes when declaring functions.

You should name class methods after their class record field names. For example, if the class record field name is **class_initialize**, then the class method should be named *ClassInitialize*. The class method name differs from the class record field name in only two respects:

- The class method name should not contain any underscores.
- The first letter of every word in the class method name should be uppercase.

There is only one exception to these naming rules and it concerns the class record field named *expose*. The preceding conventions suggest that you should name the associated class method **Expose**. Unfortunately, Xlib already defines a macro named **Expose**. Therefore, by convention, you should name this class method *Redisplay* instead of **Expose**.

By convention, you should declare the functions in the following order:

1. If your widget defines any **XmRCallProc** resources, then declare the dynamic resource defaulting methods first.
2. If your widget defines any synthetic resources, declare the synthetic resource methods next.
3. Declare all your class methods in superclass-to-subclass order. For example, when writing a primitive widget you declare any class methods from **Core** first, from **XmPrimitive** second, and from your own widget third.
4. Declare any trait methods last.

2.5.4 Step 4: Define Translations and Actions

Chapter 7 details Motif translations and actions. Motif makes the following stylistic demands on widget writers:

- You should give the translations string a name ending with *Translations*. For example, *defaultTranslations* is a good name.
- You should use the name *actions* for the *XtActionsRec* array.
- You should list exactly one action per line.
- You should not declare the actions array as a constant.

2.5.5 Step 5: Declare Resources

Chapter 6 details resources. For now, you should be aware of the following Motif resource naming conventions:

- The resource structure should be named *resources*.
- The synthetic resource structure should be named *syn_resources*.
- The constraint resource structure should be named *constraint_resources*.
- The constraint synthetic resource structure should be named *syn_constraint_resources*.
- You should not declare the *resources* array or the *syn_resources* array as a constant.

The fifth field of each resource structure defines an offset. If you want to create a widget that will be a binary compatible with future releases, then you should use the **XmPartOffset** macro to define the offset. (See Chapter 15 for details on binary compatibility.) If binary compatibility is not a goal, then you should use the **XmPartOffset** macro to define the offset.

2.5.6 Step 6: Declare the Class Record

The widget class record is your widget's table of contents. Chapter 3 details the class record for Motif primitive widgets; Chapter 4 details the class record for Motif manager widgets. For now, you should concentrate on a few stylistic points.

Make sure that all fields in the class record are commented with the formal names of the fields. For the portions of the class record defined by the Intrinsics, refer to Intrinsics documentation to find the formal names. For the portions of the class record defined by Motif, refer to Chapters 3 and 4. For example, the following excerpt from **ExmSimple** shows the proper way to comment the fields:

```
...
/* class_part_initialize */      ClassPartInitialize,
/* class_inited */               FALSE,
/* initialize */                 Initialize,
/* initialize_hook */            NULL,
/* realize */                    XtInheritRealize,
...
```

Motif recommends that the last field of every Motif class record be an *extension* field. An *extension* field must contain one of the following:

- The name of a class extension record
- *NULL*, to indicate the absence of a class extension record

Motif and the Intrinsics provide the following class extension record data types:

- **XmBaseClassExtRec** Motif strongly recommends that your widget not define this extension record.
- **CompositeClassExtensionRec** This is an Xt extension record. (See documentation on the Intrinsics for details.)

- **XmPrimitiveClassExtRec** This is a Motif extension record. (See Chapter 3 for details on its fields.) If you declare an **XmPrimitiveClassExtRec** variable in your widget, it should be named *primClassExtRec*.
- **XmManagerClassExtRec** This is a Motif extension record. (See Chapter 4 for details on its fields.) If you declare a **XmManagerClassExtRec** variable in your widget, it should be named *managerClassExtRec*.

Motif strongly recommends that all class extension record variables be declared as *static*; for example:

```
static XmPrimitiveClassExtRec  primitiveClassExtRec = ...
```

You should not declare the class record as a constant.

2.5.7 Step 7: Provide the External Definitions

Your widget must provide an external definition of the widget class record and the widget class. For example, the external definition of the widget class record for the **ExmSimple** widget is as follows:

```
externaldef (exmsimpleclassrec) ExmSimpleClassRec exmSimpleClassRec = {
```

and the external definition of the widget class pointer is as follows:

```
externaldef (exmsimplewidgetclass) WidgetClass exmSimpleWidgetClass =
              (WidgetClass) &exmSimpleClassRec;
```

Motif recommends using the **externaldef** macro instead of the *extern* keyword when declaring variables. The **externaldef** macro handles a few portability problems.

2.5.8 Step 8: Declare Any Static Variables

If your source code file requires any *static* (file scope) variables, declare them next. For instance, if your widget uses representation types, you will probably have to declare at least one static representation type variable. For example, the **ExmSimple** widget creates a new representation type. Therefore, it provides the following static variables:

```
/* The SimpleShapeNames variable holds some normalized values.
   The XmRepTypeRegister function will use these values to determine the
   legal values for a given representation type. */
static String SimpleShapeNames[] = {
        "simple_oval",
        "simple_rectangle"
};
/* Declare a representation type variable. */
static XmRepTypeId simpleShapeId;
```

2.5.9 Step 9: Declare Any Trait Record Variables

If your widget installs any traits, you should declare the trait record variable(s) here. (See Chapter 5 for details about traits. For example, the **ExmString** widget installs the *XmQTaccessTextual* trait, so it requires the following trait record variable declaration:

```
static XmConst XmAccessTextualTraitRec StringATT = {
  0,                            /* version */
  StringGetValue,
  StringSetValue,
  StringPreferredFormat,
};
```

Notice that the variable declaration contained the **XmConst** macro. If your C compiler supports the *const* keyword, then **XmConst** resolves to *const*. If your C compiler does not support the *const* keyword, then **XmConst** resolves to nothing.

2.5.10 Step 10: Write the Widget's Methods

The majority of code in most widget source code files is taken up by the code for the widget's methods. Place the widget's methods in the following order:

1. All *static* functions (in the order in which they were declared).
2. Any private global functions (in the order in which they were declared in the private header file).

3. Public functions (in the order in which they were declared in the public header file).

Table 2-2 lists all the methods of the **Simple.c** source code file in the order in which they appear in the file:

Table 2–2. Functions in ExmSimple

Method	**What the Method Does in ExmSimple:**
ClassInitialize	Creates a new representation type
ClassPartInitialize	Establishes method inheritance
Initialize	Validates resources, sets certain geometry management fields
Destroy	Calls the DestroyGC method in response to the widget's destruction
Realize	Creates the window for the widget
Resize	Determines what parts of the widget will be reduced in size when there is not enough space to display everything
Redisplay	Renders the widget after an Exposure event
SetValues	Responds to changes in resource values
QueryGeometry	Returns the preferred size of the widget
DrawVisual	Draws the widget's visual (its filled arc or filled rectangle)
DrawShadow	Draws a Motif-style shadow at the widget's outer edge
CreateGC	Creates a sensitive GC and an insensitive GC
DestroyGC	Deallocates the two GCs
SelectGC	Picks one of the two GCs
CalcVisualSize	Calculates the ideal size of the widget's visual
CalcWidgetSize	Calculates the widget's ideal size
WidgetDisplayRect	Determines the bounding box of the widget's visual

Reconfigure	Orchestrates several aspects of geometry management
ContItemSetValues	Sets new visual attributes on ExmSimple
ContItemGetValues	Gets the values of the current visual attributes of ExmSimple
SetSelectedVisual	Gets the select color of the parent
HandleRedraw	Adjusts the select color under certain situations
ExmCreateSimple	Instantiates an **ExmSimple** widget; accessible to applications

You should precede each widget method with a comment describing the purpose of the method. Following is a conventional Motif method comment:

```
/************************************************************************
 *
 *  ClassPartInitialize
 *      Processes the class fields which need to be inherited.
 *
 ************************************************************************/
```

2.6 Make the Widget Accessible to Applications

After writing a widget, you will need to test it by instantiating it from a Motif application. Motif provides two mechanisms for generating applications:

- Through a C application
- Through a UIL or WML-based application, as described in Chapter 14

It is very easy to make your new widget accessible to C applications. All you have to do is compile your widget's source code file. The resulting object file can be bound to Motif applications. You do not have to place the object file into **libXm.a**.

If you have written several related widgets, then we suggest bundling the resulting object files into an archive file (a library).

From a Motif application programmer's point of view, it is very easy to access the new widget. Application programmers need to do only the following three things:

- The application must include the public header file for the new widget For example, to access the **ExmSimple** widget, the application must include its header file, as follows:

```
#include <Exm/Simple.h>
```

- The application must instantiate the widget. For example, to instantiate the **ExmSimple** widget, call **ExmCreateSimple** or call **XtCreateWidget**, passing *exmSimpleWidgetClass* as an argument.
- On the link command line, make sure that your application gets bound to the new widget's object file(s) or archives.

The Motif directory **demos/programs/Exm/simple_app** contains a short C application that exercises the **ExmSimple** widget. This directory also contains an **Imakefile** for building the **ExmSimple** widget and the application.

2.7 MultiThread-Safe Motif Widgets

Writing a multithreaded application ideally requires that the libraries the application uses be MT-safe. Otherwise the application must ensure that only one thread is inside the library at any given time. This could lead to unnatural design and coding strategies and to sacrificing parallelism in the application. Making a library MT-safe basically means that the library should take the responsibility for ensuring consistent internal state even when the application starts multiple threads that may simultaneously invoke its API.

This section describes how to develop a MT-safe Motif widget library.

2.7.1 MT-Safety in the Xt Intrinsics

The Xt library in Release-6 of the X-Window system (X11R6) is MT-safe. A brief description of the MT-safety provided by Xt follows.

2.7.1.1 Concurrency

Concurrency refers to the number of threads that can safely execute a piece of code at the same time. Xt permits one thread per *XtAppContext* to be active. So multiple threads can be active within Xt simultaneously, if each one of them is spawned in distinct *XtAppContext*s. Thus the concurrency provided by Xt is at the *XtAppContext* level.

This model is elegant for application programmers. However it can be difficult for widget programmers since it allows multiple active threads within the widget methods concurrently, and hence all global data access within the widget library needs to be suitably protected.

Fortunately, most data in a widget is stored in the widget instance. And since a widget instance hierarchy is rooted in an *XtAppContext*, the one thread per *XtAppContext* model protects instance data from corruption due to multiple threads.

2.7.1.2 AppLock

Xt implements the one thread per *XtAppContext* policy by protecting all public entry points into the library with *AppLock*s. Every public Xt API locks the application context at entry and releases it at exit. Xt provides the following functions to lock and unlock the *XtAppContext*:

- **void XtAppLock(XtAppContext app)**
- **void XtAppUnlock(XtAppContext app)**

Note that these locks are recursive; that is, the same thread invoking **XtAppLock** multiple times with the same application context will *not* deadlock.

2.7.1.3 ProcessLock

Xt provides *ProcessLock*s to ensure that only one thread at a time can execute critical code paths. All access to global data is considered critical code and is done while the process is locked. Xt provides the following functions to lock and unlock the *ProcessLock*:

- **void XtProcessLock(void)**
- **void XtProcessUnlock(void)**

Note that these locks are recursive; that is, the same thread invoking **XtProcessLock** multiple times will not deadlock.

2.7.1.4 Deadlock

Xt avoids deadlock by mandating a simple locking hierarchy: always acquire the *AppLock* first; then acquire the *ProcessLock*.

2.7.2 MT-Safe Motif Widgets

Motif uses the same locking strategies implemented in Xt:

- Envelop all public APIs using *AppLock*s
- Protect all global data using *ProcessLock*s.

2.7.2.1 API Protection

All public entry points into the widget should be enveloped with the **XtAppLock**, **XtAppUnlock** pair. Private entry points intended for internal use only need not be protected, as the invoking public API will have acquired *AppContext* locks. Public functions that are just wrappers around Xt functions (such as **XmCreate*** functions) do not require *AppLock*s since the Xt function already does this.

Typically, widget APIs have either a *Widget*, *Display* or *Screen* parameter. The *AppContext* can be obtained from these using the following Xt functions:

- **XtAppContext XtWidgetToApplicationContext(Widget)**
- **XtAppContext XtDisplayToApplicationContext(Display *)**
- **XtAppContext XtDisplayToApplicationContext(DisplayOfScreen(Screen *))**

Widget APIs that do not have either a *Widget*, *Display* or *Screen* parameter cannot be protected using *AppLock*s. A potential deadlock exists if such a function wants to acquire the *ProcessLock* (because the function has not acquired the *AppLock* yet), resulting in a break in the locking hierarchy. In this situation, one must ensure that once the *ProcessLock* is obtained, an *AppLock* should not be attempted until the *ProcessLock* is released. Basically this means that Xt function calls, user callbacks, or other non-safe routines that can *AppLock* should not be invoked from within *ProcessLock*'ed regions in this function. For example, consider a widget public API **XmFooBar**:

```
Dimension XmFooBar(Widget w)
{
        .....
 return w->core.width;
}
```

The MT-safe version of the above function would be:

```
Dimension XmFooBar(Widget w)
{
      int return_value;
      XtAppContext app = XtWidgetToApplicationContext(w);
      XtAppLock(app);
      ......
      return_value = w->core.width;
      XtAppUnlock(app);
            return return_value;
}
```

2.7.2.2 Global Data Protection

Global data can be classified as either *read-only* or *read-write*:

- Read-only globals are those that are written into only during initialization and read thereafter. The initialization can happen either at compile time or once during runtime (typically in the Widget's *ClassInitialize* method). These globals are MT-safe and do not require any access protection, except during initialization.
- Read-Write globals can be written into at any time during program execution. All accesses must be protected by *XtProcessLock*s.

Other random pieces of global data (mostly function static variables) can be usually dealt with by either making them instance fields or stack variables. If that does not work, *ProcessLock*s can be used to protect global access regions. Exceptions are those variables that are used to communicate state information across function. These can be handled using thread specific storage to retain their semantics.

A discussion on the common global categories in Motif widgets follows.

2.7.2.2.1 Class Methods and Class Fields

A widget basically has two parts: a class record and an instance record. The instance record is per-widget and is dynamically allocated when the widget is created, and thus is *not* global data. Class records however are statically allocated global data. The class record typically contains methods and few data fields. The class methods are either set up at compile time and *never* changed thereafter or are set up at runtime during *ClassInitialize* (to resolve inheritance). Class initialization happens *once* when the first widget of that class is created. This process happens within Xt and is already under *ProcessLock*s. Thus we can be assured that these class method pointers get written into only *once* by *only* one thread.

So, we can consider the above two cases to be read-only situations and hence in general, class methods can be accessed without locks.

However, the Xt Base classes (**RectObj**, **Core**, **Constraint**, **Shell**) are exceptions since Motif does change their class method pointers at runtime to implement Pre and Post hooks. Hence we consider all the Xt Classes’ class methods to be read-write and we need to access those method pointers within *XtProcessLock*s.

Consider a code fragment that invokes the *resize* method:

```
{
      WidgetClass wc; Widget w;
      (*wc->core_class.resize)(w);

}
```

The MT-safe version would be

```
{
      WidgetClass wc; Widget w;
```

```
        XtWidgetProc resize;
        XtProcessLock();
        resize = wc->core_class.resize;
        XtProcessUnlock();
        (*resize)(w);
}
```

Most class fields are read-write and hence require *XtProcessLock* protection.

2.7.2.2.2 XContext

*XContext*s are an Xlib abstraction to implement per display storage. Since Display connections cannot be shared among *XtAppContext*s, we are ensured that the data hanging off *XContext*s also cannot be shared among *XtAppContext*s. Hence we don't need to protect these data using Locks.

However, *ProcessLock* protection may be needed to protect the initialization of the *XContext* (created using **XUniqueContext** if the initialization does not happen in a Widget's *ClassInitialize* procedure.

2.7.2.2.3 Resource Lists

Widgets define their resource lists as static global **XtResource** structures. These lists are set up at compile time and never changed thereafter. Hence these can be considered as read-only data and do not require access protection.

Motif widgets also define synthetic resource tables as static global **XmSyntheticResource** structures. These lists are also read-only and do not require protection.

2.7.2.2.4 Action Tables and Translation Tables

Widgets define their Actions and Translations using static global **XtActionRec** and **XtTranslation** structures respectively. These are set up at compile time and only modified by the (MT-safe) Xt intrinsics. Hence these can be considered read-only data and do not require access protection.

2.7.2.2.5 Traits

Motif defines the Trait mechanism as a means of sharing behavior among widget classes. A widget establishes a trait by setting up a static global **XmTrait** structure. This table is set up at compile time and its contents should not be changed thereafter. Thus Trait tables also do not require any protection.

2.7.2.2.6 Resource Converters

Xt Resource converters are used for inter-type resource conversions. The converter can return the converted data either in local static storage or in the heap. The choice is made based on the *address* passed by the caller to store the converted value in. If the address is *NULL*, static storage is used, else the memory (on the heap) pointed to by *address* is used. Since static storage should be avoided for MT-safety, always pass in a valid heap address when invoking any converter. The guidelines when using/ writing resource converters are:

- Resource converters should be able to handle both the cases of internal static storage and passed in heap storage.
- Invoke resource converters with a valid heap address for the converter to store the converted value. (Xt always invokes resource converters by passing in a valid heap address to store the converted value.)

2.7.2.2.7 Default Resource Value Procedures

XtCallProc procedures are used in widget resource lists to provide default values for a resource. Default procedures typically use static storage to store the default value. Fortunately, this usage is MT safe, since the code in Xt that invokes this procedure and copies over the default value into the resource location is within *XtProcessLock*s. Thus, the statics in default procedures present no problems.

2.7.2.3 Event Loops

Widgets may use their own event loops to implement Drag and Drop and other such esoteric features.

The event processing functions in R6 Xt that block while waiting for events drop the *XtAppLock* during the wait (**XtAppNextEvent**, **XtAppPeekEvent**, and **XAppProcessEvent**). This is done so that other threads get a chance to issue Xt calls while the event processing thread waits for events. However in the case of Drag and Drop, we really do not want other threads to get into Xt and change state while the Drag operation is in progress. There is also the danger that another event processing thread might come in and "steal" events meant for this thread.

To avoid this situation, event loops should be replaced with "busy" loops that check the event queue without giving up the *XtAppLock*.

The following code implements a "busy" event loop that does not give up the *XtAppLock* by replacing **XtAppNextEvent**:

```
while (XtAppGetExitFlag(app) == False)  {
        XEvent event;
#ifndef MT_SAFE
        XtAppNextEvent(app, &event);
        XtDispatchEvent(&event);  /* Process it */
#else  /* MT_SAFE version ...  */
        XtInputMask mask;
        while (!(mask = XtAppPending(app)))/* EMPTY */;
        /* Busy waiting - so that we don’t lose our Lock! */
        if (mask & XtIMXEvent) {  /* We have a XEvent */
       /* Get the XEvent - we know its there!*/
       /* Note that XtAppNextEvent would also process*/
       /* timers/alternate inputs */
                XtAppNextEvent(app, &event);
                /* No blocking, since an event is ready */
             XtDispatchEvent(&event);  /* Process it */
        }
        else /* Not a XEvent, its an alternate input/timer event
      - Process it. */
            XtAppProcessEvent(app, mask);
            /* No blocking, since an event is ready */
  }
```

2.7.3 MT-Safe C Library Functions

Some of the C library functions are inherently MT unsafe. A MT-safe libc implementation will provide alternatives to those functions. This section lists the common libc functions that are MT-unsafe along with their safe POSIX equivalents.

MT-Unsafe	MT-Safe
getlogin	getlogin_r
ttyname	ttyname_r
readdir	readdir_r
strtok	strtok_r
ctime, localtime...	ctime_r, localtime_r...
getgrnam, getgrgid	getgrnam_r, getgrgid_r
getpwnam, getpwuid	getpwnam_r, getpwuid_r

2.7.4 MT-Safe Motif Library Functions

The following Motif library functions are MT-safe:

- **XmFontListEntryCreate_r**
- **XmFontListCreate_r**
- **XmStringCreateFontList_r**

2.7.5 Thread Specific Storage

Occasionally, one might end up using globals to communicate state information across functions. The only way to retain the proper semantics in this case might be to use thread specific storage. The **Xthreads.h** header file in X11R6 provides a set of wrapper functions to support thread_specific storage.

The sequence of steps to creating and using Thread Specific Storage are:

- Create a unique key that identifies this storage. This step should be done only once for all the threads. The most convenient way to enforce this is to do the key creation in the concerned widget's *ClassInitialize* procedure. This also has the nice side effect of automatically being within *XtProcessLock*s. The Xthreads API for key creation is

```
xthread_key_create(xthread_key_t *key_return);
```

- Set and Get thread specific data: once a key is created, each thread may bind a value to that key. The values are specific to the binding thread and are maintained for each thread independently. Typically, this value is a pointer to heap storage where the actual data is stored. The heap is set up once for each thread during the first call to get the thread specific data. Subsequently, that heap location is used as the storage for the global variable. The Xthreads API is:

```
void xthread_getspecific(xthread_key_t key, void **value_return);
    int xthread_setspecific(xthread_key_t key, const void *value);
```

  Note that **xthread_getspecific** returns NULL the first time its invoked for a specific thread, so that gives us the chance to set up the heap storage for each thread.

Thread specific storage can be expensive, hence it should be utilized as a last resort.

2.7.6 Live Resources

For MT-safety, widget resources should be designed so that **XtGetValues** returns a copy of the resource value (in the heap). The *GetValuesHook* method or the *XmSyntheticResource export_proc* can be used to achieve this.

If a *live* pointer to the actual instance field has to be returned, flag this situation as a MT-unsafe usage so that the application programmer is aware of it.

2.7.7 CheckList

Following is a summary of the discussions in this section:

- First and foremost, avoid global and static variables. Use
  - — Instance fields to store widget specific information.

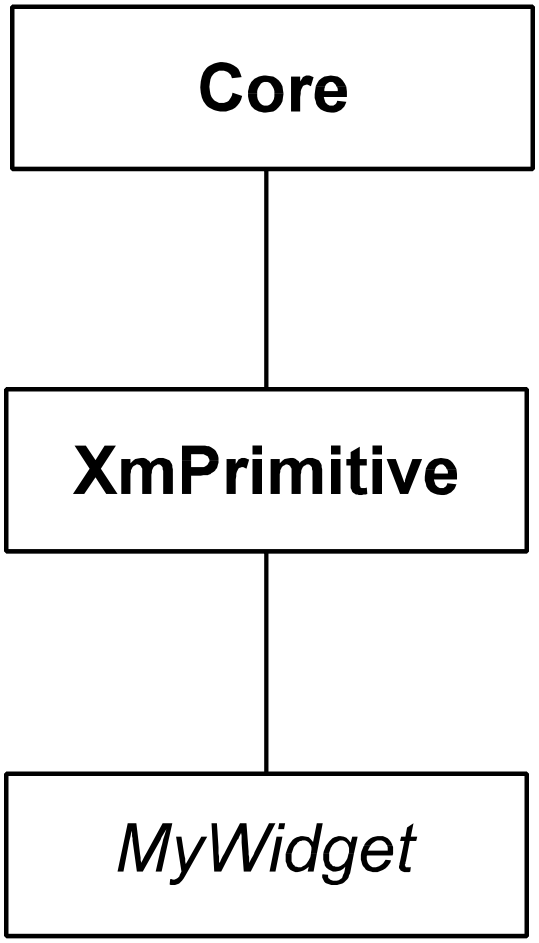
  - *XContexts* to store per Display information to be shared among multiple widgets.

- If forced to use globals or statics, protect them by either
  - Using *XtProcessLock*s (**XtProcessLock** and **XtProcessUnlock**) around all access points or
  - Using thread specific storage for globals that communicate state information across functions
- The exceptions to the above are Xt resource converters and *XtCallProc*, where it is safe to use statics. Note that resource converters should be invoked with a valid n heap address to store the converted value.
- The Xt base classes' class methods need to be protected since their function pointers can be dynamically written into by Motif. Copy the pointer into a stack variable and invoke the method through the stack variable. The copying needs to be done within *XtProcessLock*s. Use a similar strategy when accessing other fields in any class record (accesses of this nature are rare).
- Protect all public APIs with *XtAppLock*s. Call **XtAppLock** on entry into the function and **XtAppUnlock** when exiting.
- Maintain the locking hierarchy: Always acquire the *XtAppLock* first, then the *XtProcessLock*
- Investigate usage of private Event loops and make them MT-safe as described earlier.
- Copy out resource values if possible when an application does **XtGetValues** on the resource. Use the Xm synthetic resource mechanism or the *GetValuesHook* method to do this. If this cannot be done because it is inefficient, document these resources with appropriate warnings.
- Use MT safe versions of unsafe libC functions.
- Use conditional **#define**s to isolate all MT related changes, so that the code can be built non-MT-safe on platforms that do not support threading.

Chapter 3
The Widget Class Record of Primitive Widgets

The heart of every Intrinsics-based widget is its class record. The widget class record defines fields required by both the Intrinsics and Motif. The purpose of this chapter is to explain how to write a class record for subclasses of the **XmPrimitive** widget. Therefore, the widget class records described by this chapter will have the class hierarchy shown in the following figure.

Figure 3–1. Class Hierarchy of Primitive Widgets



Therefore, the class record of your primitive widget consists of

- All 32 fields from the **Core** widget
- All 7 fields from the **XmPrimitive** widget
- The extension record of the **XmPrimitive** widget
- All the fields unique to your own widget

This chapter details all these fields from a Motif widget writer's perspective. In addition, this chapter describes the instance data members of **XmPrimitive** that are accessible to your widget.

3.1 The CoreClassPart Structure

The top portion of the widget class record, which is the portion containing the 32 fields from **Core**, is called the **CoreClassPart** structure. For example, following is the **CoreClassPart** structure of the *ExmSimple* widget:

```
externaldef (exmsimpleclassrec) ExmSimpleClassRec exmSimpleClassRec = {
 { /* Here is the core class record. */
   /* superclass */                (WidgetClass)&xmPrimitiveClassRec,
   /* class_name */                "ExmSimple",
   /* widget_size */               sizeof(ExmSimpleRec),
   /* class_initialize */          ClassInitialize,
   /* class_part_initialize */     ClassPartInitialize,
   /* class_inited */              FALSE,
   /* initialize */                Initialize,
   /* initialize_hook */           NULL,
   /* realize */                   Realize,
   /* actions */                   NULL,
   /* num_actions */               0,
   /* resources */                 resources,
   /* num_resources */             XtNumber(resources),
   /* xrm_class */                 NULLQUARK,
   /* compress_motion */           TRUE,
   /* compress_exposure */         XtExposeCompressMaximal,
   /* compress_enterleave */       TRUE,
   /* visible_interest */          FALSE,
   /* destroy */                   Destroy,
   /* resize */                    Resize,
   /* expose */                    Redisplay,
   /* set_values */                SetValues,
   /* set_values_hook */           NULL,
   /* set_values_almost */         XtInheritSetValuesAlmost,
   /* get_values_hook */           NULL,
   /* accept_focus */              NULL,
   /* version */                    XtVersion,
   /* callback_private */           NULL,
   /* tm_table */                   XtInheritTranslations,
   /* query_geometry */             QueryGeometry,
   /* display_accelerator */        NULL,
   /* extension */                  NULL,
 },
```

These are the same 32 fields that every Intrinsics-based widget defines. A good Intrinsics book can explain what each of these fields means to the Intrinsics. However, we need to go beyond that. You need to explore what Motif widget writers need to know about these fields.

3.1.1 The superclass Field

The first field of the **CoreClassPart** structure is the *superclass* field. This field holds the name of the class record of your widget’s immediate superclass. If **XmPrimitive** is the immediate superclass of your widget, then your widget would specify the *superclass* field as follows:

```
/* superclass */                (WidgetClass)&xmPrimitiveClassRec,
```

3.1.2 The class_name Field

Set the **class_name** field to a string containing the name of the class; for example:

```
/* class_name */                "ExmSimple",
```

Motif will not make any attempt to interpret the characters in the string. However, it is a good idea to use alphanumeric characters only in the string. The string can be any length.

The string should begin with some prefix that uniquely symbolizes the widget group you are creating. For example, all the sample widgets associated with this group start with the **Exm** prefix. Do not pick a prefix that is used by other groups. For example, Motif reserves the prefixes *xm*, *Xm*, *_Xm*, *Xme*, *XmQT*, and *Exm*. See Chapter 2 for more information on namespace.

3.1.3 The widget_size Field

The **widget_size** field of primitive widgets holds the size of the full instance record. For example, the full instance record of the *ExmSimple* demonstration widget is named *ExmSimpleRec*; therefore, the Core class record of *ExmSimple* defines the **widget_size** field as follows:

```
/* widget_size */                sizeof(ExmSimpleRec);
```

3.1.4 The class_initialize Field

The **class_initialize** field holds the name of the widget's class initialization method. If your widget does not supply a class initialization method, you should set this field to *NULL*.

The Intrinsics call the class initialization method only once, at the first instantiation of a widget of this class. Most Motif widgets use the **class_initialize** method to register representation types. Motif provides a set of functions for registration, the most important of which is **XmRepTypeRegister**. For example, the **class_initialize** method of the *ExmSimple* widget registers the *ExmRSimpleShape* representation type as follows:

```
void
ClassInitialize( void)
{
  simpleShapeId = XmRepTypeRegister (ExmRSimpleShape, SimpleShapeNames,
                                     NULL, XtNumber(SimpleShapeNames));
}
```

See Chapter 6 for more information on representation types.

3.1.5 The class_part_initialize Field (Chained)

You can either set the **class_part_initialize** field to the name of your class part initialization method or you can set it to *NULL*. A *NULL* value indicates the absence of a class part initialization method.

This **class_part_initialize** method is chained; therefore, the Intrinsics call the **class_part_initialize** method of the **XmPrimitive** widget before calling the **class_part_initialize** method of your own widget. The **class_part_initialize** method of **XmPrimitive** contains code that allows its subclasses to inherit the following features of **XmPrimitive**. This is not to say that all primitive widgets automatically inherit these methods; the code merely makes it possible for any primitive widget that wants to inherit these features to be able to do so.

- **border_highlight** method
- **border_unhighlight** method
- **arm_and_activate** method
- **translations**

The **class_part_initialize** method of your own widget should provide the code that allows subclasses to inherit your widget's methods. For example, the *ExmSimple* widget allows its subclasses to inherit its **DrawVisual** and **DrawShadow** methods as follows:

```
if (wc->simple_class.draw_visual == ExmInheritDrawVisual)
 wc->simple_class.draw_visual = sc->simple_class.draw_visual;
if (wc->simple_class.draw_shadow == ExmInheritDrawShadow)
 wc->simple_class.draw_shadow = sc->simple_class.draw_shadow;
```

Many Motif widgets use the **class_part_initialize** method to install traits. For example, the **class_part_initialize** method of *ExmSimple* uses the following code to install the *XmQTcontainerItem* trait:

```
XmeTraitSet((XtPointer) wc, XmQTcontainerItem, (XtPointer) &simpleCIT);
```

XmeTraitSet is a routine that installs traits. (See Chapter 5 for complete details.) If you install a trait in your widget's **class_part_initialize** method, then this trait is automatically installed in all subclasses of your widget.

The **class_part_initialize** method of **XmPrimitive** installs the *XmQTcareParentVisual* trait and installs several undocumented traits.

3.1.6 The class_inited Field

All Motif widgets must set the value of this field to *False*.

3.1.7 The initialize Field (Chained)

The Intrinsics call your widget's *initialize* method when an application instantiates your widget. You typically specify one of the following for the *initialize* field:

- The name of your widget instance *initialize* method
- *NULL*, to indicate the absence of an *initialize* method

Since the *initialize* method is chained, the Intrinsics will call the *initialize* method of the **XmPrimitive** widget before calling the *initialize* method in your own widget. (**Core** does not provide an *initialize* method.) Broadly speaking, the *initialize* method of **XmPrimitive** does the following:

- Merges the keyboard traversal translations (specified in the *translations* field) with the regular widget translations (specified in the **tm_table** field). (See Chapter 7 for more details.)
- Helps implement the virtual key mechanism of Motif.
- Sets the values of the following two *primitive* flag variables to *False*: **primitive.have_traversal** and **primitive.highlighted**. Both variables are detailed later in this chapter.
- Validates the **XmNnavigationType** and **XmNunitType** resources.
- Initializes keyboard traversal.
- Establishes synthetic resources.
- Establishes a starting widget width or height if the user or application has not requested one. This is important for ensuring that window creation can happen safely whenever a subclass of **XmPrimitive** is realized.
- Creates the following three GCs: **primitive.highlight_GC**, **primitive.top_shadow_GC**, and **primitive.bottom_shadow_GC**.

The code in the *initialize* method of your own widget should check resource values and perform other start-up tasks that are relevant to your widget's creation. If any resources are expecting representation type values, you can validate the representation type values with **XmRepTypeValidValue**. For example, the *ExmSimple* widget contains one resource (*ExmNsimpleShape*) that expects an *ExmRSimpleShape* value. Therefore, the *initialize* method of *ExmSimple* validates the starting value of *ExmNsimpleShape* with the following code:

```
 if (!XmRepTypeValidValue (simpleShapeId,
                        nw->simple.simple_shape, (Widget)nw))
/* If the widget has been created without an appropriate starting
  value for XmNsimpleShape, force its starting value to ExmSHAPE_OVAL. */
 nw->simple.simple_shape = ExmSHAPE_OVAL;
```

3.1.8 The initialize_hook Field (Chained)

The **initialize_hook** field is obsolete. Motif ignores this value. Set it to *NULL*.

3.1.9 The realize Field

The *realize* field is responsible for creating the widget’s window. You typically specify either

- **XtInheritRealize**, to inherit the *realize* method of your superclass
- The name of a *realize* method defined by your widget

The *realize* method of **XmPrimitive** calls **XtCreateWindow** to generate a window. The window class of the created window will be **InputOutput**. The generated window will not propagate any button, key, or pointer motion events to its parent window(s). In order to inherit this code for your own widget, simply set the value of the *realize* field to **XtInheritRealize**. By so doing, your widget’s window will behave like other Motif primitive windows.

The *CDE 2.1/Motif 2.1—Style Guide and Glossary* does not demand that all windows conform to a certain standard. For that matter, the *CDE 2.1/Motif 2.1—Style Guide and Glossary* does not even insist that all windows be rectangular. Therefore, you are free to create your own *realize* method.

3.1.10 The actions Field

The *actions* field contains the name of the *actions* array. The *actions* array provides a correspondence between action names (as defined in the translations string) and action methods. For example, the *ExmCommandButton* demonstration widget defines the following *actions* array:

```
/* Declare the actions array. */
static XtActionsRec actions[] = {
 {"ExmCommandButtonEnter",           ExmCommandButtonEnter},
 {"ExmCommandButtonLeave",           ExmCommandButtonLeave},
 {"ExmCommandButtonArmAndActivate",  ExmCommandButtonArmAndActivate},
 {"ExmCommandButtonArm",             ExmCommandButtonArm},
 {"ExmCommandButtonActivate",        ExmCommandButtonActivate},
 {"ExmCommandButtonDisarm",          ExmCommandButtonDisarm}
};
```

The preceding actions array defines six actions.

If your widget does not define any action methods, then you should set the *actions* field to *NULL*.

(See Chapter 7 for more details about actions.)

3.1.11 The num_actions Field

The **num_actions** field holds the number of actions defined by the array in the *actions* field. If your widget does not provide an *actions* array, then you should specify 0 for the **num_actions** field. If your widget does provide an *actions* array, then you should use the **XtNumber** macro to count these actions. For example, the **ExmCommandButton** widget does provide an *actions* array, so its **num_actions** field looks as follows:

```
/* num_actions */                XtNumber(actions),
```

3.1.12 The resources Field

If your widget defines new resources, then the *resources* field must contain the name of the *resources* array. If your widget does not define any new resources, specify *NULL*.

(See Chapter 6 for information and recommendations.)

3.1.13 The num_resources Field

The **num_resources** field holds the number of resources defined by the array in the *resources* field. If your widget does not provide a *resources* array, then you should specify 0 for the **num_resources** field. If your widget does provide a *resources* array, then your widget should use the **XtNumber** macro to count these resources. For example, the **ExmSimple** widget does provide a *resources* array, so its **num_resources** field looks as follows:

```
/* num_resources */                XtNumber(resources),
```

3.1.14 The xrm_class Field

Your widget must always specify **NULLQUARK** as the value of the **xrm_class** field.

3.1.15 The compress_motion Field

The **compress_motion** field requires a **Boolean** value. Setting this value to **True** eliminates certain redundant pointer motion events from the event queue. Setting this value to **False** imposes a performance penalty but produces more accurate pointer tracking. (See documentation on the Intrinsics for more complete information.)

Motif makes no recommendation on the value of the **compress_motion** field. Within the standard Motif widget set, the value of the **compress_motion** field is generally **True**. The **XmPrimitive** widget sets this field to **True**.

3.1.16 The compress_exposure Field

The **compress_exposure** field requires a constant value. This value tells the Intrinsics how to respond to multiple exposure events. (See documentation on the Intrinsics for more complete information.)

Motif makes no recommendation on the value of the **compress_exposure** field. The **XmPrimitive** widget sets this field to **XtExposeCompressMaximal**, with the **XtExposeNoRegion** modifier enabled.

3.1.17 The compress_enterleave Field

The **compress_enterleave** field requires a **Boolean** value. Setting this value to **True** means that the Intrinsics will ignore unimportant enter event/leave event pairs. (See documentation on the Intrinsics for more complete information.)

Motif makes no recommendation on the value of the **compress_enterleave** field. However, most standard Motif primitive widgets (including the **XmPrimitive** widget itself) set this field to **True**.

3.1.18 The visible_interest Field

The **visible_interest** field requires a **Boolean** value. This value provides hints to the Intrinsics on the redisplay of partially exposed widgets. Motif does not layer any extra meaning on top of the standard Intrinsics meaning of this field.

Motif makes no recommendation on the value of this field. However, every standard Motif widget specifies **False** for this field.

3.1.19 The destroy Field (Chained)

The Intrinsics automatically call your widget's *destroy* method when your widget goes out of scope or is explicitly destroyed. The *destroy* method is chained up from subclass to superclass; therefore, the *destroy* method of **XmPrimitive** will be called after any *destroy* method that your own widget supplies. The *destroy* method of **XmPrimitive** deallocates the following three graphics contexts:

- **primitive.top_shadow_GC**
- **primitive.bottom_shadow_GC**
- **primitive.highlight_GC**

In addition, the *destroy* method of **XmPrimitive** deallocates all the widget traversal information held by this widget.

If your widget provides a *destroy* method (as it typically should), make sure it deallocates all strings, fonts, GCs, and so on, that your widget allocates.

If you do not want your widget to supply a *destroy* method, set the *destroy* field to *NULL*.

3.1.20 The resize Field

The Intrinsics automatically call the function specified in the *resize* field in response to a resize action from the parent widget. The **XmPrimitive** widget does not provide a *resize* method, so your widget cannot inherit from it.

The *resize* method of your widget lays out the widget’s visual components. One of the duties of the *resize* method is to determine what parts of the widget to display when there is not enough space to display everything. Chapter 12 details Motif’s recommendations for handling this situation. Sample code that demonstrates these recommendations is available in the *ExmSimple* and *ExmString* widgets.

3.1.21 The expose (Redisplay) Field

The Intrinsics automatically invoke the *expose* method upon receiving an exposure event. The interpretation of exposure events can be altered by the values of the **compress_exposure** and **visible_interest** fields. (See documentation on the Intrinsics for details.)

Your widget should set the *expose* field to one of the following:

- **XtInheritExpose**, to indicate that you are inheriting the *expose* method of your superclass. The **XmPrimitive** widget provides an *expose* method, and your primitive widget can inherit this method.
- The name of your widget's *expose* method. If you do provide an *expose* method, you should name it **Redisplay**.

All standard Motif primitive widgets provide their own *expose* method. However, you should not feel obligated to write an expose method for your own widget. In fact, most of the primitive widgets in the Exm demonstration widget set inherit the expose method of *ExmSimple*.

This **Redisplay** method of **XmPrimitive** examines the value of the **Boolean** field **primitive.highlighted**. If this field is set to **True**, **Redisplay** calls the **border_highlight** method of **XmPrimitive**. If this field is **False**, the expose method calls the **border_unhighlight** method of **XmPrimitive**. (The **border_highlight** and **border_unhighlight** methods are detailed later in this chapter.) In brief, **border_highlight** and **border_unhighlight** of **XmPrimitive** provide Motif-appropriate border highlighting and unhighlighting.

Your **Redisplay** method should do the following:

- Call the method responsible for redrawing the visible portions of the widget. For example, the **Redisplay** method of *ExmSimple* calls the **DrawVisual** method. In *ExmSimple*, the **DrawVisual** method renders a geometric shape.
- Call the method responsible for drawing shadow, if appropriate. The *CDE 2.1/Motif 2.1—Style Guide and Glossary* should help you determine whether shadows are appropriate for your widget. For example, shadows are appropriate for the **ExmSimple** widget. Therefore, the expose method of **ExmSimple** calls its **DrawShadow** method. The **DrawShadow** method of **ExmSimple** calls *XmeDrawShadow* to draw a Motif-style shadow. See Chapter 17 for syntactic details on the *XmeDrawShadow* and **XmeDrawPolygonShadow** routines.
- Call the method responsible for handling border highlights, if appropriate. The *CDE 2.1/Motif 2.1—Style Guide and Glossary* should help you determine whether border highlights are appropriate for your widget. You may write your own method to handle border highlights; however, we recommend that you call the *expose* method of **XmPrimitive** instead.

See Chapter 12 for more details on the *expose* method.

3.1.22 The set_values Field (Chained)

The Intrinsics automatically call the **set_values** method whenever an application calls **XtSetValues** to change any of the resources in your widget.

You typically specify one of the following for the **set_values** field:

- The name of your widget's **set_values** method
- *NULL*, which symbolizes the absence of a **set_values** method

Most of the widgets you write will specify a **set_values** method.

Your widget's **set_values** method is responsible for validating changes to resource values. Your widget can call **XmRepTypeValidValue** to validate changes to representation type resources. (See Chapter 6 for details.)

The code in a **set_values** method is usually quite similar to the code in an *initialize* method.

The **set_values** method returns a **Boolean** value. The Intrinsics monitor this returned value. If the returned value is **True**, the Intrinsics automatically invoke the widget's *expose* (**Redisplay**) method. There are many reasons why widget redisplay is necessary. For example, one subtle reason is that a widget GC has changed.

The **set_values** method is chained. Therefore, the Intrinsics will call the **set_values** method of **XmPrimitive** prior to calling the **set_values** method of your own widget. The **set_values** method of **XmPrimitive**:

- Validates changes to **XmNnavigationType** and **XmNunitType**.
- Updates keyboard traversals.
- Examines the **primitive.shadow_thickness** and **primitive.highlight_thickness** fields. If either field's value has changed, **set_values** sets its return value to **True**.
- Examines the **primitive.highlight_color** and **primitive.highlight_pixmap** fields. If either field's value has changed, **set_values** destroys the existing **primitive.highlight_GC** and then creates a new one from the updated values. If either field's value has changed, **set_values** sets its return value to **True**.
- Examines the **primitive.top_shadow_color** and **primitive.top_shadow_pixmap** fields. If either field's value has changed, **set_values** destroys the existing

primitive.top_shadow_GC and then creates a new one from the updated values. If either field's value has changed, **set_values** sets its return value to **True**.

- Examines the **primitive.bottom_shadow_color** and **primitive.bottom_shadow_pixmap** fields. If either field's value has changed, **set_values** destroys the existing **primitive.bottom_shadow_GC** and then creates a new one from the updated values. If either field's value has changed, **set_values** sets its return value to **True**.
- Calls the **border_unhighlight** method when all of the following conditions are **True**:
  - The border highlight is drawn or the widget is insensitive.
  - The focus policy is *XmImplicit*.
  - The value of the **XmNhighlightOnEnter** resource changes from **True** to **False**.

3.1.23 The set_values_hook Field (Chained)

This is an obsolete field; Motif ignores its value. Your widget should set it to *NULL*.

3.1.24 The set_values_almost Field

The Intrinsics call the method named in the **set_values_almost** field when certain geometry requests cannot be fulfilled by the parent geometry manager.

You should probably set the **set_values_almost** field to one of the following:

- The name of your widget's **set_values_almost** field.
- **XtInheritSetValuesAlmost**, to inherit the **set_values_almost** method of your superclass.

The **XmPrimitive** widget itself sets the **set_values_almost** field to **XtInheritSetValuesAlmost**, meaning that it inherits the **set_values_almost** method of **Core**.

Motif makes no recommendations for this method. However, all standard Motif primitive widgets except **XmLabel** and **XmForml**set this field to **XtInheritSetValuesAlmost**.

3.1.25 The get_values_hook Field (Chained)

The **get_values_hook** field holds the name of a method responsible for doing preprocessing on **XtGetValues** calls. If your widget has no need to do this, you should set the **get_values_hook** field to *NULL*. In fact, most standard Motif primitive widgets do set this field to *NULL*.

The **get_values_hook** field is chained; therefore, the Intrinsics will call the **get_values_hook** method of **XmPrimitive** prior to the **get_values_hook** method of your widget. The **get_values_hook** method of **XmPrimitive** implements synthetic resources for primitive widgets. (See Chapter 6 for details on synthetic resources.)

3.1.26 The accept_focus Field

Motif ignores the value of the **accept_focus** field; set it to *NULL*.

3.1.27 The version Field

The *version* field holds an enumerated constant pertaining to the compatibility of the Intrinsics themselves. The *version* field does not express any information regarding Motif compatibility, nor does Motif interpret the version field in any way.

Most widgets set this field to **XtVersion**. However, if you do not want your widget to check binary compatibility with future releases of the Intrinsics, set this field to **XtVersionDontCheck**. (See Chapter 15 for details on binary compatibility.)

3.1.28 The callback_private Field

Motif ignores the value of the **callback_private** field; you should set this field to *NULL*.

3.1.29 The tm_table Field

The **tm_table** field holds the primary translations string of your widget. Your widget should probably set this field to one of the following:

- The name of your widget's **tm_table** translations string.
- **XtInheritTranslations**, to inherit the **tm_table** translations of your superclass. The **XmPrimitive** widget does not provide any **tm_table** translations to inherit.

If your widget does not provide any translations, Motif recommends setting the **tm_table** field to **XtInheritTranslations**.

(See Chapter 7 for more details.)

3.1.30 The query_geometry Field

The Intrinsics call the **query_geometry** method when another widget (most likely the parent) or an application calls **XtQueryGeometry** to determine your widget's geometry preferences.

Your widget must specify one of the following in the **query_geometry** field:

- The name of your widget's **query_geometry** method.
- **XtInheritQueryGeometry**, to inherit the **query_geometry** method of your superclass. The **XmPrimitive** widget does not provide a **query_geometry** method to inherit.
- *NULL*, which indicates the absence of a **query_geometry** method.

(See Chapter 12 for details on geometry management.)

3.1.31 The display_accelerator Field

Motif ignores the value of the **display_accelerator** field; set it to *NULL*.

3.1.32 The Base Class extension Field

Motif strongly recommends that you set the value of the *extension* field to *NULL*. By so doing, Motif will install the correct *extension* record for you at runtime.

3.2 The PrimitiveClassPart Structure

The **PrimitiveClassPart** structure is the second chunk of the widget class record. In the **PrimitiveClassPart** structure, you must define values for seven different fields. For example, following is the **PrimitiveClassPart** structure of the *ExmSimple* widget:

```
{
 /* border_highlight */          XmInheritBorderHighlight,
 /* border_unhighlight */        XmInheritBorderUnhighlight,
 /* translations */              XtInheritTranslations,
 /* arm_and_activate */          NULL,
 /* syn_resources */             syn_resources,
 /* num_syn_resources */         XtNumber(syn_resources),
 /* extension */                 (XtPointer)&primClassExtRec,
},
```

The following subsections detail the fields of the **PrimitiveClassPart** structure.

3.2.1 The border_highlight Field

The **border_highlight** field holds the name of a method that highlights the perimeter of the widget. The **border_highlight** method is typically called by the **Redisplay** method.

You should probably set the **border_highlight** field to one of the following:

- The name of your widget's **border_highlight** method
- **XmInheritBorderHighlight**, to indicate that you are inheriting the **border_highlight** method of your superclass

Most primitive widgets should inherit the **border_highlight** method of **XmPrimitive**. The **border_highlight** method of **XmPrimitive** provides the standard Motif highlighting style for primitive widgets. In particular, the **border_highlight** method of **XmPrimitive** does the following:

- Sets the value of internal field **primitive.highlighted** to **True**.
- Examines the widget's dimensions and resources to determine whether or not the widget needs a border. For example, there is no need to draw a border if the value of the **XmNhighlightThickness** resource is 0.
- If the widget does need a border, the method draws it. The drawing characteristics (color, pixmap, and so forth) are governed by the GC stored in instance data member **primitive.highlight_GC**.

If you do write your own **border_highlight** method, it must have the following prototype:

```
void BorderHighlight(
        Widget  w)
```

where *widget* specifies the widget ID.

If you do write your own **border_highlight** method, you should probably use the **XmeDrawHighlight** function to draw the border highlight. In addition, your **border_highlight** method must set the value of instance data member **primitive.highlighted** to **True** immediately after drawing the border highlight.

The *ExmTabButton* and *ExmMenuButton* widgets both have special requirements for border highlights. Therefore, both of these widgets provide their own **border_highlight** methods rather than inherit them.

3.2.2 The border_unhighlight Field

The **border_unhighlight** field holds the name of a method that removes a widget's border. For example, if your primitive widget has a different background color than its parent, then it may be wise to remove the border (possibly in preparation for a redraw of the border). You should probably set the **border_unhighlight** field to one of the following:

- **XmInheritBorderUnhighlight**, to inherit the **border_unhighlight** method of your superclass. The **XmPrimitive** widget provides a **border_unhighlight** method, which your primitive widget can inherit.
- The name of your widget’s **border_unhighlight** method.

If the **border_highlight** field of your widget is set to **XmInheritBorderHighlight**, then you should set the **border_unhighlight** field to **XmInheritBorderUnhighlight**.

The **border_unhighlight** method of **XmPrimitive** does the following:

- Sets the value of internal field **primitive.highlighted** to **False**.
- Examines the widget’s dimensions and resources to determine whether or not there is indeed a border to erase.
- If the widget does have a border, the **border_unhighlight** method calls an internal function that unhighlights the border.

If you do write your own **border_unhighlight** method, it must have the following prototype:

```
void BorderUnhighlight(
        Widget  w)
```

where *widget* specifies the widget ID.

If your widget provides its own **border_unhighlight** method, then it must set the value of instance data member **primitive.highlighted** to **False**. To remove highlights, your method should probably call the same internal function that drew the highlights, namely, **XmeDrawHighlight** . One trick to unhighlighting is to paint over the existing highlight with the background color and background pixmap of your widget’s parent. Your widget’s parent stores its background color and background pixmap in the **manager.background_GC** instance data member.

3.2.3 The translations Field

The *translations* field holds a string that implements your widget's keyboard traversals. The complete keyboard traversal policy depends not only on the value of the *translations* field but also on the values of two resources.

You can set this field to one of the following:

- **XtInheritTranslations**, to inherit the traversal translations string of your superclass. The **XmPrimitive** widget provides a traversal translations string that your widget can inherit.
- The name of your widget's own traveral translations string.
- *NULL*, to indicate the absence of any traversal translations.

(See Chapter 7 for details.)

3.2.4 The arm_and_activate Field

The **arm_and_activate** field holds the name of a method that is activated by an osfActivate event. The named method must provide the visual simulation of a push button being pushed in as a response to the osfActivate event.

Your widget should specify one of the following for the **arm_and_activate** field:

- The name of your widget's **arm_and_activate** method.
- **XmInheritArmAndActivate**, to inherit the **arm_and_activate** method of your superclass. The **XmPrimitive** widget does not provide an **arm_and_activate** method, so you cannot inherit from it.
- *NULL*, to indicate the absence of an **arm_and_activate** method.

In the standard Motif widget set, all the button widgets provide an **arm_and_activate** method. All other primitive widgets set this field to *NULL*.

The *ExmCommandButton* widget provides the following **arm_and_activate** method:

```
static void
ArmAndActivate (
       Widget w,
       XEvent *event,
       String *params,
       Cardinal *num_params)
{
   ExmCommandButtonWidgetClass wc = (ExmCommandButtonWidgetClass)XtClass(w);
   ExmCommandButtonWidget cw = (ExmCommandButtonWidget)w;
   XmAnyCallbackStruct cb;
   /* Call DrawShadow. */
     cw->command_button.visual_armed = True;
     if (wc->simple_class.draw_shadow)
       (*(wc->simple_class.draw_shadow)) (w);

/* Eliminate any data waiting in the X Window’s output buffer. */
  XFlush (XtDisplay (cw));

/* Assuming that the XmNactivateCallback resource is set, call the
   callback routine. */
  if (cw->command_button.activate_callback) {
    cb.reason = XmCR_ACTIVATE;
    cb.event = event;
    XtCallCallbackList((Widget)cw, cw->command_button.activate_callback, &cb);
  }

/* Provide a short delay prior to the appearance of any new windows created
   by the callback. The net effect is that the ExmCommandButton will appear
   to flash on and off immediately prior to the appearance of any window that
   may overwrite it. */
  if ((cw->core.being_destroyed == False) &&
      (cw->command_button.visual_timer == 0))
    cw->command_button.visual_timer =
      XtAppAddTimeOut(XtWidgetToApplicationContext((Widget)cw),
                    (unsigned long) VISUAL_DELAY,
                     VisualDisarm, /* method that disarms */
                    (XtPointer)(cw)); /* pass this widget to VisualDisarm */
}
```

3.2.5 The syn_resources Field

The **syn_resources** field holds the name of the array that defines the widget's synthetic resources. Synthetic resources provide a mechanism for translating widget resource values between different formats. (See Chapter 6 for information on synthetic resources.)

Most primitive widgets in the standard Motif widget set provide a **syn_resources** array.

If you do not wish to define synthetic resources for your widget, set the value of this field to *NULL*.

3.2.6 The num_syn_resources Field

The **num_syn_resources** field holds the number of synthetic resources defined by the array in the **syn_resources** field. If your widget does not provide a **syn_resources** array, then you should specify 0 for the **num_syn_resources** field. If your widget does provide a **syn_resources** array, then your widget should use the the **XtNumber** macro to count these resources.

3.2.7 The extension Field

Nearly every primitive widget displays text or graphics. Any widget that does must specify a primitive class extension record. The *extension* field holds an *XtPointer* to the name of this record. You should always name your primitive class extension record **primClassExtRec**; for example:

```
/* extension */                 (XtPointer)&primClassExtRec,
```

If your primitive widget does not display any text or graphics, then you should set the *extensions* field to *NULL*. If your primitive widget does need to define an extension record, see the next section for details.

3.3 The Primitive Class Extension Record

Nearly every primitive widget should define a primitive class extension record. The primitive class extension record currently consists of seven fields. Future releases may add more fields.

The following is the primitive class extension record for the *ExmString* widget:

```
/* Here is the primitive class extension record. */
static XmPrimitiveClassExtRec primClassExtRec = {
   /* next_extension */            NULL,
   /* record_type */               NULLQUARK,
   /* version */                   XmPrimitiveClassExtVersion,
   /* record_size */               sizeof(XmPrimitiveClassExtRec),
   /* widget_baseline */           WidgetBaselines,
   /* widget_display_rect */       WidgetDisplayRect,
   /* widget_margins */            NULL,
};
```

The following subsections detail each of the fields in this structure.

3.3.1 The next_extension Field

Your widget should always set the **next_extension** field to *NULL*.

3.3.2 The record_type Field

Your widget should always set the **record_type** field to **NULLQUARK**.

3.3.3 The version Field

Your widget must set the *version* field to **XmPrimitiveClassExtVersion**.

3.3.4 The record_size Field

The **record_size** field holds the size of the primitive class extension record. You should set it as follows:

```
/* record_size */                sizeof(XmPrimitiveClassExtRec),
```

3.3.5 The widget_baseline Field

Your widget must set the **widget_baseline** field to one of the following:

- *NULL*, to indicate the absence of a **widget_baseline** method. Set the field to *NULL* if your widget does not render text to the screen.
- **XmInheritBaselineProc**, to inherit the **widget_baseline** method of your widget's superclass. **XmPrimitive** does not provide a **widget_baseline** method, so you cannot inherit from it.
- The name of your widget's **widget_baseline** method.

If your widget renders text, it must provide or inherit a **widget_baseline** method. The purpose of this method is to create a table of text baselines. That is, each entry in the table must hold the baseline of a different line of text. For example, a widget that displays six lines of text would require a six-entry baseline table. The baseline of any given line of text is a vertical offset in pixels from the origin of the widget's bounding box to the given baseline.

Several manager widgets use baseline tables. The **RowColumn** widget, for example, uses the information in the baseline tables of its children widgets to align text properly. The baseline tables are also used by the **XmWidgetGetBaselines** and **XmTextGetBaseline** functions.

In order to be compatible with other Motif widgets and functions, your **widget_baseline** method must have the following prototype:

```
Boolean WidgetBaseline(
        Widget          w,
        Dimension       **baselines,
        int             *line_count)
```

Upon the function's completion, the following must occur:

- The *baselines* variable must return a pointer to an array that contains the value of each baseline of text in the widget. Note that your widget's **widget_baseline** method must call **XtMalloc** to allocate space to hold the baselines. Conversely, the caller of the **widget_baseline** method must call **XtFree** to free the allocated space.
- The *line_count* variable must return the number of lines in the widget. You may want to use the **XmStringLineCount** function to determine the line count.
- The method itself must return a **Boolean** value that indicates whether the widget contains a baseline. If the value is **True**, the function returns a value for each baseline of text. If it is **False**, the function was unable to return a baseline value.

(See the **ExmString** demonstration widget for an example of a **widget_baseline** method.)

Several standard Motif widgets, including **XmText**, **XmTextField**, **XmLabel**, and **XmIconGadget** provide a **widget_baseline** method.

3.3.6 The widget_display_rect Field

Your widget must set the **widget_display_rect** field to one of the following:

- *NULL*, to indicate the absence of a **widget_display_rect** method. Set the field to *NULL* if your widget does not render any graphics, pixmaps, or text to the screen.
- **XmInheritDisplayRectProc**, to inherit the **widget_display_rect** method of your widget's superclass. **XmPrimitive** does not provide a **widget_display_rect** method, so you cannot inherit from it.
- The name of your widget's **widget_display_rect** method.

If your widget does render graphics, pixmaps, or text, then it should provide a **widget_display_rect** method. The purpose of this method is to define the bounding rectangle of the visible information inside the widget. The bounding rectangle is stored in an *XRectangle* structure. The *x* and *y* coordinates that are stored inside this structure represent offsets from the upper left of the widget.

If a widget displays a pixmap, then the bounding rectangle defines the perimeter of the pixmap. If the widget displays text, then the first text character marks the upper left of the bounding rectangle. The coordinates of the lower right of the bounding rectangle are defined by the last line of text and the longest line of any text in the widget. If the widget displays graphics, then the bounding rectangle is undefined. However, if the graphics area spans a clearly defined rectangle, then this rectangle should mark the range of the bounding box.

Several Motif managers use the **widget_display_rect** method to determine how to align their children. In addition, Motif applications can access the **widget_display_rect** method by calling the **XmWidgetGetDisplayRect** function.

Your **widget_display_rect** method must have the following prototype:

```
Boolean WidgetDisplayRect(
        Widget  w,
        XRectangle      *displayrect)
```

Upon the function's completion, the *displayrect* variable must describe the dimensions of the bounding rectangle. (See the **XmWidgetGetDisplayRect**(3) reference page of the *Motif 2.1—Programmer's Reference* for details on the returned bounding rectangle.)

For an example of a complete **widget_display_rect** method, see the demonstration widgets **ExmSimple** or **ExmString**.

Several standard Motif widgets provide a **widget_display_rect** method, including **XmText**, **XmTextField**, **XmLabel**, and **XmIconGadget**.

3.3.7 The widget_margins Method

You should set the **widget_margins** field to *NULL*.

3.4 The Instance Data Members of XmPrimitive

The **XmPrimitive** widget maintains the values of its resources inside the fields of the **primitive** structure. For example, the value of the **XmNhighlightThickness** resource is stored in the **primitive.highlight_thickness** field. Your primitive widget can access any of these fields. For example, the following code accesses the **primitive.highlight_thickness** field:

```
total_width_of_highlight = primitive.highlight_thickness * 2;
```

In addition to all the resource values, the **primitive** structure also holds several fields that are not tied to resources. Widget writers should become familiar with the fields shown in the following subsections.

3.4.1 The primitive.have_traversal Field

The **have_traversal** field is declared as follows:

```
Boolean   have_traversal;
```

This is a flag whose initial value is **False**. This flag becomes **True** only when both of the following happen:

- The keyboard focus policy is **XmEXPLICIT**.
- The widget gets the keyboard focus.

When the widget loses keyboard focus, this flag is reset to **False**. Similarly, if the keyboard focus policy becomes **XmPOINTER**, this flag is reset to **False**.

3.4.2 The primitive.highlighted Field

The *highlighted* field is declared as follows:

```
Boolean   highlighted;
```

This is a flag. Widgets must set this flag to **True** after drawing a highlight around the widget's border. Widgets must set this flag to **False** after removing the highlight. The flag's initial value is **False**.

3.4.3 The primitive.highlight_GC Field

The **highlight_GC** field is declared as follows:

```
GC        highlight_GC;
```

The **border_highlight** method of the **XmPrimitive** widget uses this GC when rendering the border highlight.

The **primitive.highlight_GC** contains the default GC generated by **XtGetGC**, except for the following differences:

- **XmPrimitive** sets the *foreground* field of the GC to the value of the **XmNhighlightColor** resource of **XmPrimitive**.
- **XmPrimitive** sets the *background* field of the GC to the value of the **XmNbackground** resource of **Core**.

If the **XmNhighlightPixmap** resource of **XmPrimitive** contains something other than **XmUNSPECIFIED_PIXMAP** or **None**, then the **highlight_GC** also contains the following:

- If the depth of the pixmap in **XmNhighlightPixmap** is 1, then the **fill_style** field of the GC is set to **FillOpaqueStipple**. Furthermore, the *stipple* field of the GC is set to the pixmap in **XmNhighlightPixmap**.
- If the depth of the pixmap in **XmNhighlightPixmap** is greater than 1, then the **fill_style** field of the GC is set to **FillTiled**. Furthermore, the *tile* field of the GC is set to the pixmap in **XmNhighlightPixmap**.

3.4.4 The primitive.top_shadow_GC Field

The **top_shadow_GC** field is declared as follows:

```
GC        top_shadow_GC;
```

XmPrimitive uses **primitive.top_shadow_GC** when rendering the top shadow. The **primitive.top_shadow_GC** field contains the default GC generated by **XtGetGC**, except for the following differences:

- **XmPrimitive** sets the *foreground* field of the GC to the value of the **XmNtopShadowColor** resource of **XmPrimitive**.
- **XmPrimitive** sets the *background* field of the GC to the value of the **XmNbackground** resource of **Core**.

If the **XmNtopShadowPixmap** resource of **XmPrimitive** contains something other than **XmUNSPECIFIED_PIXMAP** or **None**, then the **highlight_GC** also contains the following:

- If the depth of the pixmap in **XmNtopShadowPixmap** is 1, then the **fill_style** field of the GC is set to **FillOpaqueStipple**. Furthermore, the *stipple* field of the GC is set to the pixmap in **XmNtopShadowPixmap**.
- If the depth of the pixmap in **XmNtopShadowPixmap** is greater than 1, then the **fill_style** field of the GC is set to **FillTiled**. Furthermore, the *tile* field of the GC is set to the pixmap in **XmNtopShadowPixmap**.

3.4.5 The primitive.bottom_shadow_GC Field

The **bottom_shadow_GC** field is declared as follows:

```
GC        bottom_shadow_GC;
```

XmPrimitive uses **primitive.bottom_shadow_GC** when rendering the bottom shadow. The **primitive..bottom_shadow_GC** field contains the default GC generated by **XtGetGC**, except for the following differences:

- **XmPrimitive** sets the *foreground* field of the GC to the value of the **XmNbottomShadowColor** resource of **XmPrimitive**.
- **XmPrimitive** sets the *background* field of the GC to the value of the **XmNbackground** resource of **Core**.

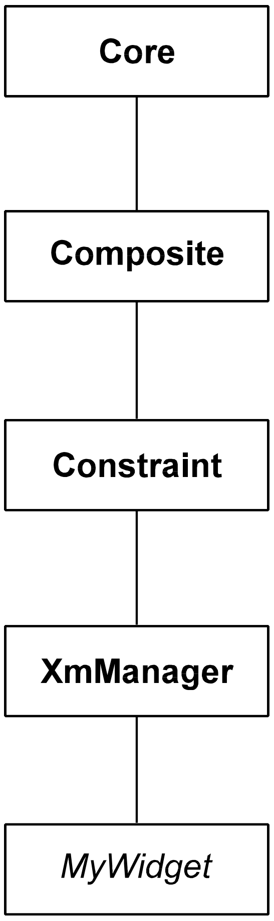
If the **XmNbottomShadowPixmap** resource of **XmPrimitive** contains something other than **XmUNSPECIFIED_PIXMAP** or **None**, then the **highlight_GC** also contains the following:

- If the depth of the pixmap in **XmNbottomShadowPixmap** is 1, then the **fill_style** field of the GC is set to **FillOpaqueStipple**. Furthermore, the *stipple* field of the GC is set to the pixmap in **XmNbottomShadowPixmap**.
- If the depth of the pixmap in **XmNbottomShadowPixmap** is greater than 1, then the **fill_style** field of the GC is set to **FillTiled**. Furthermore, the *tile* field of the GC is set to the pixmap in **XmNbottomShadowPixmap**.

Chapter 4
The Widget Class Record of Manager Widgets

This chapter explains how to write a class record for subclasses of the **XmManager** widget. As shown in Figure 4-1, the manager widgets you write will descend from the **Core**, **Composite**, **Constraint**, and **XmManager** widgets.

Figure 4–1. Class Hierarchy of Manager Widgets



Therefore, the class record of a manager widget must consist of:

- All 32 fields from the **Core** widget
- All 5 fields from the **Composite** widget
- Optionally, the fields of the extension record of the **Composite** widget
- All 7 fields from the **Constraints** widget
- Optionally, the fields of the extension record of the **Constraint** widget

- All 7 fields from the **XmManager** widget
- Optionally, the fields of the extension record of the **XmManager** widget
- All the fields unique to your own manager widget

This chapter details all these fields from a Motif widget writer’s perspective. In addition, this chapter describes the data members of **XmManager** that are accessible to your widget.

4.1 The CoreClassPart Structure

Your manager widget must contain a class record. The first portion of the class record is a **CoreClassPart** structure. For example, following is the **CoreClassPart** structure of the *ExmGrid* widget:

```
externaldef(exmgridclassrec) ExmGridClassRec exmGridClassRec =
{
  {                     /* core_class */
    /* superclass */                (WidgetClass) &xmManagerClassRec,
    /* class_name */                "ExmGrid",
    /* widget_size */               sizeof(ExmGridRec),
    /* class_initialize */          NULL,
    /* class_part_initialize */     ClassPartInitialize,
    /* class_inited */              FALSE,
    /* initialize */                Initialize,
    /* initialize_hook */           NULL,
    /* realize */                   XtInheritRealize,
    /* actions */                   NULL,
    /* num_actions */               0,
    /* resources */                 resources,
    /* num_resources */             XtNumber(resources),
    /* xrm_class */                 NULLQUARK,
    /* compress_motion */           TRUE,
    /* compress_exposure */         XtExposeCompressMaximal,
    /* compress_enterleave */       TRUE,
    /* visible_interest */          FALSE,
    /* destroy */                   NULL,
    /* resize */                    Resize,
    /* expose */                    Redisplay,
    /* set_values */                SetValues,
    /* set_values_hook */           NULL,
    /* set_values_almost */         XtInheritSetValuesAlmost,
    /* get_values_hook */           NULL,
    /* accept_focus */              NULL,
    /* version */                   XtVersion,
    /* callback_private */          NULL,
    /* tm_table */                  XtInheritTranslations,
    /* query_geometry */            QueryGeometry,
    /* display_accelerator */       NULL,
    /* extension */                 NULL,
  },
```

The following subsections explain what Motif widget writers need to know about these fields.

4.1.1 The superclass Field

The first field of the **CoreClassPart** structure is the *superclass* field. For this field, you must specify the name of the class record of your widget’s immediate superclass. For example, if your widget’s immediate superclass is **XmManager**, the superclass field will look as follows:

```
/* superclass */                (WidgetClass) &xmManagerClassRec,
```

4.1.2 The class_name Field

The **class_name** field holds a string representing the name of the class; for example:

```
/* class_name */                "ExmGrid",
```

Motif will not make any attempt to interpret the characters in the string. The string can be any length, and can be composed of alphanumeric character, hypens, and underbars.

The string should begin with some prefix that uniquely symbolizes the widget group you are creating. For example, all the sample widgets associated with this group start with the **Exm** prefix. Do not pick a prefix that is used by other groups. For example, Motif reserves the prefixes *xm*, *Xm*, *_Xm*, *Xme*, *XmQT*, and *Exm*. See Chapter 2 for more information on namespace.

4.1.3 The widget_size Field

The **widget_size** field of manager widgets holds the size of the full instance record. For example, the full instance record of the *ExmGrid* demonstration widget is named *ExmGridRec*; therefore, the Core class record of *ExmGrid* defines the **widget_size** field as follows:

```
/* widget_size */                sizeof(ExmGridRec);
```

4.1.4 The class_initialize Field

The Intrinsics call the **class_initialize** method only once, at the first instantiation of a widget of this class. You can set the **class_initialize** field to one of the following:

- The name of your widget's **class_initialize** method.
- *NULL*, to indicate the absence of a class initialization method. (About half the standard Motif manager widgets set this field to *NULL*.)

Motif makes no requirements of the code in **class_initialize** methods. However, Motif recommends registering any new representation types inside the **class_initialize** method. (See Chapter 6 for details.)

4.1.5 The class_part_initialize Field (Chained)

The Intrinsics call the **class_part_initialize** method at the first instantiation of a widget of this class or of a subclass of this class. You can set the **class_part_initialize** field to one of the following:

- The name of your widget's **class_part_initialize** method.
- *NULL*, indicating the absence of a **class_part_initialize** method.

The **class_part_initialize** method is chained; therefore, the Intrinsics call the **class_part_initialize** method of **Core**, **Composite**, **Constraint**, and **XmManager** before calling the **class_part_initialize** method of your own widget. (See documentation on the Intrinsics for information on the **class_part_initialize** methods of **Core**, **Composite**, and **Constraint**.) The **class_part_initialize** method of **XmManager** does the following:

- If your widget does not specify a **Composite** class extension record, **XmManager** creates one for you by copying the **Composite** class extension record of your widget's superclass.
- If your widget does not specify an **XmManager** class extension record, **XmManager** creates one for you by copying the **XmManager** class extension record of your widget's superclass.
- Initializes synthetic resource management
- Establishes the class *translations* field, if your widget specifies **XtInheritTranslations**

Motif makes no requirements of the code in your widget's **class_part_initialize** method. However, most Motif widgets use this method to install traits. (See Chapter 5 for details on installing traits.)

4.1.6 The class_inited Field

Motif widgets must always set the value of the **class_inited** field to False.

4.1.7 The initialize Field (Chained)

The Intrinsics call your widget's *initialize* method when an application instantiates your widget. You must specify one of the following:

- The name of your *initialize* method
- *NULL*, to indicate the absence of an *initialize* method.

The *initialize* method is chained; therefore, the Intrinsics call the *initialize* method of the **Core**, **Composite**, **Constraint**, and **XmManager** widgets before calling the *initialize* method of your own widget. (See documentation on the Intrinsics for details on the *initialize* method of **Core**, **Composite**, and **Constraint**.) The *initialize* method of **XmManager** does the following:

- Initializes values for certain instance data members of **XmManager**.
- Validates the **XmNnavigationType**, **XmNstringDirection**, and **XmNunitType** resources.
- Merges the keyboard traversal translations (specified in the *translations* field) with the regular widget translations (specified in the **tm_table** field). The *initialize* method places the traversal translations at the top of the translations table. Therefore, if a keyboard traversal translation and a regular widget translation both define an action for the same virtual key, then the action of the keyboard traversal translation takes precedence.
- Initializes keyboard traversal.
- Establishes synthetic resources.
- Creates four GCs—**manager.highlight_GC**, **manager.top_shadow_GC**, **manager.bottom_shadow_GC**, and **manager.background_GC**—by calling GC creation methods stored in **XmManager**.
- Provides a mechanism for automatic inheritance of accelerators.

The code in your own *initialize* method should check resource values and perform other start-up tasks relevant to your widget's creation. You can find a sample *initialize* method for a manager widget inside the **Grid.c** file.

4.1.8 The initialize_hook Field (Chained)

The **initialize_hook** field is obsolete. Motif ignores the stored value of this field; set it to *NULL*.

4.1.9 The realize Field

The *realize* field holds the name of the method that is responsible for creating the widget's window. If you want a standard Motif manager window, you should inherit the *realize* method of the **XmManager** widget by setting the *realize* field to **XtInheritRealize**.

The *realize* method of **XmManager** calls **XtCreateWindow** to create a window. It is impossible for the width or height of the created window to be 0. Therefore, if the widget does not yet have a width or height, the *realize* method of **XmManager** forces the width or height to 1 pixel.

The window class of the created window will be **InputOutput**. The generated window will not propagate any button, key, or pointer motion events to its parent window(s).

The default windows for managers and primitives use bit gravity differently. Default manager windows set bit gravity to **NorthWest**, while default primitive windows take the default bit gravity, **ForgetGravity**.

The *CDE 2.1/Motif 2.1—Style Guide and Glossary* does not demand that all windows conform to a certain standard. For that matter, the *CDE 2.1/Motif 2.1—Style Guide and Glossary* does not even insist that all windows be rectangular. Therefore, you are free to create your own *realize* method.

4.1.10 The actions Field

Specify the name of the array that holds the *actions* methods defined by your widget. (See Chapter 7 for details on *actions*.)

4.1.11 The num_actions Field

The **num_actions** field holds the number of actions defined by the array in the *actions* field. If your widget does not provide an *actions* array, then you should specify 0 for the **num_actions** field. If your widget does provide an *actions* array, then you should use the **XtNumber** macro to count these actions. For example, the *ExmCommandButton* widget does provide an *actions* array, so its **num_actions** field looks as follows:

```
/* num_actions */                XtNumber(actions),
```

4.1.12 The resources Field

If your widget defines new resources, specify the name of the *resources* array. If your widget does not define new resources, specify *NULL*.

Your widget will automatically inherit the resources of its superclasses. (See Chapter 6 for more details on resources.)

4.1.13 The num_resources Field

The **num_resources** field holds the number of resources defined by the array in the *resources* field. If your widget does not provide a *resources* array, then you should specify 0 for the **num_resources** field. If your widget does provide a *resources* array, then your widget should use the **XtNumber** macro to count these resources. For example, the *ExmGrid* widget does provide a *resources* array, so its **num_resources** field looks as follows:

```
/* num_resources */                XtNumber(resources),
```

4.1.14 The xrm_class Field

Your widget must always specify **NULLQUARK** as the value of the **xrm_class** field.

4.1.15 The compress_motion Field

The **compress_motion** field requires a **Boolean** value. Setting this value to **True** eliminates certain redundant pointer motion events from the event queue. Setting this value to **False** imposes a slight performance penalty but produces more accurate pointer tracking. (See documentation on the Intrinsics for more details.)

Every standard Motif manager widget sets the **compress_motion** field to **True**.

4.1.16 The compress_exposure Field

The **compress_exposure** field requires a constant value. This value tells the Intrinsics how to respond to multiple exposure events. (See documentation on the Intrinsics for more complete details.)

Motif makes no recommendations on the value of the **compress_exposure** field. The **XmManager** widget sets this field to **XtExposeCompressMaximal**.

Managers that may have gadget children should not use the **XtExposeNoRegion** flag.

4.1.17 The compress_enterleave Field

The **compress_enterleave** field requires a **Boolean** value. Setting this value to **True** means that the Intrinsics will ignore unimportant enter event/leave event pairs. (See documentation on the Intrinsics for more complete details.)

Motif makes no recommendations on the value of the **compress_enterleave** field. However, most standard Motif manager widgets, including the **XmManager** widget, set this field to True.

4.1.18 The visible_interest Field

The **visible_interest** field requires a **Boolean** value. This value provides hints to the Intrinsics on the redisplay of partially exposed widgets. Motif does not layer any extra meaning on top of the standard Intrinsics use of this field.

Motif makes no recommendations on the value of this field. However, every standard Motif widget sets this field to **False**.

4.1.19 The destroy Field (Chained)

The Intrinsics automatically call the *destroy* method when the widget is explicitly destroyed. You must specify one of the following:

- The name of your *destroy* method
- *NULL*, to indicate the absence of a *destroy* method

The *destroy* field is chained in subclass-to-superclass order; therefore, the *destroy* method of **XmManager** will be called after any *destroy* method that your own widget supplies. The *destroy* method of **XmManager** deallocates the following four graphics contexts:

- **manager.top_shadow_GC**
- **manager.bottom_shadow_GC**
- **manager.highlight_GC**
- **manager.background_GC**

In addition, the *destroy* method of **XmManager** deallocates all the widget traversal information held by this widget.

If your widget provides a *destroy* method, make sure it deallocates all strings, fonts, GCs, and so forth, allocated by your widget.

4.1.20 The resize Field

The Intrinsics automatically call the method specified in the *resize* field in response to a *resize* event. The **XmManager** widget does not provide a *resize* method, so your widget will not be able to inherit a *resize* method from **XmManager**. (See Chapter 12 for more information on geometry management.)

4.1.21 The expose (Redisplay) Field

The Intrinsics automatically invoke the *expose* method upon receiving an exposure event. The interpretation of exposure events can be altered by the values of the **compress_exposure** and **visible_interest** fields. (See documentation on the Intrinsics for complete details.)

Your widget should set the *expose* field to one of the following:

- **XtInheritExpose**, to indicate that you are inheriting the *expose* method of your superclass. However, the **XmManager** widget does not provide an *expose* method, so your widget cannot inherit from it.
- The name of your widget's *expose* method. If you do provide an *expose* method, you should name it **Redisplay**.

All standard Motif manager widgets except **XmManager** provide their own *expose* method. Motif requires that your **Redisplay** method (or the **Redisplay** method your widget inherits) call **XmeRedisplayGadgets** in order to pass exposure events down to gadget children. (See Chapter 17 for syntactic details on **XmeRedisplayGadgets**.)

4.1.22 The set_values Field (Chained)

You can either specify the name of your widget's **set_values** method or you can specify *NULL*, which symbolizes the absence of a **set_values** method. Most of the widgets you write will specify a **set_values** method.

When an application calls **XtSetValues** to change any of the resource values in your widget, the Intrinsics automatically call the **set_values** method.

Your widget's **set_values** method is responsible for validating changes to resource values. Your widget can call **XmRepTypeValidValue** to validate changes to representation type resources. (See Chapter 6 for details.)

The code in a **set_values** method is usually quite similar to the code in an *initialize* method.

The **set_values** method returns a Boolean value. The Intrinsics monitor this returned value. If the returned value is True, the Intrinsics automatically invoke the widget's *expose* (**Redisplay**) method. There are many reasons why widget redisplay could benecessary. One subtle reason is that a widget GC has changed.

The **set_values** method is chained. Therefore, the Intrinsics will call the **set_values** method of **XmManager** prior to calling the **set_values** method of your own widget. The **set_values** method of **XmManager** does the following:

- Validates changes to **XmNinitialFocus**, **XmNnavigationType**, **XmNunitType**, and **XmNstringDirection**.

- Updates keyboard traversals.
- Examines the **manager.highlight_color** and **manager.highlight_pixmap** fields. If either field's value has changed, **set_values** destroys the existing **manager.highlight_GC** and then creates a new one from the updated values. If either field's value has changed, **set_values** sets its return value to **True**.
- Examines the **manager.top_shadow_color** and **manager.top_shadow_pixmap** fields. If either field's value has changed, **set_values** destroys the existing **manager.top_shadow_GC** and then creates a new one from the updated values. If either field's value has changed, **set_values** sets its return value to **True**.
- Examines the **manager.bottom_shadow_color** and **manager.bottom_shadow_pixmap** fields. If either field's value has changed, **set_values** destroys the existing **manager.bottom_shadow_GC** and then creates a new one from the updated values. If either field's value has changed, **set_values** sets its return value to **True**.
- Examines the values of the **manager.background_pixel** or **manager.background_pixmap** fields. If either field's value has changed, **set_values** destroys the existing **manager.background_GC** and then creates a new one from the updated values. If either field's value has changed, **set_values** sets its return value to **True**.
- Checks the values of the GCs of any gadget children of the widget.
- Determines if any visual changes have occurred in your manager widget. If any have, the **set_values** method calls the *redraw* trait method of any child widget that has the *XmQTcareParentVisual* trait installed. If your manager widget needs to be redrawn as a result of this call, **set_values** sets its return value to **True**.

4.1.23 The set_values_hook Field (Chained)

The **set_values_hook** field is obsolete. Motif ignores its value; set it to *NULL*.

4.1.24 The set_values_almost Field

The Intrinsics call the method named in the **set_values_almost** field when certain geometry requests cannot be fulfilled by the geometry manager. To inherit the

set_values_almost method of your superclass, specify **XtInheritSetValuesAlmost**. The **XmManager** widget itself sets the **set_values_almost** field to **XtInheritSetValuesAlmost**, so it inherits this field from the Intrinsics widgets.

Motif makes no requirements for this method. However, all the standard Motif manager widgets set this field to **XtInheritSetValuesAlmost**. (See Chapter 12 for details on Motif geometry management.)

4.1.25 The get_values_hook Field (Chained)

The **get_values_hook** field holds the name of a method responsible for returning resource values from any subparts of your widget. If your widget has no need to do this, it should set the **get_values_hook** field to *NULL*.

The **get_values_hook** field is chained; therefore, the Intrinsics will call the **get_values_hook** method of **XmManager** prior to the **get_values_hook** method of your widget. The **get_values_hook** method of **XmManager** implements synthetic resources on manager widgets.

4.1.26 The accept_focus Field

Motif ignores the value of the **accept_focus** field; set it to *NULL*.

4.1.27 The version Field

The *version* field holds an enumerated constant pertaining to the compatibility of the Intrinsics themselves. The *version* field does not express any information regarding Motif compatibility, nor does Motif seek to interpret the *version* field in any way.

Most widgets set this field to **XtVersion**. However, if you do not want your widget to check binary compatibility with future releases of the Intrinsics, set this field to **XtVersionDontCheck**. (See Chapter 15 for details on binary compatibility.)

4.1.28 The callback_private Field

Motif ignores the value of the **callback_private** field; set it to *NULL*.

4.1.29 The tm_table Field

You should specify one of the following three possible values for the **tm_table** field:

- **XtInheritTranslations**, to inherit the **tm_table** string of your widget's immediate superclass.
- The name of your own **tm_table** string to provide **tm_table** translations specific to your widget.
- *NULL*, to indicate the absence of any **tm_table** string translations.

If your **tm_table** field is set to **XtInheritTranslations** and your widget's immediate superclass is **XmManager**, then your widget will inherit the following **tm_table** translations:

```
<BtnMotion>: ManagerGadgetButtonMotion()\n\
c<Btn1Down>: ManagerGadgetTraverseCurrent()\n\
~c<Btn1Down>: ManagerGadgetArm()\n\
~c<Btn1Down>,~c<Btn1Up>: ManagerGadgetActivate()\n\
~c<Btn1Up>: ManagerGadgetActivate()\n\
~c<Btn1Down>(2+): ManagerGadgetMultiArm()\n\
~c<Btn1Up>(2+): ManagerGadgetMultiActivate()\n\
<Btn2Down>: ManagerGadgetDrag()\n\
:<Key>osfActivate: ManagerParentActivate()\n\
:<Key>osfCancel: ManagerParentCancel()\n\
:<Key>osfSelect: ManagerGadgetSelect()\n\
:<Key>osfHelp: ManagerGadgetHelp()\n\
~s ~m ~a <Key>Return: ManagerParentActivate()\n\
~s ~m ~a <Key>space: ManagerGadgetSelect()\n\
<Key>: ManagerGadgetKeyInput()";
```

Suppose you provide your own **tm_table** string rather than inherit the **tm_table** string of **XmManager**. If you want your manager widget to handle the default button events correctly, then your **tm_table** string should be a superset of the activation translations of **XmManager**. That is, your **tm_table** string should contain the following:

```
:<Key>osfActivate: ManagerParentActivate()\n\
:<Key>osfCancel: ManagerParentCancel()\n\
~s ~m ~a <Key>Return: ManagerParentActivate()\n\
```

If you want your manager widget to support gadgets properly, then the **tm_table** string should specify all of the translations used by **XmManager**.

(See Chapter 7 for more details on translations.)

4.1.30 The query_geometry Field

When your manager is itself a child of another manager widget, the parent manager widget will call your widget’s **query_geometry** method to request your widget’s geometry preferences.

Your widget must specify one of the following for the **query_geometry** field:

- The name of your widget’s **query_geometry** method.
- **XtInheritQueryGeometry** to inherit the **query_geometry** method of your widget’s immediate superclass.

  The **XmManager** widget does not provide a **query_geometry** method. Therefore, your widget cannot inherit from it.
- *NULL*, to indicate the absence of a **query_geometry** method. Motif does not recommend this.

(See Chapter 12 for details on Motif geometry management.)

4.1.31 The display_accelerator Field

Motif ignores the value of the **display_accelerator** field; set it to *NULL*.

4.1.32 The extension Field

Motif strongly recommends that you set the value of the *extension* field to *NULL*. By doing so, Motif will install the correct extension record for you at runtime.

4.2 The CompositeClassPart Structure

The second part of the class record is the **CompositeClassPart** structure. For example, following is the **CompositeClassPart** structure of the *ExmGrid* widget:

```
{                  /* composite_class */
  /* geometry_manager */          GeometryManager,
  /* change_managed */            ChangeManaged,
  /* insert_child */              XtInheritInsertChild,
  /* delete_child */              XtInheritDeleteChild,
  /* extension */                 NULL,
},
```

The following subsections detail each field from a Motif widget writer's perspective.

4.2.1 The geometry_manager Field

The **geometry_manager** method handles geometry requests from children of your manager widget. Your widget must specify one of the following for the **geometry_manager** field:

- The name of your widget's **geometry_manager** method.
- **XtInheritGeometryManager** to inherit the **geometry_manager** method of your widget's immediate superclass. The **XmManager** widget does not provide a **geometry_manager** method, so your widget cannot inherit this method from **XmManager**.
- *NULL*, to indicate the absence of a **geometry_manager** method. Motif does not recommend this.

Most standard Motif manager widgets provide their own **geometry_manager** method. Motif makes no requirements of the **geometry_manager** method beyond those required by the Intrinsics.

The *ExmGrid* demonstration widget illustrates how to write one kind of **geometry_manager** method. You can find the source code for *ExmGrid* in **demos/widgets/Exm/lib/Grid.c**. (See Chapter 12 for an overview of geometry management in Motif.)

4.2.2 The change_managed Field

The Intrinsics call the **change_managed** method whenever:

- A managed child becomes unmanaged.
- An unmanaged child becomes managed.

Your widget must specify one of the following for the **change_managed** field:

- The name of your widget's **change_managed** method.
- **XtInheritChangeManaged**, to inherit the **change_managed** method of your widget's immediate superclass. The **XmManager** widget does not provide a **change_managed** method, so your widget cannot inherit this method from **XmManager**.
- *NULL*, to indicate the absence of a **change_managed** method. Motif does not recommend this.

Motif requires that your **change_managed** method call the **XmeNavigChangeManaged** function. If your **change_managed** method does not, Motif's keyboard traversal will not work correctly.

The *ExmGrid* demonstration widget illustrates how to write one kind of **change_managed** method. You can find the source code for *ExmGrid* in **demos/widgets/Exm/lib/Grid.c**. (See Chapter 12 for an overview of geometry management in Motif.)

4.2.3 The insert_child Method

The Intrinsics call the **insert_child** method to report the creation of a new child under control of the manager widget.

You must set the **insert_child** method to one of the following:

- **XtInheritInsertChild**, to inherit the **insert_child** method of your superclass. **XmManager** provides an **insert_child** method that your widget can inherit.
- The name of your widget's **insert_child** method.

The **insert_child** method of **XmManager** simply calls the **insert_child** method of **Composite**. (See documentation on the Intrinsics for more information on **Composite**.)

If you do write your own **insert_child** method, Motif recommends that it call the **insert_child** method of **XmManager**. For example, suppose you are writing a manager widget that can only support one child. The following **insert_child** method rejects any attempt to add a second child. If the added child is the first child, then **MyManagerInsertChild** calls the **insert_child** method of **XmManager**.

```
MyManagerInsertChild(Widget child)
{
  ExmMyManagerWidget cb = (ExmMyManagerWidget) XtParent(child);
  if (cb->composite.num_children > 1)
    XmeWarning((Widget)cb, "MyManager cannot manage more than one child");
  else
/* Call the insert_child method of XmManager to update child info. */
    (*((XmManagerWidgetClass) xmManagerWidgetClass)
     ->composite_class.insert_child) (child);
}
```

4.2.4 The delete_child Method

The Intrinsics call the **delete_child** method to report the destruction of a child under control of the manager widget. The widget ID of the deleted child is the sole argument passed to the **delete_child** method.

You must set the **delete_child** method to one of the following:

- **XtInheritDeleteChild**, to inherit the **delete_child** method of your superclass. **XmManager** provides a **delete_child** method that your widget can inherit.
- The name of your widget's **delete_child** method.

The **delete_child** method of **XmManager** does the following:

- Determines if the deleted child is the one stored in the **manager.selected_gadget** field. If it is, **delete_child** sets **manager.selected_gadget** to *NULL*. (See later in this chapter for more information on **manager.selected_gadget**.)
- Determines if the deleted child is the one indicated by the **XmNinitialFocus** resource. If it is, **delete_child** sets **XmNinitialFocus** to *NULL*.
- Determines if the deleted child is the one stored in the **manager.active_child** field. If it is, **delete_child** sets **manager.active_child** to *NULL*. (See later in this chapter for more information on **manager.active_child**.)
- Finds the widget that is the tab group manager for the deleted child. If the tab group manager is not your manager widget, **delete_child** examines the tab group manager's **manager.active_child** field. If this field holds the widget ID of the deleted child, **delete_child** sets the tab group manager's **manager.active_child** field to *NULL*.
- Calls the **delete_child** method of **Composite**.

4.2.5 The extension Field

The **Composite** class supports an extension record. The only nontrivial field in this extension record is called **accepts_objects**. If you specify the *extension* field as *NULL*, an extension record is automatically installed and **accepts_objects** is set to True. All but one of the standard Motif manager widgets (**XmRowColumn**) set the *extension* field to *NULL*.

If your manager widget requires that **accepts_objects** be False, then you must create an extension record with the **accepts_objects** field set to False. Furthermore, you must set the *extension* field to the name of your own composite class extension record.

4.3 The ConstraintClassPart Structure

The third part of the class record is the **ConstraintClassPart** structure. For example, following is the**ConstraintClassPart** structure of the *ExmGrid* widget:

```
{                /* constraint_class */
  /* resources */                 constraints,
  /* num_resources */             XtNumber(constraints),
  /* constraint_size */           sizeof(ExmGridConstraintRec),
  /* initialize */                NULL,
  /* destroy */                   NULL,
  /* set_values */                ConstraintSetValues,
  /* extension */                 NULL,
},
```

The following subsections detail each of these fields.

4.3.1 The resources (constraints) Field

The *resources* field holds the name of the constraints array. This array has the same syntax as a *resources* array. For example, following is the constraints array of the *ExmGrid* widget:

```
static XtResource constraints[] =
{
    {
        ExmNgridMarginWidthWithinCell,
        ExmCGridMarginWidthWithinCell,
        XmRHorizontalDimension,
        sizeof (Dimension),
        XtOffsetOf( ExmGridConstraintRec,
                   grid.grid_margin_width_within_cell),
        XmRImmediate,
        (XtPointer) 0
    },
    {
        ExmNgridMarginHeightWithinCell,
        ExmCGridMarginHeightWithinCell,
        XmRVerticalDimension,
        sizeof (Dimension),
        XtOffsetOf( ExmGridConstraintRec,
                   grid.grid_margin_height_within_cell),
        XmRImmediate,
        (XtPointer) 0
    },
};
```

Motif also supports synthetic constraints, which are explained later in this section.

4.3.2 The num_resources Field

The **num_resources** field holds the number of constraints defined by the array in the *resources* field of the **ConstraintClassPart** structure. If your widget does not provide a *constraints* array, then you should specify 0 for the **num_resources** field. If your widget does provide a *constraints* array, then your widget should use the **XtNumber** macro to count these resources. For example, the *ExmSimple* widget does provide a *resources* array, so its **num_resources** field looks as follows:

```
/* num_resources */                XtNumber(constraints),
```

4.3.3 The constraint_size Field

The **constraint_size** field holds the size of the full constraint structure. For example, the full constraint structure of the *ExmGrid* demonstration widget is named *ExmGridConstraintRec*; therefore, the Core class part of *ExmSimple* defines the **constraint_size** field as follows:

```
/* constraint_size */                sizeof(ExmGridConstraintRec);
```

4.3.4 The Constraint initialize Field (Chained)

To satisfy the Constraint *initialize* field, you can either specify the name of your widget’s Constraint *initialize* method or you can specify *NULL*, to indicate the absence of a Constraint *initialize* method.

The Constraint *initialize* field is chained; therefore, the Intrinsics call the Constraint *initialize* method of **XmManager** before calling the Constraint *initialize* method in your own widget. The Constraint *initialize* method of **XmManager** examines each child of your widget as it is created. The response taken by this method depends on whether the created child is a widget or is a gadget.

If the created child is a gadget, the Constraint *initialize* method of **XmManager** examines the selected events of its gadgets. If the selected events are any of the following, the Constraint *initialize* method installs an event handler:

- Motion
- Enter
- Leave

The event handler dispatches events to the appropriate gadget for analysis.

If the created child is a widget, the Constraint *initialize* method installs accelerators stored in the accelerator widget.

4.3.5 The Constraint destroy Field (Chained)

The Constraint *destroy* field holds the name of a method that is invoked when a widget is destroyed. You must specify one of the following:

- The name of your widget's Constraint *destroy* method. Your widget should provide a Constraint *destroy* method if your widget's Constraint *initialize* method allocated any dynamic memory. The Constraint *destroy* method of your widget is responsible for deallocating this dynamic memory.
- *NULL*, to indicate the absence of a Constraint *destroy* method. Your widget should probably specify *NULL* if your widget's Constraint *initialize* method did not allocate any dynamic memory.

The Constraint *destroy* method is chained in subclass-to-superclass order. Therefore, the Intrinsics will call the Constraint *destroy* method of **XmManager** after calling the Constraint *destroy* method in your own widget. The Constraint *destroy* method of **XmManager** conditionally removes the event handler that was put into place by the Constraint *initialize* method of **XmManager**. That is, the Constraint *destroy* method of

XmManager determines whether the event handler is still needed. If it is not needed, Constraint *destroy* removes it. Otherwise, Constraint *destroy* keeps it.

4.3.6 The Constraint set_values Field (Chained)

To satisfy the Constraint **set_values** field, you must specify one of the following:

- The name of your widget's Constraint **set_values** method
- *NULL*, to indicate the absence of a Constraint **set_values** method

The Constraint **set_values** field is chained; therefore, the Constraint **set_values** method of **XmManager** will be called before any Constraint **set_values** method in your own widget. The Constraint **set_values** method of **XmManager** determines whether any changes to gadget children (for instance, a change in the gadget's selected events) make it necessary to install or remove the event handler.

4.3.7 The extension Field

If you want your widget to support a **ConstraintClassPart** extension record, set the *extension* field to the name of that extension record. If you do not want this extension record, set the field to *NULL*.

The one nontrivial field in the *extension* record is **get_values_hook**. Motif makes no recommendations on this field. (See the Intrinsics documentation for details.)

4.4 The ManagerClassPart Structure

The fourth part of the class record is the **ManagerClassPart** structure. For example, following is the **ManagerClassPart** structure of the *ExmGrid* widget:

```
{                   /* manager class */
  /* translations */              XtInheritTranslations,
  /* syn_resources */             syn_resources,
  /* num_syn_resources */         XtNumber(syn_resources),
  /* syn_constraint_resources */   syn_constraints,
  /* num_syn_constraint_resources */ XtNumber(syn_constraints),
  /* parent_process */            XmInheritParentProcess,
  /* extension */                 NULL,
},
```

The following subsections explain what a Motif widget writer should know about these fields.

4.4.1 The translations Field

The *translations* field defines the keyboard traversal translations for your manager widget. For this field, you must specify one of the following:

- *NULL*, to indicate the absence of keyboard traversal translations.
- **XtInheritTranslations**, to inherit the keyboard traversal translations of the widget's superclass, which is typically **XmManager**.
- The name of your own keyboard traversal translations string.

The keyboard traversal translations for the **XmManager** widget are as follows:

```
<EnterWindow>:                         ManagerEnter()
<LeaveWindow>:                         ManagerLeave()
<FocusOut>:                       ManagerFocusOut()
<FocusIn>:                        ManagerFocusIn()
:<Key>osfBeginLine:          ManagerGadgetTraverseHome()
:<Key>osfUp:                   ManagerGadgetTraverseUp()
:<Key>osfDown:                         ManagerGadgetTraverseDown()
:<Key>osfLeft:                         ManagerGadgetTraverseLeft()
:<Key>osfRight:                     ManagerGadgetTraverseRight()
s ~m ~a <Key>Tab:              ManagerGadgetPrevTabGroup()
~m ~a <Key>Tab:                     ManagerGadgetNextTabGroup()
```

If you provide your own keyboard traversal translations string, Motif strongly recommends that the string contain all the keyboard traversal translations of **XmManager**.

4.4.2 The syn_resources Field

The **syn_resources** field holds the name of the array that defines the widget’s synthetic resources. Synthetic resources provide a mechanism for translating widget resource values between different formats. (See Chapter 6 for information on synthetic resources.)

If you do not wish to define synthetic resources for your widget, set the value of this field to *NULL*.

4.4.3 The num_syn_resources Field

The **num_syn_resources** field holds the number of synthetic resources defined by the array in the **syn_resources** field. If your widget does not provide a **syn_resources** array, then you should specify 0 for the **num_syn_resources** field. If your widget does provide a **syn_resources** array, then your widget should use the **XtNumber** macro to count these resources.

4.4.4 The syn_constraint_resources field

Synthetic constraints are to constraints what synthetic resources are to resources. In short, synthetic constraints provide a mechanism for automatic conversion of constraints between an external format and an internal format. (See Chapter 6 for details on synthetic resources.)

You must specify one of the following for the **syn_constraint_resources** field:

- The name of your widget’s synthetic constraint resources array. This array must have a base data type of **XmSyntheticResource**.
- *NULL*, to indicate the absence of any synthetic constraints.

For example, the *ExmGrid* widget defines two synthetic constraints in its synthetic constraint resources array as follows:

```
static XmSyntheticResource syn_constraints[] =
{
    {
        ExmNgridMarginWidthWithinCell, /* resource name */
        sizeof (Dimension),  /* data type of resource */
        XtOffsetOf( ExmGridConstraintRec,
                      grid.grid_margin_width_within_cell), /* offset */
        XmeFromHorizontalPixels, /* conversion routine */
        XmeToHorizontalPixels /* conversion routine */
    },
    {
        ExmNgridMarginHeightWithinCell,
        sizeof (Dimension),
        XtOffsetOf( ExmGridConstraintRec, grid.grid_margin_height_within_cell),
        XmeFromVerticalPixels,
        XmeToVerticalPixels
    },
};
```

XmeFromHorizontalPixels and **XmeToHorizontalPixels** are functions that convert a constraint resource value between pixels and real-world dimensions assuming a horizontal resolution. (See Chapter 6 for details.)

4.4.5 The num_syn_constraint_resources Field

The **num_syn_constraint_resources** field holds the number of synthetic constraints defined by the array in the **syn_constraint_resources** field. If your widget does not provide a **syn_constraint_resources** array, then you should specify 0 for the **num_syn_constraint_resources** field. If your widget does provide a **syn_constraint_resources** array, then your widget should use the **XtNumber** macro to count these constraints.

4.4.6 The parent_process Field

The **parent_process** method provides a mechanism for a child to pass an event up to its parent. Motif widgets use this mechanism to pass an osfActivate or an osfCancel event up to the first manager in the hierarchy that can process it. The osfActivate and osfCancel of all Motif primitive actions end up calling their parent's **parent_process** method.

Your manager widget must set the **parent_process** field to one of the following:

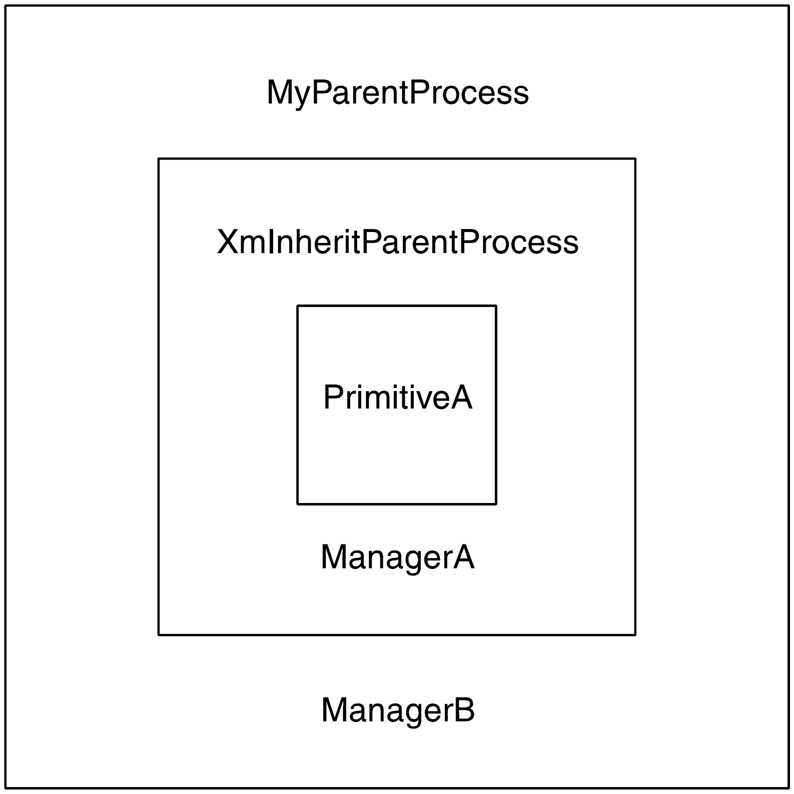
- *NULL*, to indicate the absence of any **parent_process** method. Motif does not recommend doing this.
- **XmInheritParentProcess**, to indicate that you are inheriting the **parent_process** method of your superclass. The **XmManager** widget provides a **parent_process** method that your subclass can inherit.
- The name of your widget's **parent_process** method.

The **parent_process** method of **XmManager** passes the received event up to the parent of the manager that received it. If that manager widget also inherits the **parent_process** method of **XmManager**, then the event will be passed to its parent, and so on. This trail of passed events ends when either of the following happens:

- The event reaches a manager widget that defines a **parent_process** method. This manager widget is responsible for processing the event.
- The event is passed to a parent that is not a manager widget; for example, the parent might be a Shell. In this case, the received event will be discarded.

For example, consider the hierarchy shown in Figure 4-2. Assume that **ManagerA** inherits the **parent_process** method of **XmManager**, but **ManagerB** provides its own **parent_process** method. Suppose that **PrimitiveA** receives an osfActivate event. Since **ManagerA** is the parent of **PrimitiveA**, the osfActivate event is passed up to **ManagerA**. **ManagerA** cannot process the event, so **ManagerA** passes the event up to its parent, **ManagerB**. **ManagerB** processes the osfActivate event.

Figure 4–2. A Hierarchy for Exploring the parent_process Method



On the other hand, if **ManagerB** had been a Shell rather than a manager, then the osfActivate event would have been ignored.

Many manager widgets have no need to track the osfActivate and osfCancel events of their children. For such widgets, inheriting the **parent_process** method of **XmManager** is appropriate. However, DialogBox widgets do need to know when a child has received an osfActivate or osfCancel event in order to activate the default button or the cancel button.

If you write your own **parent_process** method, it must have the following prototype:

```
Boolean ParentProcess(
        Widget                  w,
        XmParentProcessData     data)
```

The second argument to the method must be a **XmParentProcessData** structure. This structure has the following definition:

```
typedef struct XmParentProcessData {
        int             process_type;
        XEvent          *event;
        int             action;
        String          *params;
        Cardinal        *num_params;
};
```

process_type	Specifies **XmINPUT_ACTION**.
event	Points to the *XEvent* that triggered the call.
action	Specifies either **XmPARENT_ACTIVATE** for an activation event or **XmPARENT_CANCEL** for a cancel event.
params	Points to the parameter string that gets passed to the action.
num_params	Specifies the number of parameters passed in *params*.

Your **parent_process** method must return a Boolean value. Return True if the parent of your widget is a manager with a **parent_process** field set to something other than *NULL*. Otherwise, return False.

Your widget’s **parent_process** method must do one of the following tasks:

- Handle the received **XmParentProcessData** structure.
- Pass the received **XmParentProcessData** structure to the **parent_process** method of your widget’s parent (but only if your widget’s parent is a manager widget).

The following code illustrates how to do the latter task:

```
XmManagerWidgetClass manClass;

  manClass = (XmManagerWidgetClass) widget->core.widget_class;
  if (XmIsManager(widget) && manClass->manager_class.parent_process)
    return( (*manClass->manager_class.parent_process)( widget, data));
    return( FALSE) ;
```

4.4.7 The extension Field

The **XmManager** widget provides an extension record. Motif recommends that you specify the *extension* field as *NULL*. If you do, Motif will automatically supply an appropriate *extension* field for your widget. If you do not specify the *extension* field as *NULL*, you must specify the name of the manager class extension record.

4.5 The Manager Class Extension Record

The **XmManager** widget provides an **XmManagerClassExtRec** extension record. However, most standard Motif manager widgets do not define one. Nevertheless, the following is an example of one hypothetical manager widget that does:

```
static XmManagerClassExtRec MyWidgetMgrClassExtRec = {
    /* next_extension */            NULL,
    /* record_type */               NULLQUARK,
    /* version */                   XmManagerClassExtVersion,
    /* record_size */               sizeof(XmManagerClassExtRec),
    /* traversal_children */        TraversalChildren,
    /* object_at_point */           XmInheritObjectAtPointProc
};
```

The following subsections detail this extension record.

4.5.1 The next_extension Field

Your widget should always set the **next_extension** field to *NULL*.

4.5.2 The record_type Field

Your widget should always set the **record_type** field to **NULLQUARK**.

4.5.3 The version Field

Your widget must always set the *version* field to **XmManagerClassExtVersion**.

4.5.4 The record_size Field

The **record_size** field holds the size of the manager class extension record. You should set it as follows:

```
/* record_size */                sizeof(XmManagerClassExtRec),
```

4.5.5 The traversal_children Field

The **traversal_children** field holds the name of a method that defines the list of traversable children managed by your widget. Your widget must set the **traversal_children** field to one of the following:

- The name of your widget’s **traversal_children** method.
- **XmInheritTraversalChildrenProc**, to indicate that you are inheriting the **traversal_children** method of your superclass. The **XmManager** widget does not provide a **traversal_children** method, so you cannot inherit from it.
- *NULL*, to indicate the absence of a **traversal_children** method.

Most standard Motif manager widgets set this field to *NULL*. If your widget does this, then your widget will obey the same traversal rules that most other Motif manager widgets enforce. That is, Motif will view all the children in your widget’s **composite.children** list as potential candidates for traversal. However, some of these candidates may not actually be traversable. For example, a Motif traversal routine will mark a child as nontraversable if the child’s **XmNtraversalOn** resource is set to False. (See the *Motif 2.1—Programmer’s Guide* for details on traversal.)

The only reason to write your own **traversal_children** method is that you do not want all of your widget's children to be potential candidates for traversal. A **traversal_children** method must have the following prototype:

```
Boolean  TraversalChildren(
        Widget          w,
        Widget          **childList,
        Cardinal        *numChildren)
```

w	Specifies the widget ID of your widget.
childList	Returns a pointer to the list of potentially traversable children. The **traversal_children** method must dynamically allocate this list. The caller is responsible for deallocating it.
numChildren	Returns a pointer to the number of potentially traversable children returned into *childList*.

Your **traversal_children** method must return a **Boolean** value. Returning **False** means that your widget did not allocate any dynamic memory to hold the *childList*. In this case, the callers (Motif traversal routines) assume that all the children in the widget's **composite.children** list are potential candidates for traversal. Returning **True** means that your widget did allocate dynamic memory to create *childList*. Therefore, the caller is responsible for deallocating (with **XtFree**) the returned *childList*.

In order to create *childList*, your **traversal_children** method will probably start with the default child list stored in **composite.children**. Your **traversal_children** method will then

- Remove some of the children from this list
- Add some children to this list
- Keep the contents of the list the same, but rearrange their order

For example, the **XmRowColumn** widget provides a **traversal_children** method. This method begins by gathering the default child list stored in **composite.children**. The code that does this looks something like this:

```
XmRowColumnWidget  RCWid = (XmRowColumnWidget)w;
 *childList = (WidgetList) XtMalloc(sizeof(Widget) *
                              (RCWid->composite.num_children+1));
```

The method then adds the visible tear-off control widget to the beginning of the child list. (If **XmRowColumn** did not provide a **traversal_children** method, the tear-off control widget would not be part of the traversable children list.) The modified *childList* is returned to the calling Motif traversal routine, which treats it as the list of potentially traversable children.

In the preceding code, the **traversal_children** method called **XtMalloc** to allocate enough dynamic memory to store the default child list.

4.5.6 The object_at_point Field

The **object_at_point** field holds a method that returns the child most closely associated with a specified position within your manager widget. Motif calls the **object_at_point** method of your widget when an application specifies your widget as the first argument to the **XmObjectAtPoint** function. Your widget must set the **object_at_point** field to one of the following:

- **XmInheritObjectAtPointProc**, to indicate that you are inheriting the **object_at_point** method of your superclass. The **XmManager** widget provides an **object_at_point** method, and your manager widget can inherit this method.
- The name of your **object_at_point** method.

All standard Motif manager widgets except **XmContainer** inherit the **object_at_point** method of **XmManager**. The **object_at_point** method of **XmManager** uses the following rules to determine which child to return:

- If one child intersects the specified coordinate pair, that child's widget ID is returned.
- If more than one child intersects the specified coordinate pair, the visible child's widget ID is returned.
- If no child intersects the specified coordinate pair, *NULL* is returned.

If you do write your own **object_at_point** method, it must have the same prototype as the **XmObjectAtPoint** routine, namely:

```
Widget ObjectAtPoint(
        Widget          w,
        Position        x,
        Position        y)
```

widget Specifies your manager widget.

x Specifies the *x*-coordinate about which the caller is seeking child information. The *x*-coordinate is specified in pixels, relative to the left side of your *manager*.

y Specifies the *y*-coordinate about which the caller is seeking child information. The *y*-coordinate is specified in pixels, relative to the top side of *manager*.

An **object_at_point** method must return the child most closely associated with the specified *x*,*y* coordinate pair. Each **object_at_point** method is free to define "most closely associated" as it pleases. For example, your own **object_at_point** method might return the child closest to *x*,*y*, while another **object_at_point** method might return the child that is closest to just the *x*-coordinate.

4.6 The Instance Data Members of XmManager

The **XmManager** widget maintains the values of its resources inside the *manager* structure. Each resource value is stored in a different field of *manager*. For example, the value of the **XmNshadowThickness** resource is stored in the **manager.shadow_thickness** field. Your manager widget can access any of these fields; for example:

```
/* Add 2 pixels to the current shadow thickness */
 my_manager->my_manager.shadow_thickness += 2;
```

In addition to all the resource values, the *manager* structure also holds several fields not tied to resources. Widget writers should become familiar with the fields shown in the following subsections.

4.6.1 The manager.accelerator_widget Field

The **accelerator_widget** field is declared as follows:

```
Widget        accelerator_widget;
```

The **manager.accelerator_widget** field holds the name of a widget whose accelerators will be automatically propagated to all new children of this manager.

Your own manager widget may modify this field.

Suppose your own manager widget contains accelerators to be installed on all its descendants. In this case, your manager’s *initialize* method should set the **manager.accelerator_widget** field to the widget ID of your manager widget. Motif will install these accelerators in all the descendants of your manager widget. Furthermore, Motif will copy the value of the **manager.accelerator_widget** field in your widget to all manager children of your widget.

4.6.2 The manager.active_child Field

The **active_child** field is declared as follows:

```
Widget    active_child;
```

The **manager.active_child** field holds the widget ID of the gadget that has keyboard traversal focus. A value of *NULL* indicates that none of this manager’s gadgets currently have keyboard traversal focus.

Your own manager widget should not modify the value of this field.

4.6.3 The manager.background_GC Field

The **background_GC** field is declared as follows:

```
GC        background_GC;
```

XmManager creates the starting GC, which is stored in the **manager.background_GC** field. Your widget cannot modify the fields in this GC; however, your widget can deallocate this GC and create a new GC more to your liking. The values in this GC are the default values generated by **XtGetGC**, except those noted in the following list:

- The foreground is set to the value of the *XmNbackgroundPixel* resource of **Core**.

- The background is set to the value of the **XmNforeground** resource of **XmManager**.

If the widget user specifies a valid pixmap for the **XmNbackgroundPixmap** resource of **Core**, then

- The **tile** field is set to the value of the pixmap named by the **XmNbackgroundPixmap** resource.
- The **fill_style** field is set to **FillTiled**.

On the other hand, a widget user might not specify a value for the **XmNbackgroundPixmap** resource, or might specify a bad value for the resource. In either case, the values of **tile** and **fill_style** are the defaults generated by **XtGetGC**.

4.6.4 The manager.bottom_shadow_GC Field

The **bottom_shadow_GC** field is declared as follows:

```
GC        bottom_shadow_GC;
```

XmManager creates the starting GC stored in the **manager.bottom_shadow_GC** field. Your widget cannot modify the fields in this GC; however, your widget can deallocate this GC and create a new GC more to your liking. The values in this GC are the default values generated by **XtGetGC**, except those noted in the following list:

- The foreground is set to the value of the **XmNbottomShadowColor** resource of **XmManager**.
- The background is set to the value of the **XmNforeground** resource of **XmManager**.

If the widget user specifies a valid pixmap for the **XmNbottomShadowPixmap** resource of **XmManager**, then

- The **tile** field is set to the value of the pixmap named by the **XmNbottomShadowPixmap** resource.
- The **fill_style** field is set to **FillTiled**.

On the other hand, a widget user might not specify a value for the **XmNbottomShadowPixmap** resource, or might specify a bad value for the resource. In either case, the values of **tile** and **fill_style** are the defaults generated by **XtGetGC**.

4.6.5 The manager.eligible_for_multi_button_event Field

The **eligible_for_multi_button_event** field is declared as follows:

```
XmGadget  eligible_for_multi_button_event;
```

Motif uses this field to implement multiclick events on gadgets.

Your widget should not alter the value in this field.

4.6.6 The manager.event_handler_added Field

The **event_handler_added** field is declared as follows:

```
Boolean   event_handler_added;
```

The **event_handler_added** field is a flag whose initial value is False. The *initialize* method in the **ConstraintClassPart** of **XmManager** sets this flag to True when it installs a gadget event handler.

Your own manager widget should not modify the value of this field.

4.6.7 The manager.has_focus Field

The **has_focus** field is declared as follows:

```
Boolean   has_focus;
```

XmManager sets this field to **True** when the manager widget has focus and sets this field to **False** when the manager widget does not. Note that, if a child widget has focus, then **XmManager** sets this field to **False**. However, if a child gadget has focus, then **XmManager** sets this field to **True**.

Your own manager widget should not modify the value of this field.

4.6.8 The manager.highlight_GC Field

The **highlight_GC** field is declared as follows:

```
GC        highlight_GC;
```

XmManager creates the starting GC, which is stored in the **manager.highlight_GC** field. Your widget can not modify the fields in this GC; however, your widget can deallocate this GC and create a new GC more to your liking.

The values in this GC are the default values generated by **XtGetGC**, except those noted in the following list:

- The foreground is set to the value of the **XmNhighlightColor** resource of **XmManager**.
- The background is set to the value of the **XmNbackground** resource of **Core**.

If the widget user specifies a valid pixmap for the **XmNhighlightPixmap** resource of **XmManager**, then

- The **tile** field is set to the value of the pixmap named by the **XmNhighlightPixmap** resource.
- The **fill_style** field is set to **FillTiled**.

On the other hand, a widget user might not specify a value for the **XmNhighlightPixmap** resource, or might specify a bad value for the resource. In either case, the values of **tile** and **fill_style** are the defaults generated by **XtGetGC**.

4.6.9 The manager.highlighted_widget Field

The **highlighted_widget** field is declared as follows:

```
Widget   highlighted_widget;
```

The **highlight_widget** fields holds the name of the gadget currently pointed to by the pointer (typically, a mouse). If the pointer does not point to a gadget, then the value of this field will be *NULL*.

Your own manager widget should not modify the value of this field.

4.6.10 The manager.keyboard_list Field

The **keyboard_list** field is declared as follows:

```
XmKeyboardData *keyboard_list;
```

Motif no longer uses this field.

4.6.11 The manager.num_keyboard_entries Field

The **num_keyboard_entries** field is declared as follows:

```
short     num_keyboard_entries;
```

Motif does not currently use this field.

4.6.12 The manager.selected_gadget Field

The **selected_gadget** field is declared as follows:

```
XmGadget  selected_gadget;
```

Motif initializes the **selected_gadget** field to *NULL*. When a user selects a gadget (typically, by pressing a mouse button), Motif arms the gadget and sets **selected_gadget** to the ID of the selected gadget. When the user releases the mouse button, Motif disarms the gadget and sets **selected_gadget** to *NULL*.

Your widget should not alter the value in this field.

4.6.13 The manager.size_keyboard_list Field

The **size_keyboard_list** field is declared as follows:

```
short     size_keyboard_list;
```

Motif does not currently use this field.

4.6.14 The manager.top_shadow_GC Field

The **top_shadow_GC** field is declared as follows:

```
GC        top_shadow_GC;
```

XmManager creates the starting GC stored in the **top_shadow_GC** field. Your widget can not modify the fields in this GC; however, your widget can deallocate this GC and create a new GC more to your liking.

The values in this GC are the default values generated by **XtGetGC**, except those noted in the following list:

- The foreground is set to the value of the **XmNtopShadowColor** resource of **XmManager**.
- The background is set to the value of the **XmNforeground** resource of **XmManager**.

If the widget user specifies a valid pixmap for the **XmNtopShadowPixmap** resource of **XmManager**, then

- The **tile** field is set to the value of the pixmap named by the **XmNtopShadowPixmap** resource.
- The **fill_style** field is set to **FillTiled**.

On the other hand, a widget user might not specify a value for the **XmNtopShadowPixmap** resource, or might specify a bad value for the resource. In either case, the values of **tile** and **fill_style** are the defaults generated by **XtGetGC**.

Chapter 5

Traits

This chapter explains traits through text and examples. You should also see Chapter 18, which contains reference pages for all the traits.

5.1 What Is a Trait?

Motif provides widget writers with about a dozen traits. A widget writer can install any number of these traits on a widget. A trait is named by an *XrmQuark* that symbolizes a specific widget capability. If a widget holds a particular trait, then that widget is announcing a specific capability to other widgets. Conversely, if a widget does not hold a particular trait, than that widget is implicitly announcing to other widgets that it is incapable of providing a particular service.

For example, consider the *XmQTaccessTextual* trait. A widget holding this trait is announcing to other widgets that it is capable of displaying one primary text parcel. Many standard Motif widgets, including **XmText** and **XmLabel**, hold this trait. If you are writing a widget that can display one primary text parcel, then it too should hold this trait.

Table 5-1 summarizes the standard Motif traits.

Table 5–1. Standard Motif Traits

Trait Name	**A Widget Holding This Trait Can:**
XmQTaccessTextual	Display one primary text parcel
XmQTactivatable	Become a command button in a DialogBox
XmQTcareParentVisual	Borrow its parent's visual information
XmQTcontainer	Manage one or more *XmQTcontainerItem* children
XmQTcontainerItem	Become a child of an *XmQTcontainer* parent
XmQTdialogShellSavvy	Become a child of **XmDialogShell**
XmQTjoinSide	Attach itself to one side of a suitable parent
XmQTmenuSavvy	Become a menu child
XmQTmenuSystem	Manage a menu system
XmQTnavigator	Act as a navigator to a scrollable widget
XmQTscrollFrame	Handle one or more navigator widgets
XmQTspecifyRenderTable	Supply default render tables
XmQTtakesDefault	Change its appearance to show that it is the default button
XmQTtransfer	Transfer data to other widgets and/or receive data from other widgets

5.2 Why Use Traits?

The trait mechanism provides an easy way for two widgets to communicate with each other. Frequently, the two communicating widgets will be a parent and its child; however, unrelated widgets can also communicate through traits. This communication takes two forms:

- Widgets can ask each other about specific capabilities.
- Widgets can access each other's trait methods. Trait methods are defined in a trait structure variable rather than in a class record. (Trait structure variables are described later in this chapter.)

For example, consider a Motif application that needs to create a menu. To create the menu, the Motif application will need at least one **XmRowColumn** manager and some appropriate menu children. The **XmRowColumn** widget needs some way to ask each of its children if they are suitable menu children. The *XmQTmenuSavvy* trait provides a way. A child widget holding the *XmQTmenuSavvy* trait is announcing to a menu parent (**XmRowColumn**) that it is capable of becoming a menu child. Therefore, the menu parent (**XmRowColumn**) can ask each child if it holds this trait. If a child does not hold this trait, **XmRowColumn** can issue a warning that the child is not an acceptable menu child.

The *XmQTmenuSavvy* trait provides several trait methods. One of these trait methods, **disableCallback**, enables or disables the activate callback method associated with a menu child. Without a trait mechanism, there would be no way for the **XmRowColumn** widget to call **disableCallback**. After all, **XmRowColumn** shares no common ancestry with any primitive widgets. However, since **disableCallback** is a trait method, **XmRowColumn** can call **disableCallback**.

5.3 Installing a Trait

You must perform the following steps in order to install a trait on a widget:

1. Include the appropriate header file for the trait you are using.
2. Declare the trait structure variable.
3. Call the **XmeTraitSet** function.
4. Write the code for any trait methods named in the trait structure variable.

The coding for all four steps is to be added to the widget source code file. That is, you do not need to make any changes to the widget's header files.

The following subsections examine these four activities by focusing on how the *XmQTaccessTextual* trait is installed on the **ExmString** widget.

5.3.1 Step 1: Include the Appropriate Header Files

If you plan to install a trait in your widget, then your widget source code file must include the appropriate trait header files. Every widget that installs one or more traits must include the general-purpose trait header file **TraitP.h**. In addition, depending on which traits are being installed, the widget source code file must also include the appropriate trait-specific header file documented in the reference pages of Chapter 19.

For example, the widget source code file (**String.c**) for the **ExmString** widget contains the following three declarations:

```
#include <Xm/TraitP.h>
#include <Xm/AccTextT.h>
#include <Xm/SpecRenderT.h>
```

The **Xm/AccTextT.h** header file is included because *ExmString* installs the *XmQTaccessTextual* trait. In addition, the **Xm/SpecRenderT.h** header file is included because **ExmString** also uses the *XmQTspecifyRenderTable* trait.

5.3.2 Step 2: Declare the Trait Structure Variable

You must declare a trait structure variable for every trait that you plan to install. You should place this declaration in the source code file after the class record declaration but before the first widget method.

The data type of a trait structure variable must be the trait structure associated with the trait. (See Chapter 19 to find the relevant trait structure for your trait.) For example, the trait structure associated with the *XmQTaccessTextual* trait is **XmAccessTextualTraitRec**.

Motif recommends giving the trait structure variable a name that consists of the widget name followed by an acronym that symbolizes the trait. For example, *ExmString* uses the variable name **StringATT** to identify its *XmQTaccessTextual* trait structure variable.

All trait structure variable declarations should be prefixed with **static XmConst**. (See Chapter 2 for details about **XmConst**.)

For example, the **ExmString** widget declares a trait structure variable for the *XmQTaccessTextual* trait as follows:

```
static XmConst XmAccessTextualTraitRec StringATT = {
  0,                        /* version */
  StringGetValue,           /* trait method */
  StringSetValue,           /* trait method */
  StringPreferredFormat,    /* trait method */
};
```

The preceding declaration declares a trait structure variable named **StringATT**. This variable has data type **XmAccessTextualTraitRec**. This data type is defined in the **AccTextT.h** trait header file.

The first field of all trait structure variables is the *version* field. You must always specify an integer value for this field. At Motif Release 2.0, none of the standard Motif traits make any attempt to interpret the value in the version field. However, at Motif Release 2.0, you must set the *version* field to **0** since future releases of Motif may use the *version* field.

The remaining fields following the *version* field specify the names of the trait methods. By convention, the trait method names in your trait structure variables should correspond to the trait method names documented in the reference pages of Chapter 19. However, your trait method names should eliminate underscores and should capitalize the first letter of every word. For example, we named the second trait method **StringSetValue** because the trait method name **string_set_value** is documented in the *XmQTaccessTextual* trait reference page of Chapter 19.

Some traits allow you to omit certain trait methods. (See the individual reference pages of Chapter 19 for details.) To tell Motif that you are omitting a certain trait method, simply mark the appropriate field in the trait structure record as being *NULL*. For example, if we did not provide a **StringSetValue** method in the **ExmSimple** widget, the preceding declaration would have looked like this:

```
static XmConst XmAccessTextualTraitRec StringATT = {m
0,                          /* version */m
StringGetValue,             /* trait method */m
NULL,                       /* not providing this trait method */m
StringPreferredFormat,      /* trait method */
};
```

5.3.3 Step 3: Call XmeTraitSet

Call the **XmeTraitSet** function to install a trait on a widget class. Typically, your widget calls **XmeTraitSet** from the **class_part_initialize** method. By so doing, the trait is installed not only on the current widget, but also on all of its subclasses. If you want the trait installed on the current widget class only (and not its subclasses), then you should call **XmeTraitSet** from the **class_initialize** method instead of the **class_part_initialize** method.

For example, **ExmString** calls **XmeTraitSet** to install the *XmQTaccessTextual* trait as follows:

```
XmeTraitSet((XtPointer) widgetclass, XmQTaccessTextual,
           (XtPointer) &StringATT);
```

The third argument to **XmeTraitSet** holds the address of the trait structure variable created in Step 2.

Because the call to **XmeTraitSet** took place in the **class_part_initialize** method, Motif installs this trait on *ExmString* and on all its subclasses.

In some cases, you may not want the widget you are writing to inherit some of the installed traits of its superclasses. For these situations, you can prevent trait inheritance by specifying *NULL* as the third argument to **XmeTraitSet**. For example, suppose you are writing a subclass of **ExmString**, named *ExmStringSubclass*. If you do not want *ExmStringSubclass* to install *ExmQTaccessTextualTrait*, the **class_part_initialize** method of **ExmStringSubclass** should call **XmeTraitSet** as follows:

```
XmeTraitSet((XtPointer) widgetclass, XmQTaccessTextual, NULL);
```

5.3.4 Step 4: Writing Trait Methods

A trait method is a function that is callable by another widget, even if that other widget is not a subclass of the current widget. Typically, the caller is either the parent widget or the child widget of the current widget.

For example, the *XmQTaccessTextual* trait defines three trait methods. The simplest trait method, **StringPreferredFormat**, is shown as follows:

```
static int
StringPreferredFormat(
        Widget  w)
{
/* Choose XmFORMAT_XmSTRING because the ExmString widget holds its displayed
   text in XmString format (as opposed to Multibyte or WCS format). */
        return(XmFORMAT_XmSTRING);
}
```

By convention, you should place your widget's trait methods toward the end of your widget source code file, immediately prior to the publicly accessible functions.

5.4 Accessing Traits

As mentioned earlier in this chapter, there are two general ways for a widget to access the trait information of another widget:

- A widget can ask another widget if it holds a certain trait.
- Assuming that a widget does hold a certain trait, another widget can call that widget's trait methods.

The following subsections describe both ways.

5.4.1 Determine if a Widget Holds a Particular Trait

Use the **XmeTraitGet** function to determine if a specified widget holds a certain trait. This function returns *NULL* if the specified widget does not hold the trait, and a non-*NULL* value (a trait record) if the widget does hold the trait. For example, a parent widget can use the following code to determine if its child widget holds the *XmQTaccessTextual* trait:

```
if ( XmeTraitGet((XtPointer)a_widget_class, XmQTaccessTextual))
  /* Yes, a_widget_class renders a primary text block. */
else
  /* a_widget_class does not render a primary text block. */
```

5.4.2 Call Another Widget's Trait Method

If a widget does hold a trait, the **XmeTraitGet** function returns a pointer to the appropriate trait structure variable. Another widget can use this returned pointer to call trait methods. For example, suppose that a parent of **ExmString** needs to determine the preferred string format of **ExmString**. The following code from the parent does just that:

```
XmAccessTextualTrait  childs_trait_record;
int                   preferred_string_format_of_child;

 /* Get a pointer to the trait structure variable. */
   childs_trait_record =
      (XmQTaccessTextual) XmeTraitGet((XtPointer)a_widget_class,
                                       XmQTaccessTextual);

 /* Use the returned pointer to call the child's
                preferred_format trait method. */
preferred_string_format_of_child =
     childs_trait_record->preferred_format((Widget)parent_widget);
```

Note that, in some cases, the widget holding the trait has not defined a particular trait method; that is, the trait method is set to *NULL*. For that reason, the following code is an improvement over the previous example:

```
/* Does this trait method exist? */
if      (childs_trait_record->preferred_format !=
        (XmAccessTextualTraitRec) NULL)   {
/* It does exist, so call the trait method. */
        preferred_string_format_of_child =
           childs_trait_record->preferred_format((Widget)parent_widget);
        }
```

5.5 Overriding a Trait Record Variable

Trait record variables are declared as constants. Therefore, after being declared, a trait record variable must not be modified or deallocated. Consequently, if the subclass you are writing requires a different implementation of a trait than its superclass, the subclass should not attempt to modify or deallocate the trait record variable of its superclass. Instead, the subclass simply needs to reinstall the trait (with **XmeTraitSet**) on itself.

For example, suppose you are writing a subclass of *ExmString* called **ExmStringSubclass**. As you know, *ExmString* installs the *XmQTaccessTextual* trait. Suppose that **ExmStringSubclass** wants a different implementation of one of the trait methods of *XmQTaccessTextual* than **ExmString** does. In this case, **ExmStringSubclass** would install *XmQTaccessTextual* on itself.

Chapter 6

Resources

This chapter explains how to establish resources for Motif widgets. In particular, this chapter explains how to do the following:

- Define regular Motif resources
- Define synthetic resources
- Create and manage representation types

6.1 Defining Motif Resources

You define the resources of a Motif widget as you would define resources for any Intrinsics-based widget. Each resource is defined in a seven-field resource record. For example, the **ExmString** widget defines a resource record for a new **XmNalignment** resource as follows:

```
static XtResource resources[] =
{
  {
   XmNalignment,  /* name of resource */
   XmCAlignment,  /* class of resource */
   XmRAlignment,  /* representation type of resource value */
   sizeof(unsigned char),  /* space to hold resource value */
    /* the next field defines the position of the resource within
       the class record */
   XtOffsetOf( ExmStringRec, string.alignment),
   XmRImmediate,  /* kind of default value */
   (XtPointer) XmALIGNMENT_CENTER  /* default value */
   },
}
```

The third field of each resource record should contain the name of a Motif representation type. We will detail Motif representation types later in this chapter.

6.1.1 Overriding the Default of an Inherited Resource

Overriding the default of an inherited resource is straightforward. The subclass simply needs to redefine the resource. For example, the **XmPrimitive** widget defines the **XmNhighlightThickness** resource and sets its default value to **2**. However, the **ExmString** widget needs to change its default value from **2** to **0**. Therefore, the resource table of **ExmString** contains the following code:

```
static XtResource resources[] =
{
  {
   XmNhighlightThickness,
   XmCHighlightThickness,
   XmRHorizontalDimension,
   sizeof(Dimension),
   XtOffsetOf( XmPrimitiveRec, primitive.highlight_thickness),
   XmRImmediate,
   (XtPointer) 0  /* new default value */
  },
}
```

6.1.2 Dynamic Resource Defaulting

The sixth field of most resource records is set to **XmRImmediate**, meaning that the seventh field holds the resource's default value. However, you may wish to have Motif calculate the resource's default value at runtime. In order to do so, you must do the following:

- Set the sixth field of the resource record to **XmRCallProc**. The **XmRCallProc** pointer is the Motif version of the Intrinsics representation type named **XtRCallProc**. (See the Intrinsics documentation for more information on **XtRCallProc**.)
- Set the seventh field of the resource record to the name of a function that will be called at runtime in order to calculate the resource's value.

For example, consider the following definition of the **XmNrenderTable** resource from the *ExmString* demonstration widget:

```
{
      XmNrenderTable,
      XmCRenderTable,
      XmRRenderTable,
      sizeof(XmRenderTable),
      XtOffsetOf( ExmStringRec,string.render_table),
      XtRCallProc,
      (XtPointer) DefaultFont
},
```

The function specified in the seventh field must take three arguments:

1. The name of a widget
2. An integer offset
3. An *XrmValue* to hold the returned value of a resource

For example, the **DefaultFont** function is coded as follows:

```
static void
DefaultFont (
        Widget w,
        int offset,
        XrmValue *value)
```

```
{
 ExmStringWidgetClass wc = (ExmStringWidgetClass)XtClass(w);
 static XmRenderTable  f1;

 /* Find the default render table associated with the default
    render table type. */
  f1 = XmeGetDefaultRenderTable (w,
                     wc->string_class.default_render_table_type);

   value->addr = (XtPointer)&f1;
   value->size = sizeof(f1);
}
```

The **XmeGetDefaultPixel** function is particularly useful for dynamic resource defaulting. It returns the default background, foreground, top shadow, bottom shadow, or select colors for a given widget. Suppose you are creating a widget that displays a graph and that your widget defines a new resource, *XmNgraphSelectColor*, which holds the select color of the new widget. If you want the **XmNgraphSelectColor** resource to have a default value equal to the default select color of the widget, you must first define the resource record to take an **XmRCallProc** as follows:

```
{
 XmNgraphSelectColor,
 XmCGraphSelectColor,
 XmRPixel,
 sizeof(Pixel),
 XtOffsetOf(MyGraphRec, graph.graph_select_color),
 XmRCallProc,
 (XtPointer) GetDefaultSelectColor
},
```

Then, you have to write a function that calls **XmeGetDefaultPixel**; for example:

```
static void
GetDefaultSelectColor(
      Widget widget,
      int offset,
      XrmValue *value)
{
   XmeGetDefaultPixel (widget, XmSELECT, offset, value);
}
```

6.2 Synthetic Resources

Widget writers can mark certain resources as synthetic resources. A synthetic resource can automatically convert a resource value between two formats: external format and internal format.

Users and applications specify resource values in external format. As a widget designer, you should pick an external format that makes it easy for a user or application to convey information. However, an easy format for a widget user or application programmer may be a cumbersome or inefficient format for the widget writer. Therefore, Motif provides a synthetic resources mechanism. This mechanism automatically converts cumbersome external formats to a more efficient internal format. A well-chosen internal format expedites processing of the data or reduces its storage requirements.

Synthetic resources are the domain of the widget writer. An application programmer or user need never know that synthetic resources exist.

6.2.1 Defining Synthetic Resources

Defining a synthetic resource is a four-step process:

1. Define the synthetic resource inside the regular **resources** array.
2. Define the synthetic resource inside the **syn_resources** array.
3. Specify the name of the **syn_resources** array inside the class record.
4. Provide the routines that convert values between external and internal formats.

6.2.1.1 Step 1: Define the Synthetic Resource Inside the Regular resources Array

The first step is to define the resource as you would define any Motif resource, synthetic or not. In other words, you specify a resource record inside the **resources** array. For example, following is the resource definition of a resource named **XmNsomething**:

```
static XtResource resources[] =
{
    ...
    {
        XmNsomething,
        XmCSomething,
        XmRHorizontalDimension,
        sizeof (Dimension),
        XtOffsetOf(WidgetRec, widget.something),
        XmRImmediate,
        (XtPointer)42
    },
    ...
}
```

6.2.1.2 Step 2: Define the Synthetic Resource Inside the syn_resources Array

The second step is to generate a synthetic resource array. The base data type of the array is **XmSyntheticResource**. By convention, you should name the array variable **syn_resources**. Therefore, the framework of the synthetic resources array definition should appear as follows:

```
static XmSyntheticResource syn_resources[] =
  {
   /* one or more synthetic resource records */
  }
```

Each synthetic resource record contains the following five fields:

- The first field holds the resource name.
- The second field holds the size of the resource’s value.
- The third field holds the offset of the resource within the class record.
- The fourth field holds the name of a function that converts from the resource’s internal format to the resource’s external format. The specified function must be declared as an **XmExportProc** conversion routine. This function is called when an application calls **XtGetValues** to get the value of the resource.
- The fifth field holds the name of a function that converts from the resources’s external format to the resource’s internal format. The specified function must be

declared as an **XmImportProc** conversion routine. This function is called when the widget is created and when an application calls **XtSetValues** to modify the value of the resource.

For example, following is a sample synthetic resource definition:

```
static XmSyntheticResource syn_resources[] =
{
  ...
  {  /* Here is a synthetic resource record. */
   XmNsomething,
   sizeof (ResourceDatatype),
   XtOffsetOf(WidgetRec, widget.something),
   FromInternalToExternalFormat,
   FromExternalToInternalFormat
  }, /* End of synthetic resource record. */
  ...
};
```

You can set either the fourth or fifth field to *NULL*. Doing so tells Motif that you will not be supplying part of the conversion process. For example, if the fourth field is set to *NULL*, then this widget is not supplying a function to convert the resource value from internal format to external format. If the fourth field is *NULL*, a call to **XtGetValues** returns the resource's value in internal format.

6.2.1.3 Step 3: Specify the syn_resources Array Inside the Class Record

You must specify the name of the synthetic resource array in the appropriate portion of the class record. You must also specify the number of synthetic resources. For example:

```
/* syn_resources */              syn_resources,
/* num_syn_resources */          XtNumber(syn_resources),
```

6.2.1.4 Step 4: Provide the Conversion Routines

The final step is to provide the synthetic resource conversion routines themselves. In some cases, you can use one of Motif’s existing routines (such as, **XmeFromHorizontalPixels**). In other cases, you will have to write the synthetic resource conversion routines yourself. There are two kinds of conversion routines you can write:

- **XmExportProc** conversion routines, which convert a resource value from internal format to external format
- **XmImportProc** conversion routines, which convert a resource value from external format to internal format

An **XmExportProc** conversion routine has the following prototype:

```
void MyExportProcConversionRoutine(
        Widget     wid,
        int           offset,
           XtArgVal     *value)
```

wid	Specifies the widget which is requesting a synthetic resource conversion.
offset	Specifies the offset of a synthetic resource field in the widget record.
value	Specifies a pointer to a resource value in internal format and returns a pointer to that resource value converted to external format.

An **XmImportProc** conversion routine has the following prototype:

```
XmImportOperator MyImportProcConversionRoutine(
   Widget        wid,
           int      offset,
           XtArgVal      *value)
```

wid	Specifies the widget which is requesting a synthetic resource conversion.
offset	Specifies the offset of a synthetic resource field in the widget record.
value	Specifies a pointer to a resource value in external format. If the **XmImportProc** conversion routine returns **XmSYNTHETIC_LOAD**, then *value* must return the input resource value converted to internal format.

An **XmImportProc** conversion routine must return an **XmImportOperator** value, which is an enumerated type having the following possible values:

- **XmSYNTHETIC_NONE**, which means that the caller of the **XmImportProc** is not responsible for copying the converted *value* into the resource specified by *offset*.
- **XmSYNTHETIC_LOAD**, which means that the caller of the **XmImportProc** is responsible for copying the converted *value* into the resource specified by *offset*.

Motif's synthetic resource mechanism is typically the caller of the **XmImportProc**. Therefore, if your **XmImportProc** conversion routine returns **XmSYTHETIC_LOAD**, Motif synthetic resource mechanism will take care of copying (and casting) *value* into the resource specified by *offset*.

6.2.2 Distance Conversions

The synthetic resource implementation is particularly helpful for converting resource values from real-world units (like millimeters) to pixels and back again.

The Xlib and Xme drawing routines all expect pixel arguments. Unfortunately, pixels are not always a very handy unit for widget users to work in. One important problem with pixel units is that a pixel's size depends on the resolution of the screen.

To solve this problem the Motif base classes **XmPrimitive** and **XmManager** both provide the **XmNunitType** resource. This resource holds the kind of units that a given distance or position is measured in. By default, the unit type is pixels. However, the application may specify a unit type of some real-world unit, like millimeters, instead. Any resource whose value symbolizes a size, position, or distance must be able to convert these real-world units to and from pixels.

The synthetic resource mechanism provides a way to do these conversions. In fact, you do not even have to write conversion routines because Motif provides four unit conversion routines for you:

- **XmeFromHorizontalPixels** and **XmeFromVerticalPixels** convert values from any unit type named by the **XmNunitType** resource to pixels.
- **XmeToHorizontalPixels** and **XmeToVerticalPixels** convert values from pixels to the unit type named by the **XmNunitType** resource.

For example, the **ExmSimple** widget uses all four of these routines in its **syn_resources** array as follows:

```
static XmSyntheticResource syn_resources[] =
{
    {
         XmNmarginWidth,
               sizeof (Dimension),
               XtOffsetOf( ExmSimpleRec, simple.margin_width),
               XmeFromHorizontalPixels,
               XmeToHorizontalPixels
    },
    {
               XmNmarginHeight,
               sizeof (Dimension),
         XtOffsetOf( ExmSimpleRec, simple.margin_height),
         XmeFromVerticalPixels,
               XmeToVerticalPixels
    },
};
```

6.2.3 How the Synthetic Resource Mechanism Works

The synthetic resource mechanism is completely transparent to the widget writer. This is because the Motif base classes, **XmPrimitive** and **XmManager**, call the appropriate conversion routines at the appropriate times.

For example, the **class_part_initialize** method of **XmPrimitive** and **XmManager** both contain code that initializes the synthetic resource. In other words, both base classes call the routine that converts all the synthetic resource values from external to internal format Similarly, both base classes contain code in their **set_values** routines to call the external to internal format conversion routine whenever one of the resource values is changed. Finally, both base classes contain code in their **get_values_hook** routines to call the internal-to-external format conversion routine whenever an application requests the resource value.

Since the **class_part_initialize**, **set_values**, and **get_values_hook** methods are all chained, these conversion routines are automatically called for all primitive and manager widgets.

6.3 Representation Types

The third field of every Motif resource record holds the name of a Motif representation type. Motif representation types, like Xt representation types, know how to convert resource values into a different C language representation. Motif associates all its representation types with converters that convert data from one C data type (for example, a **String**) to another (for example, an **int**).

Motif representation types have the **XmR** prefix.

You should become familiar with the predefined Motif representation types. You should also know how to create your representation types when conditions warrant.

6.4 Enumerated Representation Types

Suppose you are writing a widget that defines an enumerated resource. In other words, the resource requires an enumerated constant as a value. In picking the representation type, you can do either of the following:

- Pick a predefined representation type
- Register (create) a new representation type

The following subsections examine both categories.

6.4.1 Predefined Enumerated Representation Types

If you are defining a new enumerated resource that serves a similar purpose to an existing Motif enumerated resource, then you should probably pick an existing Motif enumerated representation type rather than creating your own. Motif predefines the following enumerated representation types:

- **XmRArrowDirection** defines the following registered constants: **XmARROW_UP**, **XmARROW_DOWN**,**XmARROW_LEFT**, and **XmARROW_RIGHT**. For information on these constants, see the description of **XmNarrowDirection** in **XmArrowButton**(3).

- **XmRAlignment** defines the following registered constants: **XmALIGNMENT_BEGINNING**, **XmALIGNMENT_CENTER**, and **XmALIGNMENT_END**. For information on these constants, see the description of **XmNalignment** in **XmLabel**(3).

- **XmRDirection** defines the following registered constants:

  — **XmRIGHT_TO_LEFT_TOP_TO_BOTTOM**

  — **XmLEFT_TO_RIGHT_TOP_TO_BOTTOM**

  — **XmRIGHT_TO_LEFT_BOTTOM_TO_TOP**

  — **XmLEFT_TO_RIGHT_BOTTOM_TO_TOP**

  — **XmTOP_TO_BOTTOM_RIGHT_TO_LEFT**

  — **XmTOP_TO_BOTTOM_LEFT_TO_RIGHT**

  — **XmBOTTOM_TO_TOP_RIGHT_TO_LEFT**

  — **XmBOTTOM_TO_TOP_LEFT_TO_RIGHT**

  For information on these constants, see the description of **XmRDirection** in **XmLabel**(3)

- **XmRIndicatorType** defines the following registered constants: **XmINDICATOR_NONE**, **XmINDICATOR_FILL**, **Xm_INDICATOR_BOX**, **XmINDICATOR_CHECK**, **XmINDICATOR_CHECK_BOX**, **XmINDICATOR_CROSS**, and **XmINDICATOR_CROSS_BOX**. For information on these constants, see the description of **XmRIndicator** in **XmToggleButton**(3).

- **XmRMultiClick** defines the following registered constants: **XmMULTICLICK_KEEP** and **XmMULTICLICK_DISCARD**. For information on these constants, see the description of **XmPushButton** in **XmNmultiClick**(3)

- **XmRNavigationType** defines the following registered constants: **XmNONE**, **XmTAB_GROUP**, **XmSTICKY_TAB_GROUP**, and **XmEXCLUSIVE_TAB_GROUP**. For information on these constants, see the description of **XmNnavigationType** in **XmPrimitive**(3) or **XmManager**(3).

- **XmROrientation** defines the following registered constants: **XmVERTICAL** and **XmHORIZONTAL**. For information on these constants, see the description of **XmNorientation** in **XmScrollBar**(3).

- **XmRSelectionMode** defines the following registered constants: **XmNORMAL_MODE** and **XmADD_MODE**. For information on these constants, see the description of **XmNselectionMode** in **XmList**(3).
- **XmRSeparatorType** defines the following registered constants: **XmSINGLE_LINE**, **XmDOUBLE_LINE**, **XmSINGLE_DASHED_LINE**, **XmDOUBLE_DASHED_LINE**, **XmNO_LINE** **XmSHADOW_ETCHED_IN**, **XmSHADOW_ETCHED_OUT**, **XmSHADOW_ETCHED_IN_DASH**, and **XmSHADOW_ETCHED_OUT_DASH**. For information on these constants, see the description of **XmNseparatorType** in **XmSeparator**(3).
- **XmRShadowType** defines the following registered constants: **XmSHADOW_IN**, **XmSHADOW_OUT**, **XmSHADOW_ETCHED_IN** and **XmSHADOW_ETCHED_OUT**. For information on these constants, see the description of **XmNshadowType** in **XmFrame**(3).
- **XmRUnitType** defines the following registered constants: **XmPIXELS**, **Xm100TH_MILLIMETERS**, **Xm1000TH_INCHES**, **Xm100TH_POINTS**, **Xm100TH_FONT_UNITS**, **XmINCHES**, **XmCENTIMETERS**, **XmMILLIMETERS**, **XmPOINTS**, and **XmFONT_UNITS**. For information on these constants, see the description of **XmNunitType** in **XmPrimitive**(3)or **XmManager**(3).
- **XmRVisualEmphasis** defines the following registed constants: **XmSELECTED** and **XmNOT_SELECTED**. For information on these constants, see the description of **XmNvisualEmphasis** in **XmIconGadeget**(3).

For example, the previous list shows that Motif predefines the representation type **XmRAlignment**. The only legal values that can be assigned to a resource having this representation type are **XmALIGNMENT_BEGINNING**, **XmALIGNMENT_CENTER**, and **XmALIGNMENT_END**. For details on what these three constants symbolize, see the description of the **XmNalignment** resource in the **XmLabel** reference page. This reference page appears in the *Motif 2.1—Programmer's Reference*. When defining a resource with similar requirements to **XmNalignment**, you should specify a representation type of **XmRAlignment**. For example, the *ExmString* widget defines a resource named **XmNalignment** that controls the alignment of text within the widget. The wise choice for the data type of **XmNalignment** is **XmRAlignment**, as shown in the following resource record:

```
...
{
 XmNalignment,
 XmCAlignment,
 XmRAlignment,
 sizeof(unsigned char),
 XtOffsetOf( ExmStringRec,string.alignment),
 XmRImmediate,
 (XtPointer) XmALIGNMENT_CENTER
},
...
```

One of the primary responsibilities of a widget is to check for valid resource values. Using predefined representation types simplifies this chore by turning it into an easy two-step process:

1. Call the **XmRepTypeGetId** function to get the identification number of the representation type.
2. Call the **XmRepTypeValidValue** function to check the validity of the resource's value.

For example, the **class_initialize** method of the *ExmString* widget gets the identification number with the following code:

```
static XmRepTypeId alignmentId;
  alignmentId = XmRepTypeGetId(XmRAlignment);
```

The **initialize** and **set_values** methods of the **ExmString** widget check resource values as follows:

```
Boolean   ValidValue;
  ValidValue = XmRepTypeValidValue(alignmentId,
                                   my_widget->string.alignment,
                                   my_widget);
  if (!ValidValue)
    /* take appropriate corrective response */
```

6.4.2 Defining Your Own Enumerated Representation Types

If none of the predefined representation types match the needs of your resource, then you can create a new representation type. Registering a new representation type is a five-step process:

1. In the widget source code file, you must declare an array variable to hold the normalized names of the possible values of the resource.
2. In the widget source code file, you must declare a global variable of type **XmRepTypeId**.
3. In the widget source code file (probably in the **class_initialize** or **class_part_initialize** method), you must call the **XmRepTypeRegister** routine.
4. In the widget public header file, you must define a string equivalent for the new representation type.
5. In the widget public header file, you must specify enumerated constants for the possible values of the new representation type.

For example, the **ExmSimple** widget defines a representation type named *ExmRSimpleShape*. The following is the declaration of the array variable (Step 1):

```
static String SimpleShapeNames[] = {
        "simple_oval",
        "simple_rectangle"
};
```

And following is the declaration of the **XmRepTypeId** variable (Step 2):

```
static XmRepTypeId simpleShapeId;
```

The **class_initialize** method calls **XmRepTypeRegister** as follows (Step 3):

```
simpleShapeId = XmRepTypeRegister (ExmRSimpleShape, SimpleShapeNames,
                                   NULL, XtNumber(SimpleShapeNames));
```

Notice how the array variable created in Step 1 is passed as an argument to **XmRepTypeRegister**. Furthermore, note that the **XmRepTypeId** variable is assigned the return value.

The public header file declares a string equivalent of the new representation type as follows (Step 4):

```
#define ExmRSimpleShape "ExmSimpleShape"
```

The public header file also specifies the appropriate enumerated constants (Step 5):

```
enum { ExmSHAPE_OVAL=0, ExmSHAPE_RECTANGLE=1 };
```

6.4.3 The Size of Enumerated Representation Types

The fourth field of all resource records holds the size of the resource value. If the resource has an enumerated representation type, then the fourth field should be set as follows:

```
sizeof(unsigned char)
```

6.5 Nonenumerated Representation Types

Representation types are also useful for defining nonenumerated resource values. Table 6-1 lists all the useful nonenumerated Motif representation types and their C data type equivalents.

Table 6–1. Common Nonenumerated Representation Types

Motif Representation Type	**Converts to C Data Type**
XmRHorizontalDimension	**Dimension**
XmRHorizontalPosition	**Position**
XmRVerticalDimension	**Dimension**
XmRVerticalPosition	**Position**
XmRBitmap	**Pixmap**
XmRPixmap	**Pixmap**
XmRDynamicPixmap	**Pixmap**
XmRRenderTable	**XmRenderTable**
XmRXmString	**XmString**
XmRXmStringTable	**XmString ***

XmRTabList	**XmTabList**
XmRValueWcs	**wchar_t ***

The fourth field of each resource record specifies the size required to store the resource's value. The second column of Table 6-2 will help you determine this size. For example, the **ExmString** widget defines a resource record for the **XmNrenderTable** resource.This resource defines a representation type of **XmRRenderTable**. According to the table, the C data type associated with this representation type is **XmRenderTable**. Therefore, the resource record specifies the size of the resource's value as follows:

```
{
        XmNrenderTable,
        XmCRenderTable,
        XmRRenderTable,
        sizeof(XmRenderTable),  /* size of the resource */
        XtOffsetOf( ExmStringRec,string.render_table),
        XtRCallProc,
        (XtPointer) DefaultFont
},
```

6.5.1 Pixmap Converter for Scaling

To support the automatic scaling of **Pixmap**s, the following representation types are available:

- **XmRNoScalingBitmap**
- **XmRNoScalingDynamicPixmap**

The converters for these representation types ignore the scaling ratio that might be present in their print shell hierarchy and behave normally; that is, they convert *XBM* and *XPM* files to **Pixmap**. These are used when **Pixmap** is employed for tiling, rather than when it is used for images and icons.

The semantics of the **XmRBitmap** and **XmRDynamic** representation type converters apply a scaling ratio to the resulting **Pixmap** that is equal to the print shell resolution divided by the value of the **XmNdefaultPixmapResolution** of their **XmPrintShell**. No scaling is applied if the widget for which a **Pixmap** resource is being converted is not a descendant of **XmPrintShell**. See Chapter 16 for more information on **XmPrintShell**.

6.5.2 Dimensions and Positions Representation Types

Resources whose values symbolize dimensions or positions should use the following representation types:

- **XmRHorizontalDimension**, which converts a horizontal dimension to a **Dimension**.
- **XmRHorizontalPosition**, which converts a horizontal position to a **Position**.
- **XmRVerticalDimension**, which converts a vertical dimension to a **Dimension**.
- **XmRVerticalPosition**, which converts a vertical position to a **Position**.

For example, the **ExmSimple** widget associates the **XmRHorizontalDimension** representation type with the **XmNmarginWidth** resource. Similarly, the *XNmarginHeight* resource is associated with the **XmRVerticalDimension** representation type. The unit type conversion functions (**XmeFromHorizontalPixels**, **XmeFromVerticalPixels**, **XmeToHorizontalPixels**, and **XmeToVerticalPixels**) depend on these representation types.

6.5.3 Bitmap and Pixmap Representation Types

Motif provides the following three representation types for converting bitmaps and pixmaps.

- **XmRBitmap**, which converts an input **xbm** file to a **Pixmap** of depth 1.
- **XmRPixmap**, which converts an input **xbm** or **xpm** file to a **Pixmap** with the same depth as the widget’s visual.
- **XmRDynamicPixmap**, which depends on the value of the **XmNbitmapConversionModel** resource of **XmScreen**. If this resource’s value is **XmMATCH_DEPTH**, **XmRDynamicPixmap** converts an input **xbm** or **xpm** file to a **Pixmap** of appropirate depth for the widget. If this resource’s value is **XmDYNAMIC_DEPTH**, **XmRDynamicPixmap** converts an input **xbm**

file to a **Pixmap** of depth 1 and an input **xpm** file to a **Pixmap** the same depth as the widget.

6.6 Motif Representation Types in XmPrimitive and XmManager

Tables 6-2 and 6-3 describe the representation types used by the resources of **XmPrimitive** and **XmManager**.

Table 6–2. Representation Types for the Resources of XmPrimitive

Resource	**Representation Type**
XmNbottomShadowColor	**XmRPixel** (synonym for **XtRPixel)**
XmNbottomShadowPixmap	**XmRNoScalingDynamicPixmap**
XmNconvertCallback	**XmRCallProc** (synonym for **XtRCallProc)**
XmNforeground	**XmRPixel** (synonym for **XtRPixel)**
XmNhelpCallback	**XmRCallProc** (synonym for **XtRCallProc)**
XmNhighlightColor	**XmRPixel** (synonym for **XtRPixel)**
XmNhighlightOnEnter	**XmRBoolean** (synonym for **XtRBoolean)**
XmNhighlightPixmap	**XmRNoScalingDynamicPixmap**
XmNhighlightThickness	**XmRHorizontalDimension**
XmNnavigationType	**XmRNavigationType**
XmNpopupHandlerCallback	**XmRCallProc** (synonym for **XtRCallProc)**
XmNshadowThickness	**XmRHorizontalDimension**
XmNtopShadowColor	**XmRPixel** (synonym for **XtRPixel)**
XmNtopShadowPixmap	**XmRNoScalingDynamicPixmap**
XmNtraversalOn	**XmRBoolean** (synonym for **XtRBoolean)**

XmNunitType	**XmRUnitType**
XmNuserData	**XmRPointer** (synonym for **XtRPointer)**

Table 6–3. Representation Types for the Resources of XmManager

Resource	**Representation Type**
XmNbottomShadowColor	*XmRPixel*
XmNbottomShadowPixmap	**XmRNoScalingDynamicPixmap**
XmNforeground	**XmRPixel**
XmNhelpCallback	**XmRCallback**
XmNhighlightColor	**XmRPixel**
XmNhighlightPixmap	**XmRNoScalingDynamicPixmap**
XmNinitialFocus	**XmRWidget**
XmNlayoutDirection	**XmRDirection**
XmNnavigationType	**XmRNavigationType**
XmNpopupHandlerCallback	**XmRCallback**
XmNshadowThickness	**XmRHorizontalDimension**
XmNstringDirection	**XmRStringDirection**
XmNtopShadowColor	*XmRPixel*
XmNtopShadowPixmap	**XmRNoScalingDynamicPixmap**
XmNtraversalOn	**XmRBoolean**
XmNunitType	**XmRUnitType**
XmNuserData	**XmRPointer**

XmPrimitive and **XmManager** both provide representation types for the same seven resources of **Core**. Table 6-4 summarizes these representation types.

Table 6–4. Representation Types for Core Resources

Resource	**Representation Type**
XmNbackgroundPixel	**XmRPixel (**synonym for **XtRPixel**)
XmNbackgroundPixmap	**XmRPixmap**
XmNborderWidth	**XmRHorizontalDimension**
XmNheight	**XmRVerticalDimension**
XmNwidth	**XmRHorizontalDimension**
XmNx	**XmRHorizontalPosition**
XmNy	**XmRVerticalPosition**

Chapter 7
Translations and Actions

This chapter explains what Motif widget writers need to know about translations and actions. A closely related concept, keyboard traversal, is also detailed in this chapter.

7.1 Defining Translations in the Class Record

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

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

For example, the **ExmMenuButton** widget provides the following string for its **tm_table** field:

```
static char defaultTranslations[] =
"<EnterWindow>:                MenuButtonEnter()\n\
<LeaveWindow>:                    MenuButtonLeave()\n\
<BtnDown>:                   BtnDown()\n\
<BtnUp>:                        BtnUp()\n\
:<Key>osfActivate:      ArmAndActivate()\n\
:<Key>osfCancel:           MenuEscape()\n\
:<Key>osfHelp:                    MenuButtonHelp()\n\
~s ~m ~a <Key>Return:        ArmAndActivate()\n\
~s ~m ~a <Key>space:    ArmAndActivate()";
```

and the following array for the *actions* field:

```
static XtActionsRec Actions[] = {
           {"ArmAndActivate",   ArmAndActivate},
           {"BtnDown",                BtnDown},
           {"BtnUp",                  BtnUp},
           {"MenuButtonEnter",  MenuButtonEnter},
           {"MenuButtonLeave",  MenuButtonLeave},
           {"MenuButtonHelp",   MenuButtonHelp}
};
```

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

- *NULL*, meaning that the translations described by the **tm_table** string of the **CoreClassPart** are the only translations defined in the class record.
- **XtInheritTranslations**, meaning that you are inheriting the translations field of your superclass.
- A *translations* string of your own devising. If your widget does do this, the Intrinsics will place the additional translations at the top of the translations table. Therefore, the additional translations take precedence over the translations defined in **CoreClassPart**. (See the *ExmMenuButton* demonstration widget for an example of an additional translations string.)

Most standard Motif widgets set the additional translation field either to *NULL* or to **XtInheritTranslations**. If you are subclassing the **XmPrimitive** widget and specify **XtInheritTranslations**, you will inherit the following actions:

```
<Unmap>:                        PrimitiveUnmap()\n\
<FocusIn>:                   PrimitiveFocusIn()\n\
<FocusOut>:                   PrimitiveFocusOut()\n\
:<Key>osfActivate:      PrimitiveParentActivate()\n\
:<Key>osfCancel:           PrimitiveParentCancel()\n\
:<Key>osfBeginLine:     PrimitiveTraverseHome()\n\
:<Key>osfUp:               PrimitiveTraverseUp()\n\
:<Key>osfDown:                     PrimitiveTraverseDown()\n\
:<Key>osfLeft:                     PrimitiveTraverseLeft()\n\
:<Key>osfRight:                 PrimitiveTraverseRight()\n\
~s ~m ~a <Key>Return:        PrimitiveParentActivate()\n\
s ~m ~a <Key>Tab:            PrimitivePrevTabGroup()\n\
~m ~a <Key>Tab:                 PrimitiveNextTabGroup()";
```

(All these actions are documented in the reference page for **XmPrimitive**; see the *Motif 2.1—Programmer's Reference* for details.) By specifying **XtInheritTranslations** in a subclass of **XmPrimitive**, your widget will automatically inherit the standard keyboard traversal translations for primitive widgets.

If you are subclassing the **XmManager** widget and specify **XtInheritTranslations**, your subclass will inherit the following actions:

```
<EnterWindow>:                  ManagerEnter()\n\
<LeaveWindow>:                  ManagerLeave()\n\
<FocusOut>:                ManagerFocusOut()\n\
<FocusIn>:                 ManagerFocusIn()\n\
:<Key>osfBeginLine:  ManagerGadgetTraverseHome()\n\
:<Key>osfUp:            ManagerGadgetTraverseUp()\n\
:<Key>osfDown:                  ManagerGadgetTraverseDown()\n\
:<Key>osfLeft:                  ManagerGadgetTraverseLeft()\n\
:<Key>osfRight:              ManagerGadgetTraverseRight()\n\
s ~m ~a <Key>Tab:       ManagerGadgetPrevTabGroup()\n\
~m ~a <Key>Tab:              ManagerGadgetNextTabGroup()";
```

(All these actions are documented in the reference page for **XmManager**; see the *Motif 2.1—Programmer's Reference* for details.)

7.2 Conflicts Between tm_table and translations

The translations defined by the **translations** field take precedence over the translations defined in the **tm_table** field. In other words, if the same event appears in both **translations** and **tm_table**, then the translation appearing in **tm_table** will override the translation appearing in **translations**. This rather simple rule can be the source of subtle problems in widget writing.

7.3 Virtual Keysyms

Xlib and the Intrinsics provide a level of mapping between physical keys and keysyms. Motif adds one more level of mapping on top of keysyms; Motif can map keysyms to virtual keysyms. Table 7-1 describes the purpose of some common Motif virtual keysyms.

Table 7–1. Purpose of Common Motif Virtual Keysyms

Virtual Keysyms	**Purpose**
osfActivate	Activates a default action
osfAddMode	Toggles the selection mode between Normal and Add mode
osfBackSpace	Deletes the previous character
osfBeginLine	Moves the cursor to the beginning of the line
osfCancel	Cancels the current operation
osfClear	Clears the current selection
osfCopy	Copies to the clipboard
osfCut	Cuts to the clipboard
osfDelete	Deletes data
osfDeselectAll	Deselects the current selection
osfDown	Moves the cursor down
osfEndLine	Moves the cursor to the end of the line

osfHelp	Calls the function specified by the **XmNhelpCallback** resource
osfInsert	Toggles between Replace and Insert mode (if specified without modifiers)
osfLeft	Moves the cursor left
osfMenu	Activates the Popup Menu
osfMenuBar	Traverses to the MenuBar
osfPageDown	Moves down one page
osfPageLeft	Moves left one page
osfPageRight	Moves right one page
osfPageUp	Moves up one page
osfPaste	Pastes from the clipboard
osfPrimaryPaste	Pastes the primary selection
osfRestore	Restores a previous setting
osfRight	Moves the cursor right
osfSelect	Establishes the current selection
osfSelectAll	Selects an entire block of data
osfSwitchDirection	Toggles the string layout direction
osfUndo	Undoes the most recent action
osfUp	Moves the cursor up

We recommend that the translations string of Motif widgets use Motif virtual keysyms instead of traditional keysyms whenever possible. For example, the following well-defined translations string uses five different Motif virtual keysyms:

```
static char defaultTranslations[] = "\
:<Key>osfActivate:      PrimitiveParentActivate()\n\
:<Key>osfCancel:        PrimitiveParentCancel()\n\
:<Key>osfHelp:          PrimitiveHelp()\n\
:<Key>osfDelete:            MyDeletionMethod()\n\
:<Key>osfInsert:            MyInsertionMethod()\n\
```

Virtual keysyms let your widget provide consistent behavior across a wide variety of keyboards. (For more information on virtual keysyms, refer to the *CDE 2.1/Motif 2.1—Style Guide and Glossary* and to the **VirtualBindings** reference page of the *Motif 2.1—Programmer's Reference*.)

The following subsections take a closer look at some commonly used bindings for certain virtual keysyms.

7.3.1 The osfActivate and osfCancel Virtual Keysyms

The **PrimitiveParentActivate** action is ordinarily bound to the osfActivate virtual keysym. The **PrimitiveParentCancel** action is ordinarily bound to the osfCancel virtual keysym.

When activated, the **PrimitiveParentActivate** action routine typically does the following:

- Examines the value of the **XmNdefaultButton** resource in the managing widget. This resource holds the widget ID of one of the widgets that it is managing.
- Invokes the **arm_and_activate** method of the widget named by **XmNdefaultButton**.

In other words, the ultimate responsibility for handling the **PrimitiveParentActivate** action belongs to the widget named by **XmNdefaultButton**.

The **PrimitiveParentCancel** action routine works like **PrimitiveParentActivate**. The only difference is that **PrimitiveParentCancel** depends on the **XmNcancelButton** resource rather than **XmNdefaultButton**.

(See Chapter 4 for more details on the **parent_process** method.)

7.3.2 The osfHelp Virtual Keysym

You may associate any action routine with the osfHelp virtual keysym. One interesting choice of an action routine is **PrimitiveHelp**.

Specifying **PrimitiveHelp** causes Motif to invoke the callback associated with the widget's **XmNhelpCallback** resource. It is quite possible, however, that your widget does not define such a callback. In that case, **PrimitiveHelp** looks for a help callback inside the widget that is managing your widget.

For example, the **ExmString** widget binds the **PrimitiveHelp** action to the osfHelp virtual keysym. Suppose that an **XmForm** widget is managing several *ExmString* widgets. Furthermore, suppose that none of the *ExmString* widgets define a help callback. In this case, **PrimitiveHelp** automatically tries to find a help callback inside the **XmForm** widget. Therefore, the help callback of **XmForm** can serve as the help callback for all the widgets that it manages. If the **XmForm** widget does not define a help callback, then Motif looks for a help callback in the widget that manages the **XmForm** widget. Motif keeps going up the parent chain until it finds a help callback.

7.3.3 The osfMenuBar and osfMenu Virtual Keysyms

You may associate any action with the virtual keysyms osfMenuBar and osfMenu; however, there is a good possibility that Motif will never execute the action you specify.

If your application defines a menu bar, the menu bar will grab any osfMenuBar events before your widget can process it. Similarly, if your application defines any popup menus, the popup menu widget will grab any osfMenu events before your widget can process it.

7.3.4 The osfBeginLine, osfEndLine, osfLeft, osfRight, osfUp, and osfDown Virtual Keysyms

You should be very careful about defining any of the following virtual keysyms inside the **tm_table** string of the **CoreClassPart**:

- osfBeginLine
- osfEndLine
- osfLeft
- osfRight

- osfUp
- osfDown

As we mentioned earlier in this chapter, the **translations** field of the **PrimitiveClassPart** takes precedence over the **tm_table** field of the **CoreClassPart**. The **PrimitiveClassPart** of **XmPrimitive** defines actions for all six of the preceding virtual keysyms. Therefore, if your widget inherits these translations (by specifying **XtInheritTranslations**), then Motif will ignore any translations defined for these virtual keysyms in the **tm_table** field. Therefore, if you really do want to provide a non-default binding for any of these virtual keysyms, you should set the **translations** field of the **PrimitiveClassPart** to either *NULL* or to the name of a new **translations** string. Another possibility is to update the **translations** string as part of your widget's *initialize* method.

7.4 Enter Actions

The X Window System generates an **EnterWindow** event whenever the pointer crosses into your widget. In order to get the proper traversal behavior, whatever action is bound to **EnterWindow** needs to call the **PrimitiveEnter** action of **XmPrimitive**. This action encapsulates the appropriate Motif response to an **EnterWindow** event. In short, this action does the following:

- Gives this widget the keyboard focus if the keyboard focus policy is *XmImplicit*. (If the keyboard focus policy is *XmExplicit*, the **PrimitiveEnter** method does not change the keyboard focus.)
- Highlights the widget if **primitive.highlight_on_enter** is set to **True**.

The simplest way to invoke **PrimitiveEnter** is to associate it with **EnterWindow** in the *actions* array as follows:

```
static XtActionsRec ActionsList[] = {
      {"Enter",         PrimitiveEnter},
      ...
}
```

If your widget requires special behavior on an **EnterWindow**, then your widget's **EnterWindow** action should envelop **PrimitiveEnter**. For example, the following action reacts properly:

```
MyEnterEventMethod(Widget    w,
                   XEvent   *event,
                   String   *params,
                   Cardinal *num_params)
{
 /* special behavior on <EnterWindow> */
   ...

 /* Call the PrimitiveEnter function. */
   XtCallActionProc(w, "PrimitiveEnter", event, params, num_params);
}
```

7.5 Leave Actions

The X Window System generates a **LeaveWindow** event whenever the pointer passes out of your widget. In order for traversal to work properly, your widget must call the **PrimitiveLeave** action of **XmPrimitive**. This action encapsulates the appropriate Motif response to a **LeaveWindow** event. In short, this action does the following:

- Removes the keyboard focus from this widget if the keyboard focus policy is *XmImplicit*. (If the keyboard focus policy is *XmExplicit*, the **PrimitiveLeave** action does not change the keyboard focus.)
- Unhighlights the widget if **primitive.highlight_on_enter** is set to **True**.

You can invoke **PrimitiveLeave** either from within the *actions* array or by enveloping it inside an action.

7.6 Mouse Bindings

The *CDE 2.1/Motif 2.1—Style Guide and Glossary* defines a model for mouse button operations. By following this model, your widget's behavior will be consistent with other Motif widgets. A brief description of the bindings for a 3-button mouse appears in Table 7-2:

Table 7–2. Motif Mouse Keysyms

Physical Key	**Virtual Buttons**	**Purpose**
Button1	**Select**	Selects and activates
Button2	**Transfer**	Transfers data, including primary paste and drag operations
Button3	**Menu**	Activates Popup Menus

For a more detailed description of mouse button bindings, see the *CDE 2.1/Motif 2.1—Style Guide and Glossary*. Note that, unlike virtuals keysyms, virtual button bindings may not be used in translation tables.

7.7 Keyboard Traversal

Most Motif users move a mouse in order to move the cursor from one widget to another. However, the *CDE 2.1/Motif 2.1—Style Guide and Glossary* insists that Motif applications be usable even when a mouse is not available. For that reason, most Motif applications allow users to traverse between widgets by pressing the tab key or by pressing one of the arrow keys.

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

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

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

The **XmNnavigationType** resource determines whether the widget is a tab group. The **XmPrimitive** widget establishes a default value of **XmNONE** for this resource. By contrast, the **XmManager** widget sets the default to *XmTabGroup*. Widget writers may want to set different defaults. (See the *Motif 2.1—Programmer's Guide* for complete details on the various kinds of tab groups.)

The preceding suggestions tell you everything you really need to know in order to handle keyboard traversal in most widgets. However, a few specialized widgets require special care. For example, in the standard Motif widget set, the **XmText** widget falls into this "special care" category because it reserves the tab key and the arrow keys for purposes other than widget traversal. If you are writing a widget that falls into this category, you will not be able to inherit the traversal translations of **XmPrimitive** and **XmManager**. Instead, your widget will have to provide its own actions.

Chapter 8
Using Xme Functions Within a Widget

Motif provides widget writers with several dozen convenience routines known collectively as the Xme functions. Motif provides these to simplify widget writing. Widget writers should use these Xme functions whenever possible.

The Xme routines perform the following kinds of widget-writing services:

- Render Motif-style geometric objects
- Implement resolution independence
- Implement data transfer between and within widgets
- Handle traits
- Handle other miscellaneous services pertaining to widget writing

This chapter provides an overview of the Xme functions. Chapter 17 contains reference documentation for each Xme function.

Xme functions were not available in releases of Motif prior to Release 2.0. Prior to Release 2.0, Motif provided several dozen undocumented _Xm routines for widget writers and for internal use. Motif strongly recommends that you not use any of these _Xm routines in your widgets.

Many of the _Xm routines have Xme equivalents. For example, **_XmClearBorder** is equivalent to **XmeClearBorder** . To help users make the transition from _Xm routines to Xme routines, the Motif Release 2.0 **XmP.h** header file contains **#define** statements that convert *some* of the old _Xm calls into their new Xme equivalents. Thus, if your code calls any of the _Xm routines listed in these **#define** statements, and you compile against the Release 2.0 header files, the C preprocessor will automatically convert the _Xm calls to the proper Xme calls. However, Motif does not guarantee that these **#define** statements will continue to be available in future releases.

8.1 Calling an Xme function

The Xme functions are bound into the standard Motif library (typically stored in file **libXm.a**). Therefore, you do not need to bind your widget with any special libraries in order to call an Xme function. Similarly, you do not need a Motif source license in order to call an Xme function.

You call an Xme function just as you would call any Xm convenience function. That is, you must include the appropriate header file, and you must provide the correct type and number of arguments to the call. The reference pages in Chapter 17 detail the required information for each Xme function.

8.2 Xme Trait Functions

Motif provides two Xme functions for handling traits. These functions are summarized in Table 8-1. See Chapter 5 for details on traits.

Table 8–1. Xme Trait Functions

Xme Function	**What It Does:**
XmeTraitGet	Returns the trait record associated with a given object
XmeTraitSet	Installs a trait on a specified widget class

8.3 Rendering Geometric Objects

Motif provides several Xme functions for rendering geometric objects (arrows, diamonds, and so on). Whenever possible, your Motif widgets should render geometric objects with these Xme routines. By using the Xme functions, you are ensuring that the geometric objects in your Motif widget will look like the geometric objects in other Motif widgets.

Table 8-2 lists the Xme functions pertaining to drawing.

Table 8–2. Xme Drawing Functions

Xme Function	What It Does:
XmeClearBorder	Clears any rectangular area (such as a widget border)
XmeDrawArrow	Draws a Motif-style, three-dimensional arrow
XmeDrawCircle	Draws a Motif-style, three-dimensional circle
XmeDrawDiamond	Draws a Motif-style, three-dimensional diamond
XmeDrawHighlight	Draws a Motif-style highlight around a widget's border
XmeDrawIndicator	Draws a Motif-style cross or check mark
XmeDrawPolygonShadow	Draws a Motif-style, three-dimensional shadow around a polygon
XmeDrawSeparator	Draws a Motif-style, three-dimensional line to separate two visual components
XmeDrawShadows	Draws a Motif-style, three-dimensional shadow around a rectangle

For example, the **ExmSimple** widget calls the **XmeDrawShadows** function to draw its border shadows. In order to make this call, the **Simple.c** file includes the header file for the Xme drawing functions as follows:

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

The call to **XmeDrawShadows** appears as follows:

```
XmeDrawShadows (XtDisplay (sw), XtWindow (sw),
                sw->primitive.top_shadow_GC,
                sw->primitive.bottom_shadow_GC,
                sw->primitive.highlight_thickness,
                sw->primitive.highlight_thickness,
                sw->core.width - (2 * sw->primitive.highlight_thickness),
                sw->core.height - (2 * sw->primitive.highlight_thickness),
                sw->primitive.shadow_thickness,
                XmSHADOW_ETCHED_OUT);
```

Most Motif widgets inherit routines to draw and erase window decorations. However, if your widget takes the responsibility for its own window decorations, then your widget will draw the decorations with **XmeDrawHighlight** and **XmeDrawShadows** and clear the decorations with **XmeClearBorders**. Actually, the **XmeClearBorders** call can be used to clear out any rectangular area, not just window decorations.

If you want to draw shadows around any polygon (instead of just a rectangle), call **XmeDrawPolygonShadows** instead of **XmeDrawShadows**.

XmeDrawDiamond, **XmeDrawCircle**, and **XmeDrawIndicator** are particularly useful for creating toggle indicators.

8.4 Xme Resolution Independence Functions

Motif provides four Xme functions that govern resolution independence. (See Chapter 6 for information about resolution independence.)

The four functions shown in Table 8-3 are typically called from a synthetic resource table rather than from a widget method.

Table 8–3. Xme Resolution Independence Functions

Xme Function	What It Does:
XmeFromHorizontalPixels	Converts from pixels to real-world dimensions based on horizontal resolution of the screen
XmeFromVerticalPixels	Converts from pixels to real-world dimensions based on vertical resolution of the screen

XmeToHorizontalPixels	Converts from real-world dimensions to pixels based on horizontal resolution of the screen
XmeToVerticalPixels	Converts from real-world dimensions to pixels based on vertical resolution of the screen

8.5 Xme String Functions

Table 8-4 contains a list of Xme functions that are useful for manipulating strings.

Table 8–4. Xme String Functions

Xme Function	**What It Does:**
XmeGetDefaultRenderTable	Returns the default render table associated with a specified widget
XmeGetDirection	A compound string parse procedure (**XmParseProc**) to insert a direction component
XmeGetLocalizedString	Returns a localized version of the input string
XmeGetNextCharacter	A compound string parse procedure (**XmParseProc**) to insert a character
XmeNamesAreEqual	Compares two strings for equality
XmeRenderTableGetDefaultFont	Gets information on the default font associated with a specified render table

See Chapter 9 for complete details on using these functions.

8.6 Implementing Data Transfer

Motif provides widget writers with several Xme functions for implementing the Uniform Transfer Model (UTM). UTM is a mechanism for transferring data between and within widgets. UTM was introduced in Motif Release 2.0. (See Chapter 10 for details.)

Table 8-5 summarizes the Xme functions useful for transferring data between and within widgets.

Table 8–5. Xme Data Transfer Functions

Xme Function	**What It Does:**
XmeClipboardSink	Transfers data from the clipboard to a widget
XmeClipboardSource	Places data on the clipboard
XmeConvertMerge	Merges data converted during a transfer operation
XmeDragSource	Starts a drag and drop operation
XmeDropSink	Establishes a widget as a drop site
XmeGetLocaleAtom	Returns the encoding of the locale
XmeGetTextualDragIcon	Returns an icon widget symbolizing a textual drag operation in progress
XmeNamedSink	Transfers data from the named selection to a widget
XmeNamedSource	Takes ownership of a named selection
XmePrimarySink	Transfers data from the primary selection to a widget
XmePrimarySource	Takes ownership of the primary selection
XmeSecondarySink	Establishes a widget as the destination for secondary transfer
XmeSecondarySource	Takes ownership of the secondary selection
XmeSecondaryTransfer	Transfers data from the secondary selection to the destination widget

XmeStandardConvert	Converts selections to standard targets
XmeStandardTargets	Returns a list of standard Motif targets
XmeTransferAddDoneProc	Establishes a procedure to be called when data transfer is complete

In addition to the list of Xme routines shown in the previous table, Motif provides many Xm data transfer routines. The Xm routines are detailed in the *Motif 2.1—Programmer's Reference*.

8.7 Xme Geometry Functions

Motif provides the Xme calls shown in Table 8-6 to help widget writers manage geometry:

Table 8–6. Xme Geometry Functions

Xme Function	**What It Does:**
XmeConfigureObject	Changes a child's position, size, or border width
XmeRedisplayGadgets	Redisplays all the gadget children of a manager
XmeReplyToQueryGeometry	Handles standard geometry requests

See Chapter 12 for examples involving these functions.

8.8 Xme Focus/Traversal Functions

Motif provides the Xme calls shown in Table 8-7 to help widget writers handle focus changes.

Table 8–7. Xme Focus Functions

Xme Function	What It Does:
XmeAddFocusChangeCallback	Registers a callback for focus changes
XmeNavigChangeManaged	Helps a **change_managed** method establish the correct keyboard traversal policy
XmeRemoveFocusChangeCallback	Removes a focus change callback

8.9 Xme Miscellaneous Functions

Motif also provides the Xme functions shown in Table 8-8.

Table 8–8. Xme Miscellaneous Functions

Xme Function	What It Does:
XmeCreateClassDialog	Creates a dialog shell containing the specified widget
XmeGetDefaultPixel	Finds a color associated with the given widget
XmeGetNullCursor	Returns the null cursor associated with a given display
XmeGetPixmapData	Returns details about a cached pixmap
XmeMicroSleep	Suspends execution for a specified number of microseconds
XmeGetHomeDirName	Returns the pathname of the user’s home directory
XmeQueryBestCursorSize	Finds the best cursor size
XmeResolvePartOffsets	Allows writing of binary-compatible applications and widgets
XmeSetWMShellTitle	A compound string function that updates the window manager title
XmeVirtualToActualKeysyms	Finds the physical keysyms associated with a given virtual keysym
XmeWarning	Writes a warning message to the standard error stream

Chapter 9

Handling Textual Data

You will probably not find too many reasons to write your own text widget. After all, the standard Motif widget set already provides a variety of editable and noneditable text widgets. However, even if you do not write a text widget, you may end up writing a widget that contains a textual component. For example, perhaps the widget you are writing will display a caption or a title.

This chapter explains how to handle text in Motif widgets. In particular, this chapter will explore the following topics:

- The recommended resources for handling text in a Motif widget
- The *XmQTaccessTextual* trait
- The Xme functions that are useful for manipulating textual data

For demonstration purposes, the following subsections examine the **ExmString** widget.

9.1 Text Versus Compound Strings

Motif widgets should handle textual information as compound strings (**XmString**) rather than as simple character strings. Compound strings support multiple fonts, tab lists, and multiple colors; simple character strings do not.

9.2 Editable Versus Noneditable Text

When you are writing a widget that displays text, you must decide whether the text will be editable or noneditable. The standard Motif widget set provides two editable text widgets (**XmText** and **XmTextField**) and one noneditable text widget (**XmLabel**).

Editable text is much harder to code than noneditable text. One reason is because widgets with editable text must provide a much larger set of action methods than their noneditable counterparts. For example, a widget with editable text needs to provide actions that allow the user to navigate to different portions of the text. Editable text widgets usually provide actions that allow the user to cut, copy, and paste text. In addition, editable text widgets must support the range selection model. This model allows users to select a contiguous range of elements in a collection. Noneditable widgets need not support any selection model. Consequently, data transfer is much easier to implement in a noneditable text widget than in an editable text widget.

We do not want to discourage you from providing editable text. We merely want to warn you about its complexity.

The **ExmString** widget demonstrates noneditable text. Motif does not currently provide an Exm demonstration widget that demonstrates editable text.

9.3 Recommended Resources for Compound String Widgets

Any Motif widget displaying a compound string should support at least the following resources:

- A resource whose value holds the render table.

- A resource whose value holds the compound string itself.
- A resource whose value specifies the alignment of the text within the widget.
- A resource whose value determines the layout direction of the text. That is, this resource tells the widget whether the text should read from left to right or from right to left.

It is not a requirement that each widget define all of these resources. In fact, many of these resources will be provided by the widget's superclass. However, each widget displaying text needs to examine the values of these resources and display the text accordingly.

The following subsections examine how the **ExmString** demonstration widget uses these four resources.

9.3.1 The Render Table Resource

The render table resource holds the value of the render table that is used to display the text. All render table resources should have a representation type of **XmRRenderTable**.

Some Motif widgets call this resource **XmNrenderTable**. However, you may wish to provide a somewhat more specialized name for this resource. For example, if you are writing a widget named **MyGraph**, the render table resource could be called **MyNgraphRenderTable**.

In order to be consistent with other Motif widgets, the render table resource should set its default value through an **XtRCallProc** procedure rather than through **XmRImmediate** data. The **XtRCallProc** procedure for the **ExmString** widget is called **DefaultFont**, and it is defined as follows:

```
static void
DefaultFont (
        Widget w,
        int offset,
        XrmValue *value
)
{
 ExmStringWidgetClass wc = (ExmStringWidgetClass)XtClass(w);
 static XmRenderTable  f1;

 /* Find the default render table associated with the default
    render table type. */
   f1 = XmeGetDefaultRenderTable(w,
                     wc->string_class.default_render_table_type);

   value->addr = (XtPointer)&f1;
   value->size = sizeof(f1);
}
```

The **XmeGetDefaultRenderTable** routine finds the default render table. The second argument to **XmeGetDefaultRenderTable** specifies the default render table type. This field must contain one of the following three values:

- **XmLABEL_RENDER_TABLE**
- **XmBUTTON_RENDER_TABLE**
- **XmTEXT_RENDER_TABLE**

ExmString stores the default render table type in a field of the String class record. For the **ExmString** widget, the default render table type is to **XmLABEL_RENDER_TABLE**. However, subclasses of **ExmString** may override this value. For example, the **ExmCommandButton** widget sets its default render table type to **XmBUTTON_RENDER_TABLE**.

See the reference page for **XmeGetDefaultRenderTable** in Chapter 17 for more details.

9.3.2 The Compound String Resource

The compound string resource holds the compound string itself. Compound string resources should have a representation type of **XmRXmString**.

When a user accesses a compound string resource by calling **XtGetValue**, your widget must return a copy of the compound string resource's value. That is, your widget must make a copy of the string resource value, probably by calling **XmStringCopy**. Then, your widget must return a pointer to this copy rather than returning the original. Your widget should use Motif's **syn_resource** mechanism to handle this requirement.

For example, the **ExmString** widget provides a resource named **ExmNcompoundString** that holds the value of one compound string. Therefore, **ExmString** provides the following synthetic resource array:

```
static XmSyntheticResource syn_resources[] =
{
   {
        ExmNcompoundString,
        sizeof(XmString),
        XtOffsetOf(ExmStringRec, string.compound_string),
        GetValuesCompoundString,
        NULL
   }
};
```

The preceding declaration tells Motif to call the **GetValuesCompoundString** routine prior to returning the value of **ExmNcompoundString**. The **GetValuesCompoundString** routine is as follows:

```
GetValuesCompoundString(
        Widget w,
        int resource,   /* unused */
        XtArgVal *value)
{
 ExmStringWidget sw = (ExmStringWidget) w;
 XmString  string;

 /* All Motif widgets are responsible for making a copy of an XmString
    resource whenever an application accesses the resource through a call
    to XtGetValues. */
   string = XmStringCopy(sw->string.compound_string);

   *value = (XtArgVal) string;
}
```

9.3.3 The Alignment Resource

The alignment resource holds a value that symbolizes the justification of the text. The alignment resource should have a representation type of **XmRAlignment**. The valid values for this representation type are documented in the **XmLabel** reference page, under the description of the **XmNalignment** resource. (See the *Motif 2.1—Programmer's Reference* for **XmLabel(3x)**.)

The positioning of text within your widget should depend on a combination of the alignment resource and the layout direction resource. (See the next section for details.)

9.3.4 The Layout Direction Resource

Your widget should examine the layout direction resource provided by **XmPrimitive** or **XmManager**. We do not recommend creating your own layout direction resource. The **XmManager** and **XmPrimitive** widgets both provide an **XmNlayoutDirection** resource, and your widget should probably not override this value.

The combination of the layout direction resource and the alignment resource should determine the layout and alignment of the text. For example, suppose the value of the alignment resource is **XmALIGNMENT_BEGINNING** and the value of the layout direction resource is **XmLEFT_TO_RIGHT**. In this case, the left sides of the lines of text should be vertically aligned with the left edge of the widget window. On the other hand, if the layout direction value changes to **XmRIGHT_TO_LEFT**, then the right sides of the lines of text should be vertically aligned with the right edge of the widget window.

The **AlignmentDirection** method of the **ExmString** widget demonstrates how to determine where the text starts. The starting location of the text is particularly important for the resize method of **ExmString**. The text will be positioned in one of the following ways:

- The text will be centered.
- The text will start at the left side of the widget.
- The text will start at the right side of the widget.

The following is the code for the **AlignmentDirection** method:

```
static void
AlignmentDirection(
        Widget w)
{
 ExmStringWidget sw = (ExmStringWidget)w;

   if (sw->string.alignment == XmALIGNMENT_CENTER)
   /* The text will be centered. */
     sw->string.text_starts_here = ExmCENTER_STRING;

   else if (
       (XmDirectionMatch(sw->primitive.layout_direction, XmLEFT_TO_RIGHT) &&
        sw->string.alignment == XmALIGNMENT_BEGINNING)
                                ||
       (XmDirectionMatch(sw->primitive.layout_direction, XmRIGHT_TO_LEFT) &&
        sw->string.alignment == XmALIGNMENT_END))
   /* The string will start at the left side of the widget. */
     sw->string.text_starts_here = ExmSTART_STRING_LEFT_SIDE;

   else if (
       (XmDirectionMatch(sw->primitive.layout_direction, XmLEFT_TO_RIGHT) &&
        sw->string.alignment == XmALIGNMENT_END)
                                ||
       (XmDirectionMatch(sw->primitive.layout_direction, XmRIGHT_TO_LEFT) &&
        sw->string.alignment == XmALIGNMENT_BEGINNING))
   /* The string will start at the right side of the widget. */
     sw->string.text_starts_here = ExmSTART_STRING_RIGHT_SIDE;
}
```

9.4 Rendering the Compound String

Assuming that your widget provides code for the four resources described in the previous section, the **XmStringDraw** function is a good choice for rendering the text onto the screen. The **XmStringDraw** function takes 11 arguments, 4 of which are the values of the resources described in the previous section.

For example, the **ExmString** widget calls **XmStringDraw** inside its **DrawVisual** method. The call appears as follows:

```
/* If the compound string is not NULL and if there is enough space in the
   widget to draw at least a little portion of the compound string, then
   render the string with XmStringDraw. */
```

```
if (sw->string.compound_string &&
   (sw->simple.visual.width != 0) &&
   (sw->simple.visual.height != 0)) {
  XmStringDraw (XtDisplay(sw), XtWindow(sw),
                sw->string.render_table,
                sw->string.compound_string,
                wc->simple_class.select_gc(w),
                sw->simple.visual.x, sw->simple.visual.y,
                sw->simple.visual.width, sw->string.alignment,
                sw->primitive.layout_direction, NULL);
  ...
```

9.5 The XmQTaccessTextual Trait

Any widget that displays a primary block of text should install the *XmQTaccessTextual* trait. On the other hand, if the text in your widget is playing a supporting role, then you should not install the *XmQTaccessTextual* trait. For example, if the only text in your widget is a pixmap caption, then you would not install this trait on the widget. Furthermore, if the widget displays multiple blocks of text (such as the **XmList** widget does), then such a widget would not install this trait.

Table 9-1 suggests that this trait is extensively used inside the standard widget set.

Table 9–1. XmQTaccessTextual Access and Use in the Motif Toolkit

Widget	Installs	Accesses	Usage Notes
XmComboBox	No	Yes	Calls the **getValue** and **setValue** trait methods
XmLabel	Yes	No	Provides all three trait methods
XmLabelGadget	Yes	No	Provides all three trait methods
XmNotebook	No	Yes	Does not call a particular trait method; just examines its children to see if they have this trait installed
XmSelectionBox	No	Yes	Calls the **getValue** and **setValue** trait methods
XmSpinBox	No	Yes	Calls the **setValue** trait methods
XmText	Yes	No	Provides all three trait methods
XmTextField	Yes	No	Provides all three trait methods

The *XmQTaccessTextual* trait provides three methods. These trait methods are called by another widget, typically by the parent of a widget holding the *XmQTaccessTextual* widget. For example, the parent calls the **setValue** trait method to set the primary text block of its child. The parent calls the **getValue** trait method to find out what primary text block its child is displaying. Finally, the parent can call the **preferredFormat** trait method to find out what text format the child prefers to store its text in.

Different textual widgets hold their text in different formats. Currently, Motif supports three different text formats:

- *XmFORMAT_XmSTRING* symbolizes Motif compound string (**XmString**) format.
- **XmFORMAT_WCS** symbolizes a string in the wide-character string format defined by ANSI C.
- **XmFORMAT_MBYTE** format symbolizes a string in the multibyte string format defined by ANSI C.

The **ExmString** widget installs the *XmQTaccessTextual* trait and defines all three trait methods. The following subsections examine these trait methods.

9.5.1 The getValue Trait Method

The **getValue** trait method returns a copy of the text currently held by the **XmQTaccessTextual** widget. Typically, this value will be held inside a resource. For example, the **ExmString** widget holds its text value inside the **ExmNcompoundString** resource. Therefore, the current text value of the **ExmString** widget is easily obtained through the following call:

```
XtVaGetValues(w, ExmNcompoundString, &value, NULL);
```

The current text value is now stored in compound string format inside the *value* variable.

The widget that calls **getValue** must specify the string format in which it expects to receive the returned value. Therefore, most of the code inside the **getValue** trait method converts the text from its native format (in this case, *XmFORMAT_XmSTRING*) to one of the two other supported formats. Converting from **XmString** format to multibyte or wide-character string format requires a call to **XmStringUnparse**.

(See the code for **ExmString** for complete details.)

9.5.2 The setValue Trait Method

The **setValue** trait method changes the text and/or text format of a widget's primary text block. The **setValue** trait method takes three arguments:

- The widget
- The new text
- The format in which the caller is passing the new text string

If the new text has the same format as the widget's native format, no conversions are necessary. For example, the native format of the **ExmString** widget is **XmFORMAT_XmSTRING**. Therefore, if the caller passes the new text in the **XmFORMAT_XmSTRING** format, the **setValue** trait method merely needs to assign the text to the **ExmNcompoundString** resource. On the other hand, if the caller passes the new text in **XmFORMAT_MBYTE** (multibyte text) format, **setValue** must convert the new text from **XmString** format to **XmFORMAT_MBYTE** format.

Following is the complete **setValue** trait method of the **ExmString** widget:

```
StringSetValue(
     Widget w,
     XtPointer string,
     int format)
{
 Arg       args[1];
 XmString  temp;
 Boolean   freetemp;
 int       length;
 char     *str;
 wchar_t  *str2;

  /* The caller will pass a new value for ExmNcompoundString. This new
     value will be passed in the "string" argument. However, there is
     no guarantee that the input "string" will be passed in XmString format.
     If the input "string" is passed in WCS or MULTIBYTE format, then we
     must convert the "string" into XmString format. Once the "string"
     is in XmString format, we can use it as the new value of
     ExmNcompoundString. */
    switch (format)    {
        case XmFORMAT_XmSTRING:
                   temp = (XmString) string;
                   freetemp = False;
                   break;

        case XmFORMAT_WCS:
                   str2 = (wchar_t *) string;
                 /* How long is str2? */
                   length = 0;
                   while (str2[length] != 0)
                      length++;
                 /* malloc enough space to hold str */
                   str = (char*) XtMalloc(MB_CUR_MAX * (length+1));
                   wcstombs(str, str2, MB_CUR_MAX * (length+1));
                   XtFree((char*)string);
                   string = str;

                 /* Falling through to XmFORMAT_MBYTE */

        case XmFORMAT_MBYTE:
                   temp = XmStringCreateLocalized(string);
                   freetemp = True;
                   break;

        default:
                   XmeWarning((Widget)w, UNSUPPORTED_FORMAT);
                   return;
    }

 /* Assign the new string to ExmNcompoundString. */
    XtSetArg(args[0], ExmNcompoundString, temp);
    XtSetValues(w, args, 1);

    if (freetemp)
      XmStringFree(temp);
}
```

9.5.3 The preferredFormat Trait Method

Every textual widget has some native format in which it stores the text itself. The **preferredFormat** trait method returns the name of this native format. For example, Table 9-2 shows the native format of the **XmQTaccessTextual** widgets in the standard Motif widget set.

Table 9–2. Native Format of XmQTaccessTextual Widgets

Widget	**Native Format**
XmLabel	**XmFORMAT_XmSTRING**
XmLabelGadget	**XmFORMAT_XmSTRING**
XmText	**XmFORMAT_MBYTE**
XmTextField	**XmFORMAT_MBYTE**

The native format of **ExmString** is **XmFORMAT_XmSTRING**. Therefore, the entire **preferredFormat** trait method is simply as follows:

```
static int
StringPreferredFormat(
     Widget w)
{
   return(XmFORMAT_XmSTRING);
}
```

If the calling widget knows the textual widget's native format prior to calling **getValue** or **setValue**, then needless conversions can be avoided.

Chapter 10

Data Transfer Between Widgets with UTM

Motif provides several mechanisms for transferring data from one widget to another. For example, Motif users can use any of the following mechanisms to move text between any two **XmText** widgets:

- Primary selection
- Secondary selection
- Clipboard
- Drag and drop

From a user's point of view, each of these data transfer methods is rather different; each one is accomplished by pressing different combinations of keys or mouse buttons. From a Motif widget writer's point of view, however, there are a number of similarities in implementing them. These similarities in data transfer implementation fit under the umbrella term "Uniform Transfer Model" (UTM).

UTM is not a user-accessible data transfer mechanism. In other words, there is no magic sequence of keys that will allow a user to transfer data through UTM. Rather UTM is simply a way for a widget writer (and possibly, an application programmer) to implement one of the four data transfer mechanisms.

UTM gives the widget writer the following arsenal of software for implementing data transfer:

- A group of Xme functions
- A group of Xm functions
- The *XmQTtransfer* trait
- Two new callback structures

One way to think of UTM is that it is a convenient front end (or wrapper) to existing data transfer mechanisms. You could, for example, implement primary transfer with existing Xt and Motif calls. However, you would probably find it far easier to implement primary transfer through UTM. You will almost certainly find it easier to implement data transfer through UTM when your widget supports more than one transfer mechanism.

UTM is completely compatible with the ICCCM (the *Inter-Client Communications Conventions Manual*). In fact, to get the most out of this chapter, you should already be comfortable with the material in the ICCCM. We do, however, provide the following tutorial for those whose knowledge may need a little refreshing.

10.1 A Data Transfer Tutorial

Transferring data between two widgets is analogous to transferring data between two computers. In both cases, data transfer requires both a channel and a communications protocol.

Consider how two computers transfer a file. Before the transfer, one computer owns the file to be transferred and the other computer (the receiver) does not. In order for the transfer to take place, some sort of channel (probably a modem or a cable) must connect the two machines. However, having a physical link between the two machines is not enough; the two machines must also agree on the rules or protocols governing the file transfer. For example, the computer that owns the file needs some way to tell the destination computer that it is ready to send the file. The destination computer needs to signal back to the owner computer when it is ready to receive. At the end of the file transfer, the destination computer needs to acknowledge receipt of the file.

Now let’s consider how two widgets transfer data. Before the data transfer, one widget owns the data to be transferred and the other widget (the destination widget) does not. In order for the transfer to take place, some sort of channel must connect the two widgets. The X Windows server acts as this virtual channel. However, having a channel is not enough; the two widgets must also share the same data transfer protocol. This protocol is defined by the ICCCM; the recommended Motif implementation of the ICCCM is called UTM.

The ICCCM protocol defines the kinds of messages passed between the owner widget and the destination widget. The two widgets can belong to the same client or to two different clients.

10.1.1 Problems Solved by the Protocol

Many newcomers wonder why the ICCCM protocol has to be so complicated. One answer is that each widget is a self-contained object that knows how to take care of itself but ignores other widgets. Therefore, widgets monitor their own events only and ignore the events of other widgets.

For example, suppose a user wants to copy some data from one widget (the source) to another (the destination). To do this, the user must first select the data that is to be copied. The user then moves the pointer to the destination widget and indicates where the data should be transferred. Since the destination widget does not receive the source widget’s events, the destination widget does not have the following information:

- The data the user selected for copying
- What kind of data (such as text, a pixmap, a color) the user has selected
- The format data is in or how much data there actually is

The ICCCM protocol solves these problems by forcing all widgets to use atoms to describe the data to be transferred.

10.1.2 Atoms

The ICCCM requires that the owner and destination use atoms to solve these categorization needs. In brief, an atom is a number that symbolizes a particular data category or a particular data transfer selection. For example, the ICCCM specifies that an atom called *PIXMAP* be used to symbolize the transfer of a pixmap. As another example, the ICCCM specifies that an atom called *CLIPBOARD* by used to symbolize a clipboard selection.

Atoms like *PIXMAP* that specify a data category are called target atoms or, more simply, targets. Atoms like *CLIPBOARD* that specify a kind of transfer mechanism are called selection atoms.

10.1.3 Target Atoms

Widgets can display many kinds of information. Most of the displayed information (and some of the nondisplayed information) should be transferrable. The widget represents each of these different kinds of transferrable information as a target atom.

For example, consider the **XmLabel** widget. As you might expect, **XmLabel** allows users to make copies of the displayed text or pixmap. What is somewhat surprising is that **XmLabel** can also transfer other kinds of information. For example, **XmLabel** can transfer information about its foreground and background colors. In fact, the relatively simple **XmLabel** widget can transfer over a dozen different targets.

10.1.4 Selection Atoms

Each selection is represented by an atom. The display contains only one selection of each type. It is owned by a client or by no one and, if owned, is attached to a window of the owning client. Any client may assert or remove ownership of a selection.

The data represented by the selection is internal to the client that owns the selection.

Motif uses the following selection atoms:

- *PRIMARY*, which identifies the principal selection. This selection atom is defined by the ICCCM.
- *SECONDARY*, which identifies a means of exchanging data without disturbing the primary selection. This selection atom is defined by the ICCCM.
- *CLIPBOARD*, which is the selection often used to cut or copy data from one client and paste it into another. A client transfers data to the clipboard by asserting ownership of this selection. A client transfers data from the clipboard by requesting conversion of the selection. A separate client may also represent the clipboard. This client can notice when it loses the selection (because another client wants to transfer data to the clipboard), then request a conversion of the selection and finally reassert ownership. This selection atom is defined by the ICCCM.
- *_MOTIF_DROP*, which is the selection used in a drag and drop operation. This selection atom is specific to Motif; that is, the ICCCM does not define the *_MOTIF_DROP* selection atom.

10.1.5 Defining Atoms

The **Atom** data type is used to declare an atom variable. Use the Xlib function **XInternAtom** to assign an atom value to your atom variable. For example, the following code creates an **Atom** variable named *TEXT* and associates the correct atom value with the *TEXT* variable.

```
Atom TEXT = XInternAtom(XtDisplay(w), "TEXT", False);
```

The second argument to **XInternAtom** is an atom string name. You can specify an atom string name from any of the following four categories:

- An atom string name defined by the ICCCM; for example, *TARGETS*.
- A Motif atom string name that is a redefinition of an ICCCM atom string name. Table 10-1 lists all these atom string names along with their ICCCM equivalents. You can assume that all of the Motif atom string names have the same meaning as their ICCCM equivalents.
- A Motif atom string name that has no ICCCM equivalent.
- An atom string name that you make up yourself.

We recommend that you avoid making up new atom string names whenever possible. If you do make up your own, the atom string names you pick should follow the naming conventions documented in the ICCCM.

You can specify an atom string name as a constant having the **XmS** prefix; for example:

```
Atom TARGETS = XInternAtom(XtDisplay(w), XmSTARGETS, False);
```

or you can specify an atom string name as a string without the **XmS** prefix; for example, the following line of code produces the same results as the previous line of code:

```
Atom TARGETS = XInternAtom(XtDisplay(w), "TARGETS", False);
```

Table 10–1. Motif Atoms That Are Equivalent to ICCCM Atoms

Atom String Name	**ICCCM Equivalent**
XmSCLIENT_WINDOW	*CLIENT_WINDOW*
XmSCLIPBOARD	*CLIPBOARD*
XmSCOMPOUND_TEXT	*COMPOUND_TEXT*
XmSDELETE	*DELETE*
XmSFILE	*FILE*
XmSFILE_NAME	*FILE_NAME*
XmSINCR	*INCR*
XmSINSERT_PROPERTY	*INSERT_PROPERTY*
XmSINSERT_SELECTION	*INSERT_SELECTION*
XmSLENGTH	*LENGTH*
XmSLINK_SELECTION	*LINK_SELECTION*
XmSMULTIPLE	*MULTIPLE*
XmSNULL	*NULL*
XmSTARGETS	*TARGETS*
XmSTEXT	*TEXT*

XmSTIMESTAMP	*TIMESTAMP*
XmSWM_STATE	WM_STATE

As we mentioned earlier, Motif also provides some atom string names that have no ICCCM equivalent. Motif atom string names that have no ICCCM equivalent have a prefix of *_MOTIF* are as follows:

- **XmS_MOTIF_CLIPBOARD_TARGETS**
- **XmS_MOTIF_COMPOUND_STRING**
- **XmS_MOTIF_DEFERRED_CLIPBOARD_TARGETS**
- **XmS_MOTIF_DESTINATION**
- **XmS_MOTIF_DROP**
- **XmS_MOTIF_ENCODING_REGISTRY**
- **XmS_MOTIF_EXPORT_TARGETS**
- **XmS_MOTIF_LOSE_SELECTION**
- **XmS_MOTIF_RENDER_TABLE**
- **XmS_MOTIF_SNAPSHOT**

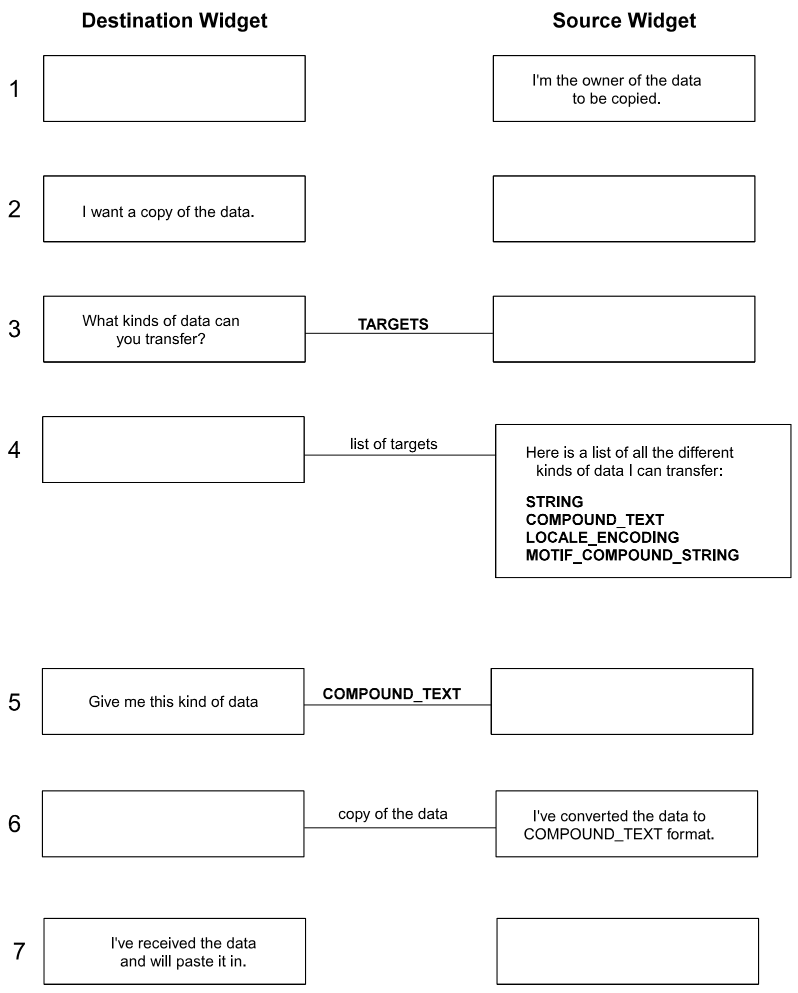
This chapter explains all of these unique Motif atom string names.

10.1.6 A Typical Data Transfer Sequence

The data transfer protocol defines a kind of dialog between the source widget and the destination widget. This dialog is somewhat different for each data transfer mechanism. For example, the dialog of a clipboard transfer is more complicated than that of a primary transfer. Nevertheless, we can describe a kind of idealized dialog that is general enough to be reasonably accurate for all four data transfer mechanisms.

This idealized data transfer sequence consists of the seven steps shown in Figure 10-1.

Figure 10–1. The Data Transfer Dialog


Destination Widget
Source Widget
1
I'm the owner of the data to be copied.
2
I want a copy of the data.
3
What kinds of data can you transfer?
TARGETS
4
list of targets
Here is a list of all the different kinds of data I can transfer:
STRING
COMPOUND_TEXT
LOCALE_ENCODING
MOTIF_COMPOUND_STRING
5
Give me this kind of data
COMPOUND_TEXT
6
copy of the data
I've converted the data to COMPOUND_TEXT format.
7
I've received the data and will paste it in.

The user starts a data transfer (Step 1) by specifying the source widget of the data transfer. The user typically specifies the source widget by selecting some portion of a widget; for example, a line of displayed text. The user makes a selection through some keyboard or mouse event. For example, a user can select a line in an **XmText** widget by positioning the cursor anywhere on the line and then clicking the left mouse button three times. In some cases, Step 1 actually consists of two parts. In the first part, the user selects the data to be transferred. In the second part, the user enters some other event to specify the data transfer mechanism. For example, the user might hold down the middle mouse button to start a drag operation.

The user also indicates the destination widget (Step 2) of the data transfer. The user does this by changing focus to the destination widget and then entering some event. For example, the user specifies the destination widget of a drag and drop operation by moving the cursor to the destination widget and then releasing the middle mouse button.

After completing Steps 1 and 2, the user's role in data transfer is over. The remaining steps must all be implemented by the widget.

Don't forget that, at this stage, the destination widget has no idea what kind of data has been selected in the source widget. Therefore, at Step 3, the destination widget has to ask the source widget for a list of all the targets that the source widget can transfer. The destination widget does this by sending the source widget a message asking for its *TARGETS* list.

The source widget responds to the request for *TARGETS* (Step 4) by returning a list of all the atoms that it knows how to transfer. For example, in Figure 10-1, the source widget claims to know how to transfer the following targets:

- *STRING*
- *COMPOUND_TEXT*
- *LOCALE_ENCODING*
- *MOTIF_COMPOUND_STRING*

In practice, the returned list of *TARGETS* will be considerably longer than in this example.

The source widget does not place the atoms in the returned *TARGETS* list in any particular order. For example, the first atom in the *TARGETS* list is not necessarily the atom that the source widget wants most to transfer.

In Step 5, the destination widget examines the list of target atoms returned by the source widget. From this list, the destination widget chooses the atom that most closely meets its needs. In Figure 10-1, the destination widget selected *COMPOUND_TEXT*.

In Step 6, the source widget converts the data that the user has selected into the data representation that the destination widget has requested. For example, in Figure 10-1, the source widget has to convert the selected data to *COMPOUND_TEXT*. In some cases, the source widget does not have to do much work to convert the data. For example, perhaps the source widget stores text as *COMPOUND_TEXT* anyway. In other cases, the source widget might have to fight a considerable battle to convert the data into the proper representation. For example, perhaps the source widget stores text in **XmString** format. In this case, the source widget will have to convert the string from **XmString** format to *COMPOUND_TEXT* format.

A surprisingly high percentage of a data transfer widget is devoted to data conversion. When the source widget is finished converting the data, the source widget sends the converted data to the destination widget.

In Step 7, the destination widget pastes the received data into the appropriate portion of the widget.

You may be wondering what happens if the destination widget wants more than one kind of data from the source widget. For example, suppose that the destination widget wants the source widget to transfer both a pixmap and a colormap. There are two possible ways to handle this. One possibility is that Steps 5, 6, and 7 will be repeated. Another possibility is that, in Step 5, the destination widget can request a multiple data transfer, and, in Step 6, the source widget can honor that request by passing several different kinds of data back at the same time.

10.2 Overview of Implementing UTM

Suppose you want the widget you are writing to support data transfer. To implement data transfer by using UTM, you must do the following:

1. Decide which data transfer mechanism(s) your widget will support. For example, perhaps your widget will support both the primary and clipboard transfers. Supply translations appropriate for these data transfer mechanisms.
2. Write action methods for these translations; have these action methods call the appropriate UTM Xme set-up routines.
3. Install the *XmQTtransfer* trait on your widget.

4. Provide a **convertProc** trait method if your widget is to be a source for data transfer. Source widgets must support the **XmNconvertCallback** resource.
5. Provide a **destinationProc** trait method if your widget is to be a destination for data transfer. Destination widgets must support the **XmNdestinationCallback** resource.
6. Provide a transfer procedure if your widget is to be a destination for data transfer.

This section details each of these steps. Throughout this section and the remainder of the chapter, we will examine how the **ExmStringTransfer** widget implements UTM. The **ExmStringTransfer** demonstration widget is a subclass of **ExmString**. The only feature that **ExmStringTransfer** layers on top of **ExmString** is its support of the following three data transfer mechanisms:

- Primary
- Clipboard
- Drag and drop

ExmStringTransfer gives us an excellent opportunity to isolate the code that implements UTM.

ExmStringTransfer has two important limitations. First, **ExmStringTransfer** does not let users transfer portions of the displayed text. In other words, a user may only transfer the entire contents of the **ExmStringTransfer** widget. By contrast, a more powerful widget such as **XmText** lets users copy a selected word, line, paragraph, and so forth.

Second, **ExmStringTransfer** only knows how to do copy operations; it cannot do a move operation. In fact, when **ExmStringTransfer** is a destination, the incoming string will overwrite any existing displayed string.

The sample application stored in directory **demos/programs/Exm/app_in_c** contains two **ExmStringTransfer** widgets. You might want to build this application and experiment with it. For example, try to copy data from one **ExmStringTransfer** widget to another. Then, try to copy text from a widget in another application to one of the **ExmStringTransfer** widgets.

10.2.1 Supply Appropriate Translations

You must decide what data transfer mechanisms your widget will support. Ideally, your widget should support as many data transfer mechanisms as possible. Writing a widget that supports multiple data transfer mechanisms is only slightly more time consuming than writing a widget that supports only one data transfer mechanism. Most of the data transfer code is independent of a particular form of data transfer.

You might be writing a read-only widget. In other words, the user is not allowed to transfer data into the widget. For these widgets, you need only supply the source side of data transfer; you can omit the destination side.

One way to help you decide what data transfer mechanisms your widget should support is to examine how the widgets in the Motif toolkit support data transfer. (By the way, as of Motif Release 2.0, all the widgets in the Motif toolkit use UTM to implement data transfer.) Table 10-2 summarizes the data transfer methods supported in the standard widget set. When an entry says "Source only" or "Drag only," it means that a user can copy data in this widget but cannot cut data out of it or paste data into it. For example, a user can copy text from an **XmLabel** to an **XmText** widget but cannot copy text from an **XmText** to an **XmLabel** widget.

Table 10–2. Data Transfer 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)

Users trigger data transfer by pressing keys or mouse buttons. Your widget must supply data transfer translations compatible with the *CDE 2.1/Motif 2.1—Style Guide and Glossary*.

Widgets typically provide two different kinds of translations to support data transfer.

The first kind of data transfer translation allows the user to select the data to be transferred. For example, the **XmText** widget provides several translations that allow a user to select different subsets of the displayed text. These subsets range from one character to the entire text. By contrast, **XmLabel** does not allow a user to select a portion of its displayed text; the user can only select the entire displayed text.

In addition to providing selection translations, your widget must also supply translations that let the user initiate and complete the data transfer. For example, if a widget is to support drag and drop, the widget must supply a translation that allows the user to mark the start of the drag operation. (Typically, the user starts a drag operation with a Button2 or Button2Press event.) The four different kinds of data transfer each require different translations.

The easiest way to supply appropriate Motif translations for your widget is to copy the data translations of the standard Motif widget that is most similar to your own. Translations for standard Motif widgets are documented in the *Motif 2.1—Programmer's Reference*.

10.2.2 Write Actions for These Translations

Your widget must provide actions that respond to the user's data transfer request. These action methods have to call the appropriate UTM Xme set-up routine. The routine your action calls depends on the type of data transfer, and on whether the action method is for the source side of the transfer or the destination side of the transfer.

If you are writing an action method that represents the source side of a data transfer, the action method must call one of the following routines:

- **XmePrimarySource**
- **XmeSecondarySource**
- **XmeClipboardSource**
- **XmeDragSource**

For example, suppose a widget is to serve as a source for a clipboard transfer. This widget must respond to a selection translation by calling **XmeClipboardSource**.

Broadly speaking, each of the preceding Xme routines makes the following sequence of calls:

1. It calls the application's **XmNconvertCallback** procedures, if any.
2. It calls the widgets's **convertProc** trait method.

Actions in destination widgets must call one of the following routines:

- **XmePrimarySink**
- **XmeSecondarySink**
- **XmeClipboardSink**
- *XmeDragSink*

For example, suppose a widget is to serve as a destination for a clipboard transfer. When the user triggers a clipboard paste, the clipboard paste action must call **XmeClipboardSink**.

Broadly speaking, each of the preceding four routines trigger the following sequence of calls:

1. It calls the widget's destination pre-hook procedure, if any.
2. It calls the application's **XmNdestinationCallback** procedures, if any.
3. It calls the widget's **destinationProc** trait method.

The procedures and callbacks are described later in this section.

10.2.3 Install The XmQTtransfer Trait

Your widget must install the *XmQTtransfer* trait in order to support UTM data transfers.

Your widget can install *XmQTtransfer* as it would install any trait; namely by calling **XmeTraitSet**. Typically, the call is made from a **class_part_initialize** method.

Widgets capable of acting as a source for data transfer must provide a **convertProc** trait method. Widgets capable of acting as a destination for data transfer must provide at least a **destinationProc** trait method. Frequently, a widget will be both a source and a destination. In this case, your widget must supply both trait methods. Destination widgets may optionally also provide a **destinationPreHookProc** trait method.

For example, the **ExmStringTransfer** widget installs the *XmQTtransfer* trait as part of its **ClassPartInitialize** method. The code that does the installation is as follows:

```
ClassPartInitialize(WidgetClass widgetclass)
{
  XmeTraitSet((XtPointer) widgetclass, XmQTtransfer, (XtPointer)
              &StringTrT);
}
```

where the **StringTrT** variable is declared earlier in the **StringTrans.c** file as follows:

```
static XmConstXmTransferTraitRec StringTrT = {
  0,                                         /* version */
  (XmConvertCallbackProc) ConvertProc,         /* convertProc */
  (XmDestinationCallbackProc) DestinationProc, /* destinationProc */
  NULL,                                        /*no destinationPreHookProc */
};
```

10.2.4 Supply a convertProc Trait Method

Source widgets must supply a **convertProc** (convert procedure) trait method. The **convertProc** method is one of the trait methods of the *XmQTtransfer* trait. (See Chapter 18 for complete syntactic details on the *XmQTtransfer* trait.) UTM automatically calls **convertProc** whenever an object (typically, a destination widget) requests a conversion from the widget.

The **convertProc** has the same prototype as any Intrinsics callback. Therefore, the third argument to **convertProc** is *call_data*. UTM will pass an **XmConvertCallbackStruct** to the *call_data* argument. The **XmConvertCallbackStruct** contains all the raw data required for the conversion. For example, the *targets* member of this structure contains the name of the target atom that the destination procedure wants converted.

Your **convertProc** must respond to the following two general kinds of requests:

- A request for the list of targets supported by your widget. For example, the requestor may ask your **convertProc** for *TARGETS*.
- A request to convert the selection to a specific target. For example, the requestor may ask your **convertProc** to convert the selection to a *COMPOUND_TEXT* format.

Later on in this chapter, we will discuss the specific targets your widget should support.

10.2.5 Supply a destinationProc Trait Method

Your widget must supply a **destinationProc** (destination procedure) trait method in order to serve as a destination widget in a data transfer. The **destinationProc** trait method is one of the trait methods of the *XmQTtransfer* trait. (See Chapter 18 for complete syntactic details on the *XmQTtransfer* trait.) UTM automatically calls **destinationProc** whenever a user requests that data be transferred to the destination widget. The **destinationProc** trait method is responsible for requesting data from the source widget and for pasting this data into the destination widget. However, the **destinationProc** trait method will ask a transfer procedure to take over some of these responsibilities.

UTM calls **destinationProc** after calling any **XmNdestinationCallback** procedures defined by the application. The toolkit passes **destinationProc** an **XmDestinationCallbackStruct** structure. This structure contains all the fields necessary to make requests to the source widget.

The **destinationProc** trait method should typically call *XmTransferValue* to request a conversion from the source widget *XmTransferValue* automatically calls the source widget's **convertProc**. When the **convertProc** finishes, *XmTransferValue* typically calls the destination widget's transfer procedure. The name of the transfer procedure is identified by the third argument to the *XmTransferValue* call.

The **destinationProc** trait method can terminate a data transfer by calling *XmTransferDone*.

XmTransferValue, together with *XmTransferDone*, form the UTM replacements for the following older calls:

- **XtGetSelectionValue**
- **XtSetSelectionValue**
- **XmClipboardStartRetrieve**
- **XmClipboardRetrieve**
- **XmClipboardEndRetrieve**
- **XmDropTransferStart**
- **XmDropTransferAdd**

10.2.6 Supply a Transfer Procedure

If you are writing a destination widget, you will probably have to write one or more transfer procedures. UTM calls a transfer procedure when the **convertProc** trait method finishes the conversion. UTM passes the transfer procedure a pointer to an **XmSelectionCallbackStruct**. (See the **XmTransferDone**(3) reference page of the

Motif 2.1—Programmer's Reference for details on this structure.) The *value* member of **XmSelectionCallbackStruct** holds the selection converted to the target that the destination widget has requested. The target procedure will take that converted target and paste it into the destination widget.

10.3 UTM and the Application

So far, we have focused on the work that widgets do to implement a UTM data transfer. However, it is possible that a Motif application will implement some or even all of a UTM data transfer. UTM allows application programmers to register two kinds of callbacks:

- **XmNconvertCallback** procedures to be registered on widgets that can be the source of a UTM data transfer.
- **XmNdestinationCallback** procedures to be registered on widgets that can be the destination of a UTM data transfer.

In addition, an application's **XmNdestinationCallback** procedure can provide its own transfer procedure.

10.3.1 Conversion Routines

UTM automatically calls any **XmNconvertCallback** procedures immediately prior to calling the **convertProc** trait method of the source widget. UTM automatically passes an **XmConvertCallbackStruct** to the application's **XmNconvertCallback** procedures. These procedures are free to modify the fields of this structure. UTM will pass this structure, modifications and all, to your source widget's **convertProc** trait method.

If the application does not provide any **XmNconvertCallback** procedures, then UTM simply calls the **convertProc** trait method. If the application does provide **XmNconvertCallback** procedures, then UTM conditionally calls **convertProc** depending on the value in the *status* member of the **XmConvertCallbackStruct**. If the *status* member holds **XmCONVERT_DONE** or **XmCONVERT_REFUSE**, then UTM will not call the source widget's **convertProc** trait method. If the *status* member holds some other value, then UTM will call the source widget's **convertProc** trait method.

Throughout this chapter, we use the term "conversion routines" to refer to a combination of the **XmNconvertCallback** procedures of the application and the **convertProc** trait method of the source widget.

10.3.2 Destination Routines

UTM automatically calls any **XmNdestinationCallback** procedures immediately prior to calling the **destinationProc** trait method.

Throughout this chapter, we use the term "destination routines" to refer to a combination of the **XmNdestinationCallback** procedures of the application and the **destinationProc** trait method of the destination widget.

10.3.3 Application Interaction

Broadly speaking, an application's UTM callbacks can interact with your widget's UTM trait methods in any of the following three ways:

1. The application might not provide any UTM callbacks.
2. The application might provide UTM callbacks that do all the UTM transfer work. In this case, UTM will not call your widget's UTM trait methods.
3. The application might provide UTM callbacks that do some, but not all, of the UTM transfer work.

In case 1, your widget must be prepared to do all the conversion and transfer work itself. Preparing for this case simply means that your widget's UTM trait methods must be self-sufficient. In other words, your widget must handle all the targets that your widget documentation says it handles.

In case 2, your widget's UTM trait methods never get called.

In case 3, your widget's UTM trait methods must react appropriately to data passed by the application's UTM callback procedures. Consider the following situation. Suppose you write a widget that displays text. Being a good widget writer, you code your widget so that its UTM trait methods support all the standard textual targets. However, suppose an application programmer decides to supplement your widget’s list of targets by writing a UTM application callback that supports an additional textual target, **_FORMATTED_TEXT**. Therefore, the following things must happen:

- When the destination widget asks for a list of *TARGETS*, the source widget must somehow provide a list of all of its original targets plus the **_FORMATTED_TEXT** target.
- When the destination widget asks for a conversion to **_FORMATTED_TEXT**, the **XmNconvertCallback** procedures of the application must do the conversion.
- When the destination widget asks for a conversion to any other target, the **convertProc** trait method must do the conversion.

Of the preceding three bulleted items, only the first requires extra coding by the widget writer. When an application adds a new target, the application must set the *status* member of the **XmConvertCallbackStruct** to **XmCONVERT_MERGE**. (Remember that UTM automatically passes an **XmConvertCallbackStruct** to your widget’s **convertProc** trait method.) Therefore, your widget’s **convertProc** trait method should always examine the *status* member to see if it holds **XmCONVERT_MERGE**. If it does, **convertProc** should call **XmeConvertMerge**. The following code illustrates how this is done:

```
if (cs -> status == XmCONVERT_MERGE) {
  XmeConvertMerge(value, type, format, length, cs);
  XtFree((char *) value);
}
```

The **XmeConvertMerge** routine merges two groups of list-like targets. In this case, **XmeConvertMerge** merges the sole target from the application **_FORMATTED_TEXT** with the list of targets already supported by your widget.

10.4 Primary Transfer Details

This section details primary transfers.

10.4.1 Primary Transfer: Step-by-Step

A primary transfer involves interaction among a user, an application, a source widget, and a destination widget. Following are the steps involved in a typical primary transfer:

1. The user selects some portion of the source widget. The source widget probably highlights the selection.
2. The source widget calls **XmePrimarySource**. **XmePrimarySource** establishes an internal function (we will call it **FunctionP**) that UTM will call later during the transfer.
3. The user moves the cursor to the destination widget and presses some mouse button or key sequence to initiate a primary paste.
4. The Intrinsics call the action method of the destination widget that is associated with the primary paste event. This action method calls **XmePrimarySink**.
5. **XmePrimarySink** calls the destination widget's **destinationPreHookProc** trait method, if it exists.
6. **XmePrimarySink** calls any **XmNdestinationCallback** procedures that the application has attached to the destination widget.
7. **XmePrimarySink** conditionally calls the **destinationProc** trait method of the destination widget. We will use the term "destination routines" to refer to the combination of the **XmNdestinationCallback** application procedures and the **destinationProc** trait method. Typically, one of the destination routines will call *XmTransferValue* and request a list of the source widget's *TARGETS*.
8. UTM calls **FunctionP**, which in turn calls the conversion routines of the source widget. The conversion routines are responsible for returning a list of the convertible targets.
9. UTM returns control to the transfer procedure named by *XmTransferValue*. The transfer procedure examines the list of returned targets and picks the most appropriate one. The transfer procedure then calls *XmTransferValue* and asks for the chosen target. For this example, let us assume that the **destinationProc** picked the *COMPOUND_TEXT* target.
10. UTM calls **FunctionP**, which in turn calls the conversion routines again. This time, **FunctionP** asks the conversion routines to convert the selection to *COMPOUND_TEXT* format. The conversion routine is responsible for setting the appropriate fields in the **XmConvertCallbackStruct** For example, the conversion routine should place the converted text into the *value* member of the

XmConvertCallbackStruct. If a conversion routine can do the conversion, it sets the *status* member of the **XmConvertCallbackStruct** to **XmCONVERT_DONE**.

11. UTM returns control to the transfer procedure of the destination widget. Assuming all went well, the converted text will be stored in the *value* member of the **XmSelectionCallbackStruct**. The transfer procedure need only paste the converted text into the appropriate place in the destination widget.

10.4.2 Sample Translations for a Primary Transfer

The *ExmStringTransfer* demonstration widget supports primary transfer. The following translations are relevant to the primary transfer:

```
<Btn1Down>:  ExmStringTransferMoveFocus()ExmStringTransferCopyPrimary()\n\
<Btn2Down>:  ExmStringTransferProcessDrag()\n\
```

A user starts a primary copy by pressing Btn1Down. However, the Btn2Down translation is far more puzzling because Btn2Down could mean either a primary paste or the start of a drag operation. *ExmStringTransferProcessDrag* solves this puzzle by determining whether or not *ExmStringTransfer* currently owns the primary selection. If it does, *ExmStringTransferProcessDrag* calls the *ExmStringTransferPastePrimary* routine.

10.4.3 Sample Copy Primary Action

The *ExmStringTransfer* widget contains the following relatively simple copy primary routine:

```
static void
ExmStringTransferCopyPrimary(
     Widget w,
     XEvent *event,
     String *params,
     Cardinal *num_params)
{
 Time time;
 ExmStringTransfer stw = (ExmStringTransfer) w;
  /* First we must obtain a timestamp. This is required for
```

```
     ICCCM compliance. XtLastTimestampProcessed holds the timestamp
     from the last event the Intrinsics saw with a timestamp. */
  time = XtLastTimestampProcessed(XtDisplay(w));

  /* Own the primary selection. Indicate this to the user by
     reversing the foreground and background in the text rendition
     (eventually) */
  stw->string_transfer.own_primary = True;

  /* Once we call XmePrimarySource,  the widget's transfer trait
     convert method will get called if a destination wishes to
     obtain the PRIMARY selection. */
  XmePrimarySource(w, time);

  /* Update the widget to show the new state */
  stw->core.widget_class->core_class.expose(w, NULL, NULL);
}
```

In the preceding code, **stw->string_transfer.own_primary** is a **Boolean** field. *ExmStringTransfer* sets this field to **True** when the widget holds the primary selection and **False** when the widget relinquishes the primary selection.

10.4.4 Sample Paste Primary Action

The most important part of the paste primary routine of *ExmStringTransfer* is its call to **XmePrimarySink**. Following is the complete *ExmStringTransferPastePrimary* routine:

```
static void
ExmStringTransferPastePrimary(
     Widget w,
     XEvent *event,
     String *params,
     Cardinal *num_params)
{
  Time time;
  unsigned int op;
  /* First we must obtain a timestamp. This is required for
     ICCCM compliant ownership of a selection. We use
     XtLastTimestampProcessed which holds the timestamp
     from the last event the Intrinsics saw with a timestamp. */
  time = XtLastTimestampProcessed(XtDisplay(w));

  /* We determine the right operation to perform by looking
     at the modifiers present */
  if (event -> xbutton.state & ShiftMask) {
    if (event -> xbutton.state & ControlMask)
      op = XmLINK;
    else
      op = XmMOVE;
  } else
    op = XmCOPY;

  /* Calling XmePrimarySink will start the process of requesting
     the PRIMARY selection to be pasted using the transfer trait
     destination callback for this widget class */
  XmePrimarySink(w, op, NULL, time);
}
```

The second argument to **XmePrimarySink** (called *op* in the preceding code) describes the kind of operation that the user has requested. UTM will copy the value of *op* into the *operation* member of the **XmDestinationCallbackStruct**.

10.5 Clipboard Transfer Details

This section explains some of the finer points of UTM clipboard transfer. As we shall see, it is quite a bit more complicated than UTM primary transfer. This section provides the following information:

- A high-level overview of clipboard transfers
- A description of deferred clipboard transfers and snapshots
- A step-by-step detailed walk through of a UTM clipboard transfer
- Sample code to implement a UTM clipboard transfer

10.5.1 Compatibility with Previous Releases

Motif provides a large set of *XmClipboard* routines. (See the *Motif 2.1—Programmer's Reference* for details on each routine, or see the *Motif 2.1—Programmer's Guide* for a summary of the routines.)

Although these routines are still supported, we recommend that you use UTM instead. In other words, you should use UTM as your interface to the clipboard.

10.5.2 High-Level Overview of Clipboard Transfers

A clipboard transfer really consists of two separate transfers. In the first transfer, the source widget transfers the data to the clipboard. The clipboard acts as a pseudo-destination widget for this first transfer. In the second transfer, the clipboard transfers the data to the destination widget. For this second transfer, the clipboard acts as a pseudo-source widget.

When the source widget transfers data to the clipboard, the source widget has no idea what target the destination widget might request. Therefore, the source widget typically converts the selection to several different targets and transfers each of these targets to the clipboard. For example, if the source widget displays text, the source widget might transfer the selected text to the clipboard one time for each of the five standard textual targets. When the destination widget requests the data from the clipboard, the destination widget will (hopefully) ask for one of the five standard textual targets. If it does, the clipboard can transfer the selection to the destination widget.

Clipboard transfer is inherently less efficient than primary transfer. However, Motif does provide a somewhat more efficient mechanism for clipboard transfer called "deferred" clipboard transfer.

10.5.3 Immediate Versus Deferred Clipboard Targets

The main problem with any clipboard transfer is that the source widget has to copy many different versions of the selection to the clipboard. Using deferred clipboard transfer does not really solve this problem; however, it does make the copying operations significantly cheaper. That is because the source widget need only copy "pointers" or references to the selection, rather than copying the selection itself.

UTM always asks your clipboard source widget to provide a list of its immediate clipboard targets (_MOTIF_CLIPBOARD_TARGETS) and its list of deferred clipboard targets (_MOTIF_DEFERRED_CLIPBOARD_TARGETS). It is up to you to decide which clipboard targets fall into which category. It is acceptable to make all clipboard targets immediate, or to make them all deferred, or to use some combination of the two. One rule of thumb is that a clipboard target should be deferred if it is relatively expensive to copy it to the clipboard.

10.5.4 Snapshots

This section explains how you must use snapshots to implement deferred clipboard targets.

If your source widget is to act as a clipboard source, UTM will ask your **convertProc** trait method to convert the _MOTIF_DEFERRED_CLIPBOARD_TARGETS target. Your **convertProc** trait method should respond to this request as it would respond to any request for a list of targets; namely, by building a list of deferred clipboard targets. For example, suppose we want the list of deferred clipboard targets to consist of _MOTIF_COMPOUND_STRING and *STRING*. In this case, the following **convertProc** code should be sufficient:

```
ConvertProc(
     Widget w,
     XtPointer client_data,
     XtPointer call_data)
{
 XmConvertCallbackStruct *cs = (XmConvertCallbackStruct *) call_data;
 Atom _MOTIF_DEFERRED_CLIPBOARD_TARGETS = XInternAtom(XtDisplay(w),
                         "_MOTIF_DEFERRED_CLIPBOARD_TARGETS", False);
 Atom _MOTIF_COMPOUND_STRING = XInternAtom(XtDisplay(w),
                         "_MOTIF_COMPOUND_STRING", False);
 Atom STRING = XInternAtom(XtDisplay(w), "STRING", False);
 ...

  if (cs->target = _MOTIF_DEFERRED_CLIPBOARD_TARGETS)  {
    Atom *targs = (Atom *) XtMalloc(sizeof(Atom) * 2);
    int   n;
    n = 0;
    targs[n] = _MOTIF_COMPOUND_STRING; n++;
    targs[n] = STRING; n++;

    cs->value = (XtPointer)targs;
    cs->type = XA_ATOM;
    cs->length = n;
    cs->format = 32;
  }
  ...
```

Since the *length* member of the **XmConvertCallbackStruct** was non-zero, UTM will call the **convertProc** trait method again This time, UTM will ask your **convertProc** trait method to convert the _MOTIF_SNAPSHOT target. Your **convertProc** must respond to this target by doing the following:

- Save a "snapshot" of the selection.
- Generate a new atom that uniquely distinguishes this snapshot.

A snapshot is nothing more than a copy of the current clipboard selection in whatever format you decide to save it in. (UTM does not define how the snapshot must be stored.) If you want, you can go ahead and convert the selection to _MOTIF_COMPOUND_STRING format and to *STRING* format and then save the converted selection somewhere in your widget. However, it will probably be more efficient just to save the selection somewhere safe and to worry about doing the conversions later.

The distinguisher atom must be unique, so you are going to have to devise some algorithm for generating unique atom string names. One idea is to base the snapshot atom string name on the widget ID itself. For example:

```
static Atom
GenerateUniqueDistinguisherAtom(Widget w)
{
  char  distinguisher_atom_string[32];
  ExmMyWidget  my_widget = (ExmMyWidget)w;
   sprintf(distinguisher_atom_string, "_EXM_SNAP_%lX_%X", (long)w,
                 my_widget.snap_counter++);
   return (XInternAtom(XtDisplay(w), distinguisher_atom_string, False));
}
```

Then your **convertProc** trait method can convert the _MOTIF_SNAPSHOT request as follows:

```
 ...
 Atom *distinguisher_pointer;

 if (cs->target = _MOTIF_SNAPSHOT)  {
   /* Save the clipboard selection somewhere. */
     ...

   /* Now return the distinguisher atom. */
     distinguisher_pointer = (Atom *) XtMalloc(sizeof(Atom));

    *distinguisher_pointer = GenerateUniqueDistinguisherAtom(w);
     cs->value = (XtPointer)distinguisher_pointer;
     cs->type  = XA_ATOM;
     cs->length = 1;
     cs->format = 32;
 }
 ...
```

UTM stores this distinguisher atom on the clipboard. This one distinguisher atom symbolizes both the deferred clipboard targets (*STRING* and _MOTIF_COMPOUND_STRING). Later, the destination widget may request one of the deferred clipboard targets; for example, *STRING*. If it does, UTM will call your widget's **convertProc** trait method, passing the following in the **XmConvertCallbackStruct**:

- *STRING* as the *target* member
- The distinguisher atom as the *selection* member

Your widget's **convertProc** trait method must respond to this request by finding the snapshot and converting it to *STRING*. When the conversion is completed, UTM automatically passes the converted data to the clipboard. Then, the clipboard automatically passes the converted data to the destination widget.

10.5.5 Clipboard Transfer: Step by Step

This section takes you step by step through a typical clipboard transfer. Remember that this is a typical transfer, but it is by no means the only way a clipboard transfer might happen. In this section, we use the term "conversion routines" to mean a combination of any **XmNconvertCallback** routines of the application and the **convertProc** trait method of the source widget.

Following are the steps:

1. The user selects some portion of the source widget. The source widget will probably highlight the selection.
2. The user presses some mouse button or key sequence to initiate a clipboard cut or a clipboard copy. The Intrinsics respond to the event by calling the appropriate action method in the source widget.
3. The called action method itself calls **XmeClipboardSource**.
4. **XmeClipboardSource** asks the conversion routines to provide a list of _MOTIF_CLIPBOARD_TARGETS.
5. The conversion routine returns the list of immediate clipboard targets that it intends to place on the clipboard. The conversion routine should not include *TARGETS* in this list.
6. **XmeClipboardSource** places the list of _MOTIF_CLIPBOARD_TARGETS on the clipboard.
7. **XmeClipboardSource** calls the conversion routines one time for each target in the list of _MOTIF_CLIPBOARD_TARGETS. So, for example, if four targets were returned in _MOTIF_CLIPBOARD_TARGETS, **XmeClipboardSource** calls the conversion routines four times, each time asking the conversion routine to convert the selection to a different immediate target.
8. **XmeClipboardSource** asks the conversion routines to provide a list of _MOTIF_DEFERRED_CLIPBOARD_TARGETS.
9. Regardless of the number of deferred clipboard targets, **XmeClipboardSource** calls the conversion routines only once. This call asks the conversion routines to convert the selection to the _MOTIF_SNAPSHOT target.
10. The conversion routines convert the _MOTIF_SNAPSHOT target by saving a snapshot of the data and by returning a distinguisher atom that uniquely identifies the snapshot.

11. **XmeClipboardSource** copies each deferred clipboard target to the clipboard, but it does not copy any converted data to the clipboard. For each deferred clipboard target, **XmeClipboardSource** sets the *selection* member to the distinguisher atom.
12. If the user had requested a clipboard cut operation, **XmeClipboardSource** asks the conversion routine to convert the selection to the *DELETE* target. The conversion routine responds to this target by deleting the selection.
13. **XmeClipboardSource** establishes a callback that is automatically called whenever a request is made to convert data to a deferred clipboard target. We will call that callback **CallbackD**.
14. The user moves the cursor to the destination widget and presses some mouse button or key sequence to initiate a clipboard paste.
15. The Intrinsics call the action method of the destination widget that is associated with the clipboard paste event. This action method calls **XmeClipboardSink**.
16. **XmeClipboardSink** calls the destination widget's **destinationPreHookProc** trait method, if any.
17. **XmeClipboardSink** then calls the destination routines associated with the destination widget. The first destination routines called are the **XmNdestinationCallback** procedures that the application has associated with the destination widget. The next destination routine called is the **destinationProc** trait method of the destination widget. Typically, one of the destination routines will ask the clipboard to return its list of *TARGETS*.
18. The clipboard responds to a *TARGETS* request by returning a list of all the targets in _MOTIF_CLIPBOARD_TARGETS and _MOTIF_DEFERRED_CLIPBOARD_TARGETS, plus the two additional targets *TIMESTAMP* and *TARGETS*.
19. The destination routine examines the returned list of *TARGETS* and requests the most desirable one from the clipboard.
20. The clipboard examines the request. If the requested target is an immediate clipboard target, the clipboard transfers the data back to the destination routine. Skip to Step 22.
21. If the requested target is a deferred clipboard target, UTM calls **CallbackD**, which was the callback established by **XmeClipboardSource** to handle this kind of request.

22. **CallbackD** asks the conversion routines to convert the requested target. **CallbackD** will set the *selection* member to the distinguisher atom.
23. The conversion routine locates the snapshot data and converts it to the requested target.
24. UTM transfers the converted data to the transfer procedure of the destination routine.
25. When the snapshot data is no longer needed, **CallbackD** asks the conversion routines to convert the snapshot to the *DONE* target, using the distinguisher atom as the selection.
26. UTM returns control to the transfer procedure associated with a destination routine. The transfer procedure must paste the transferred data into the appropriate place in the destination widget.

10.5.6 Sample Translations for a Clipboard Transfer

The *ExmStringTransfer* demonstration widget supports clipboard transfer. The following translations are relevant to clipboard transfer:

```
:<Key>osfPaste: ExmStringTransferPasteClipboard()\n\
:<Key>osfCut: ExmStringTransferCopyClipboard()\n\
:s <Key>osfInsert: ExmStringTransferPasteClipboard()\n\
:s <Key>osfDelete: ExmStringTransferCopyClipboard()\n\
```

To copy the text in an *ExmStringTransfer* widget to the clipboard, a user can either press the **osfCut** or **osfDelete** virtual keys. Despite their ominous names, these keys do not remove or alter any text in the *ExmStringTransfer* widget. A user pastes data from the clipboard to an *ExmStringTransfer* widget by pressing the **Shift** **osfInsert** or **Shift** **osfDelete** virtual keys.

10.5.7 A Sample Copy Clipboard Routine

The clipboard copy routine of *ExmStringTransfer* is shown as follows:

```
static void
ExmStringTransferCopyClipboard(
     Widget w,
     XEvent *event,
     String *params,
     Cardinal *num_params)
{
  Time time;

  /* First we must obtain a timestamp. This is required for
     ICCCM compliant ownership of a selection. We use
     XtLastTimestampProcessed which holds the timestamp
     from the last event the Intrinsics saw with a timestamp. */
  time = XtLastTimestampProcessed(XtDisplay(w));

  /* When we call XmeClipboardSource,  either the Motif clipboard
     will request the current selection data or an external
     clipboard manager will obtain the data. When the data is
     obtained, the CLIPBOARD selection will be owned by the
     data holder */
  XmeClipboardSource(w, XmCOPY, time);
}
```

10.5.8 A Sample Paste Clipboard Routine

The clipboard paste routine of the *ExmStringTransfer* widget is shown as follows:

```
static void
ExmStringTransferPasteClipboard(
     Widget w,
     XEvent *event,
     String *params,
     Cardinal *num_params)
{
  /* Calling XmeClipboardSink will start the process of requesting
     the CLIPBOARD selection to be pasted using the transfer trait
     destination callback for this widget class */
  XmeClipboardSink(w, XmCOPY, NULL);
}
```

10.6 Drag and Drop Transfer Details

Drag and drop is similar to primary transfer. Like primary transfer, drag and drop transfers the selection directly to another widget without going through a clipboard. Unlike primary transfer, drag and drop provides sophisticated visuals to help the user understand various details of the transfer. For example, the source icon of a drag and drop transfer symbolizes the kind of object being dragged.

As a widget writer, you are responsible for supplying the UTM underpinnings of a drag and drop operation.You may also provide various drag and drop icons; however, the application or user may override your choices. This chapter focuses on the UTM underpinnings of drag and drop. That is, we will focus on the kinds of activities that widget writers have to do in order to implement drag and drop in a widget. We will not focus on the large family of drag and drop visuals. (For information on these topics, see the *Motif 2.1—Programmer’s Guide*.)

10.6.1 Compatibility with Previous Releases

Drag and drop was introduced in Motif Release 1.2. If you wrote a drag and drop application or widget at Release 1.2, it should continue to work at Motif Release 2.0. However, if you are writing a widget at Motif Release 2.0, you should use UTM to implement any drag and drop features.

We recommend converting any existing pre-UTM drag and drop code to UTM.

10.6.2 Drag and Drop Transfers: Step by Step

This section takes you step by step through a typical drag and drop transfer. Remember that this is a typical transfer, but it is by no means the only way a drag and drop transfer might happen. In this section, we use the term "conversion routines" to mean a combination of any **XmNconvertCallback** routines of the application and the **convertProc** trait method of the source widget.

Following are the steps:

1. The user selects some portion of the source widget. The source widget will probably highlight the selection.
2. The user presses some mouse button or key sequence to initiate a drag operation. Typically, the user initiates a drag operation by pressing Btn2Down. The Intrinsics respond to the event by calling the appropriate action method in the source widget.
3. The called action method itself calls **XmeDragSource**. The call to **XmeDragSource** can specify any **XmDragContext** resources. (**XmeDragSource** will ultimately instantiate an **XmDragContext** widget.)
4. **XmeDragSource** asks the conversion routines to provide a list of _MOTIF_EXPORT_TARGETS.
5. The conversion routines must respond to this request by returning a list of draggable targets. This list may or may not consist of all the targets that the conversion routines can convert. Typically, this list consists only of the targets that the destination widget is most likely to request.
6. **XmeDragSource** calls **XmDragStart**, which in turn instantiates an **XmDragContext**. The **XmDragContext** displays the appropriate drag icons. **XmeDragSource** sets the **XmNexportTargets** resource of **XmDragContext** to the list of draggable targets. Similarly, **XmeDragSource** sets the **XmNnumExportTargets** resource of **XmDragContext** to the number of draggable targets.
7. The user drags these icons towards the destination widget. The user typically does this by continuing to press Btn2Down while moving the mouse.
8. When the user is over the destination widget, the user releases some mouse button or presses some key sequence to initiate a drop operation. Typically, the user initiates the drop operation by releasing the mouse button, generating Btn2Up.
9. If the destination widget has not been registered as a drop site, UTM will reject the attempted drop and the transfer will end. If the destination widget has been registered as a drop site, the transfer will continue. A widget registers itself as a drop site by calling **XmeDropSink**. Note that the widget calls **XmeDropSink** when the widget is first instantiated, not as a response to the Btn2Up event. **XmeDropSink** is the UTM interface to the *XmDropSite* registry.
10. UTM calls the destination widget’s **destinationPreHookProc** trait method, if any. UTM passes an **XmDestinationCallbackStruct** as the *call_data* argument.
11. UTM then calls the destination routines associated with the destination widget. The first destination routines called are the **XmNdestinationCallback** procedures

that the application has associated with the destination widget. The next destination routine called is the **destinationProc** trait method of the destination widget. UTM passes a pointer to an **XmDropProcCallbackStruct** in the *destination_data* member of the **XmDestinationCallbackStruct** (The members of the **XmDestinationCallbackStruct** are documented in the *XmDropSite* reference page.)

12. A destination routine typically calls *XtGetValues* on the **XmNexportTargets** resource of the **XmDragContext** widget.
13. A destination routine examines the returned **XmNexportTargets** and calls *XmTransferValue* to request the most desirable one. If none of the targets in **XmNexportTargets** are desirable, a destination routine will call *XmTransferValue* and ask for the complete list of *TARGETS*. (Remember that **XmNexportTargets** does not necessarily hold a complete list of targets, but *TARGETS* does.)
14. UTM asks the conversion routines of the source widget to convert the requested target.
15. A conversion routine attempts to convert the selection to the desired target.
16. When the conversion is complete, UTM transfers the converted data to the transfer procedure of the destination widget.
17. The transfer procedure will find the converted selection in the *value* member of the **XmSelectionCallbackStruct**. The transfer procedure need only paste the converted selection into the appropriate place in the destination widget.

10.6.3 Sample Translations for a Drag and Drop Transfer

ExmStringTransfer provides only one drag and drop translation, which is as follows:

```
<Btn2Down>:             ExmStringTransferProcessDrag()\n\
```

Notice that there is no explicit action method associated with Btn2Up. In other words, *ExmStringTransfer* does not need to provide code that detects when the user is signalling to do a drop. Instead, *ExmStringTransfer* simply registers itself as a potential drop site. Once registered, the drag and drop system built into Motif will catch the **Btn2Up** event.

To register itself as a potential drop site, *ExmStringTransfer* has to specify a list of its import targets. That is, the widget must describe the kinds of targets that can be copied into it. The majority of the **RegisterDropSite** method establishes this import target list. The actual registration of the import targets as a drop site takes place through the **XmeDropSink** call.

10.6.4 A Sample Drag Action

The *ExmStringTransferProcessDrag* method initiates a drag operation by initializing the **XmDragContext**. (See the *Motif 2.1—Programmer’s Guide* for more information on **XmDragContext**.) The code for *ExmStringTransferProcessDrag* appears as follows:

```
static void
ExmStringTransferProcessDrag(
     Widget w,
     XEvent *event,
     String *params,
     Cardinal *num_params)
{
 ExmStringTransferWidget stw = (ExmStringTransferWidget) w;
 Arg args[4];
 Cardinal n;

  if (! stw -> string_transfer.own_primary) {
    ExmStringTransferPastePrimary(w, event, params, num_params);
    return;
  }

  /* Initialize DragContext resources. We want the drag icon to
     indicate text, and we support only the COPY operation. */
  n = 0;
  /* Normal drag and drop behavior will be to use the foreground
     and background of the widget when creating the cursor or
     or the pixmap drag icon */
  XtSetArg(args[n], XmNcursorBackground, stw->core.background_pixel);  n++;
  XtSetArg(args[n], XmNcursorForeground, stw->primitive.foreground);  n++;
  /* We use the default text drag icon, the same as used in
     the standard Motif widgets. */
  XtSetArg(args[n], XmNsourceCursorIcon, XmeGetTextualDragIcon(w));  n++;
  XtSetArg(args[n], XmNdragOperations, XmDROP_COPY); n++;
  (void) XmeDragSource(w, NULL, event, args, n);
}
```

XmeDragSource ultimately creates a DragIcon; the first three resources—**XmNcursorBackground**, **XmNcursorForeground**, and **XmNsourceCursorIcon**—control the appearance of that DragIcon. The convenience function **XmeGetTextualDragIcon** gets the standard Motif textual DragIcon If you are writing a widget that is dragging something other than text, then you would specify a different DragIcon.

10.6.5 Sample Action to Register a Drop Site

ExmStringTransfer can be a drop site. To be a drop site, the widget's **Initialize** method calls a routine named **RegisterDropSite**. **RegisterDropSite** builds a list of drop (import) targets and then calls **XmeDropSink**. The **RegisterDropSite** routine is as follows:

```
static void
RegisterDropSite(
     Widget w)
{
  Atom TEXT = XInternAtom(XtDisplay(w), "TEXT", False);
  Atom COMPOUND_TEXT = XInternAtom(XtDisplay(w), "COMPOUND_TEXT", False);
  Atom LOCALE_ATOM = XmeGetEncodingAtom(w);
  Atom MOTIF_C_S = XInternAtom(XtDisplay(w), "_MOTIF_COMPOUND_STRING", False);
  Atom targets[5];
  Arg args[2];
  int n, nt;

  /* Set up import targets. These are the targets from which we can
     generate a compound string when a drop is made. */
  nt = 0;
  targets[nt++] = MOTIF_C_S;
  targets[nt++] = COMPOUND_TEXT;
  targets[nt++] = LOCALE_ATOM;
  if (LOCALE_ATOM != XA_STRING) {
    targets[nt++] = XA_STRING;
  }
  if (LOCALE_ATOM != TEXT) {
    targets[nt++] = TEXT;
  }
  /* If you add any more targets, bump the array size. */

  n = 0;
  XtSetArg(args[n], XmNimportTargets, targets); n++;
  XtSetArg(args[n], XmNnumImportTargets, nt); n++;
  XmeDropSink(w, args, n);
}
```

According to the code, the user can drop any one of several different standard textual targets into the widget.

10.7 Writing a Conversion Routine

This section explains how to write a conversion routine that can convert various targets. In this section, we will examine how **ExmStringTransfer** implemented its **convertProc** trait method.

Although we have assumed that you are a widget writer working on a **convertProc** trait method, the conversion information in this section also applies to an application programmer writing an **XmNconvertCallback** procedure. Throughout this section, we use the term "conversion routine" to mean either a **convertProc** trait method or an **XmNconvertCallback** procedure.

When writing a conversion routine, you must do the following tasks:

1. Declare **Atom** variables for every atom that the conversion routine uses.
2. Provide code that converts requests for lists of targets supported by your widget. For example, your conversion routine must be able to return a list of *TARGET* atoms.
3. Provide code that converts all the standard targets.
4. If your widget can be the source of textual data, then your conversion routine should be able to convert the standard textual targets.
5. Provide code that converts any other targets supported by your widget.

6. Provide code that properly handles requests for targets not supported by your widget.

The remainder of this section explores all of the preceding tasks.

10.7.1 Declare Atoms

A conversion routine must declare **Atom** variables for every target referenced in the convert procedure. The **ConvertProc** trait method of *ExmStringTransfer* used **XInternAtom** as follows to create most of its **Atom** variables:

```
enum { CLIPBOARD, TEXT, COMPOUND_TEXT, FOREGROUND,
       BACKGROUND, TARGETS, MOTIF_DROP,
       MOTIF_C_S, MOTIF_EXPORT_TARGETS,
       CLIPBOARD_IMMEDIATE, LOSE_SELECTION,
       /* special values */ LOCALE_ATOM, NUM_ATOMS };
char* atom_names[] = {
    XmSCLIPBOARD, XmSTEXT, XmSCOMPOUND_TEXT, "FOREGROUND",
    "BACKGROUND", XmSTARGETS, XmS_MOTIF_DROP,
    XmS_MOTIF_COMPOUND_STRING, XmS_MOTIF_EXPORT_TARGETS,
    XmS_MOTIF_CLIPBOARD_TARGETS, XmS_MOTIF_LOSE_SELECTION };
Atom atoms[NUM_ATOMS];

assert(XtNumber(atom_names) == NUM_ATOMS - 1);
XInternAtoms(XtDisplay(w), atom_names, NUM_ATOMS - 1, False, atoms);
atoms[LOCALE_ATOM] = XmeGetEncodingAtom(w);
```

Wherever the code referenced one of these atoms (say "MOTIF_C_S"), that would be replaced by an index into the array ("atoms[MOTIF_C_S]"). The *LOCALE_ATOM* was the only exception; **ConvertProc** called **XmeGetEncodingAtom** as follows to get the *LOCALE_ATOM*:

```
Atom LOCALE_ATOM = XmeGetEncodingAtom(w);
```

10.7.2 Provide Target Lists

The ICCCM insists that source widgets respond to a *TARGETS* request by returning a list of all the target atoms that your widget can convert. If your widget serves as a source for a clipboard transfer, then your conversion routine must also be able to handle the request for a list of _MOTIF_CLIPBOARD_TARGETS and _MOTIF_DEFERRED_CLIPBOARD_TARGETS. (See Section 10.5 for more details on this target.) If your widget serves as a source for a drag and drop transfer, then your conversion routine must also be able to handle the request for a list of _MOTIF_EXPORT_TARGETS. (See Section 10.6 for more details on this target.)

To help handle the *TARGETS* request, Motif provides the **XmeStandardTargets** routine. This routine carves out enough dynamic memory to hold a list of all the standard and nonstandard targets your widget supports. Then, this routine fills the first slots of this dynamic memory with the names of the standard targets. It is up to your code to fill in the remaining slots with the names of the nonstandard targets.

For example, the **ExmStringTransfer** widget needs to respond to *TARGETS* by returning a list consisting of all the standard targets and all the standard textual targets. The following code fragment shows how to build this list of targets:

```
XmConvertCallbackStruct *cs = (XmConvertCallbackStruct *) call_data;
...
 if (cs->target == TARGETS) {
   /* We convert the standard targets, plus up to five additional
      standard textual targets. */
   Atom *targs = XmeStandardTargets(w, 5, &n);

   targs[n] = MOTIF_C_S; n++;
   targs[n] = COMPOUND_TEXT; n++;
   targs[n] = LOCALE_ATOM; n++;
   if (LOCALE_ATOM != XA_STRING) {
     targs[n] = XA_STRING; n++;
   }
   if (LOCALE_ATOM != TEXT) {
     targs[n] = TEXT; n++;
   }

   value = (XtPointer) targs;
   type = XA_ATOM;
   length = n;
   format = 32;
   ...
 }
```

(**ConvertProc** will eventually assign the values of the *value*, *type*, *length*, and *format* variables to their respective **XmConvertCallbackStruct** member fields.)

Although *TARGETS* should include all the standard targets, the more specialized target list requests (such as _MOTIF_EXPORT_TARGETS) do not have to. If you do not want the standard targets to be part of a targets list, use **XtMalloc** instead of **XmeStandardTargets**; for example:

```
if (cs -> target == MOTIF_EXPORT_TARGETS) {
  Atom *targs = (Atom *) XtMalloc(sizeof(Atom) * 7);
```

10.7.3 Convert Standard Targets

All Motif source widgets should be able to convert requests for the following standard targets:

BACKGROUND
Your widget transfers the value of Core's **XmNbackground** resource as type *PIXEL*.

CLASS Your widget finds the first shell in its widget hierarchy that has a WM_CLASS property and transfers the contents as text in the current locale.

CLIENT_WINDOW
Your widget finds the first shell in the widget hierarchy and transfers its window as type *WINDOW*.

COLORMAP
Your widget transfers the value of Core's **XmNcolormap** resource as type *COLORMAP*.

FOREGROUND
Your widget transfers the value of Primitive's or Manager's **XmNforeground** resource as type *PIXEL*.

NAME Your widget finds the first shell in the widget hierarchy that has a WM_NAME property and transfers the contents as text in the current locale.

TARGETS Your widget transfers, as type *ATOM*, a list of all the targets your widget can provide.

TIMESTAMP
Your widget transfers the timestamp used to acquire the selection as type *INTEGER*.

_MOTIF_RENDER_TABLE
Your widget transfers the value of its render table resource if one exists, or else the default render table for the widget, as type *STRING*.

_MOTIF_ENCODING_REGISTRY
Your widget transfers the source widget’s encoding registry as type *STRING*. The value is a list of *NULL* separated items in the form of tag encoding pairs. This target symbolizes the transfer target for the Motif Segment Encoding Registry. Widgets and applications can use this registry to register text encoding formats for specified render table tags. Applications access this registry by calling **XmRegisterSegmentEncoding** and **XmMapSegmentEncoding**. A destination widget can request the _MOTIF_ENCODING_REGISTRY target when transferring an **XmString** between two applications.

Fortunately, your conversion routine does not have to write conversion code for most of these targets. Your conversion routine can call **XmeStandardConvert**, which automatically converts 9 of the 10 standard targets. **XmeStandardConvert** cannot convert *TIMESTAMP*; however, the Intrinsics can. Even though **XmeStandardConvert** can convert *TARGETS*, your own conversion routine should typically take responsibility for doing this conversion. That is, **XmeStandardConvert** will convert *TARGETS* by returning a list of all the standard targets only. This returned list will be insufficient if your conversion routine can also convert some nonstandard targets.

For example, the following code from the **ConvertProc** trait method of *ExmStringTransfer* demonstrates how to convert the standard targets:

```
XmConvertCallbackStruct *cs = (XmConvertCallbackStruct *) call_data;
  ...
  XmeStandardConvert(w, NULL, cs);
```

```
if (cs -> value == NULL) /* could not convert the target */
  cs -> status = XmCONVERT_REFUSE;
else /* successfully converted the target */
     cs -> status = XmCONVERT_DONE;
```

10.7.4 Convert Standard Textual Targets

Textual widgets vary a lot in the richness of the text they support. Some textual widgets only store the ASCII values of each character. Other textual widgets store a great wealth of information about the text, such as the fonts or the locale encoding. When a user tries to transfer data between two different kinds of textual widgets, there is no guarantee that the two widgets will share the same level of text "richness." For example, perhaps the source widget stores font information about its text, and the destination widget simply stores the ASCII value of each character.

The source widget must be able to supply the destination widget with text at the richest level that the source widget supports. In addition, the source widget must also be able to supply text at all poorer levels beneath that richest level. Here are the levels of text, from poorest to richest:

STRING
: The text includes only characters in ISO8859-1 plus TAB and NEWLINE. It is known as the C locale.

LOCALE_ENCODING
: This is a string in the specified locale. The value of *LOCALE_ENCODING* in the C locale is *STRING*.

TEXT
: This target returns either *LOCALE_ENCODING* or *COMPOUND_TEXT*.

COMPOUND_TEXT
: This is an encoding the X Windows System supplies to deal with strings that have multiple encodings.

_MOTIF_COMPOUND_STRING
: This is text in **XmString** format.

For example, a widget that displays text in **XmString** format must be able to convert requests from destination widgets for all five textual targets.

A selection owner can use **XmbTextListToTextProperty** or **XwcTextListToTextProperty** to convert text in its own locale to a text property. The type of the property is determined by the composition of the text and by the encoding style passed to **XmbTextListToTextProperty**. Encoding styles exist for converting text to *STRING*, *COMPOUND_TEXT*, and the encoding of the locale. Another encoding style specifies conversion to *STRING* if all the characters in the text can be so converted, or otherwise to *COMPOUND_TEXT*.

A Motif application that has text in compound strings can use **XmCvtXmStringToCT** to convert a compound string to compound text The application can then place the compound text in the requestor’s property by using type *COMPOUND_TEXT*.

STRING, *COMPOUND_TEXT*, and the locale encoding can also be selection targets.To obtain a text selection in its own locale, an application can request conversion to one of these targets and can then call **XmbTextPropertyToTextList** or **XwcTextPropertyToTextList** to convert the returned property to text in the current locale. An application can also request conversion to *TEXT*, but there is no guarantee that it can convert the returned property to text in the current locale.

One possible strategy is first to request conversion to *TARGETS*. If one of the returned targets is the encoding of the current locale (as determined by a call to **XmbTextListToTextProperty** with an encoding style of **XTextStyle**), the application can request conversion to that target. Otherwise, if one of the returned targets is *COMPOUND_TEXT*, the application can request conversion to that target. If neither the locale encoding nor *COMPOUND_TEXT* is one of the returned targets, the application can request conversion to *STRING* or *TEXT* if the selection owner supports one of those targets.

A Motif application that has text in compound strings can request conversion of a selection to *COMPOUND_TEXT* and can then use **XmCvtCTToXmString** to convert the returned property to a compound string.

The **ConvertProc** trait method of *ExmStringTransfer* does not do these conversions itself. Instead, **ConvertProc** calls the **ConvertCompoundString** routine as follows to handle the text conversions:

```
if (cs->target == MOTIF_C_S ||
    cs->target == COMPOUND_TEXT || cs->target == TEXT ||
    cs->target == LOCALE_ATOM   || cs->target == XA_STRING) {
  /* Convert the compound string to the appropriate target. */
  cstatus = ConvertCompoundString(w, cstring, cs -> target, &value,
                                 &type, &format, &length, &nchars);
```

10.7.5 Targets with Side Effects for the Owner

Some targets have side effects for the owner. Among these targets are the following:

DELETE The owner deletes the selection and, if successful, returns a zero-length property of type *NULL*.

INSERT_SELECTION
The requestor places in its specified window property a pair of atoms that names a selection and a target. The owner requests conversion of the specified selection to the specified target and places the result at the location of the selection named in the *INSERT_SELECTION* request. The owner then returns a zero-length property of type *NULL*. For example, the Motif Text widget uses this target with the destination selection when it asks the owner of the destination selection to insert the secondary selection at the destination.

INSERT_PROPERTY
The requestor places in its specified window property some data to be inserted at the location of the selection named in the request. The owner then returns a zero-length property of type *NULL*.

_MOTIF_LOSE_SELECTION
UTM sends this atom to the former owner of the selection when that selection is lost. Upon receiving this message, your widget is responsible for changing the visuals of the formerly selected region to indicate that it is no longer selected.

10.7.6 Summary of Target Conversion

The following fragment suggests the framework for an idealized **convertProc** trait method:

```
static void
ConvertProc(Widget w,
            XtPointer client_data,
            XtPointer call_data)
{
 /* Cast call_data to an XmConvertCallbackStruct. */
   XmConvertCallbackStruct *cs = (XmConvertCallbackStruct *)call_data;

 /* Declare Atom variables for all targets used in converProc */
   Atom TEXT = XInternAtom(XtDisplay(w), "TEXT", False);
   ...

 /* Handle requests for TARGETS by building a list of all standard
    targets and all X nonstandard targets. */
   nonstandard_targets = 5;
   if (cs->target == TARGETS)
     Atom *targs = XmeStandardTargets(w, nonstandard_targets, &n);

 /* If your widget supports CLIPBOARD transfer, handle requests for
    _MOTIF_CLIPBOARD_TARGETS and _MOTIF_DEFERRED_CLIPBOARD_TARGETS. */
   if (cs->target == _MOTIF_CLIPBOARD_TARGETS)
     XtMalloc(...) /* generate list of immediate clipboard targets */
   else if (cs->target == _MOTIF_DEFERRED_CLIPBOARD_TARGETS)
     XtMalloc(...) /* generate list of deferred clipboard targets */

 /* If your widget supports drag, handle requests for _MOTIF_EXPORT_TARGETS. */
   if (cs->target == _MOTIF_EXPORT_TARGETS)
     XtMalloc(...) /* generate list of draggable targets */

 /* If a previous conversion routine has supplied a list of some kind of
    target, merge it with the list your conversion routine has created. */
   if (cs->status == XmCONVERT_MERGE)
     XmeConvertMerge(cs->value, cs->type, cs->format, cs->length, cs);
     ...

 /* Handle requests for nonstandard targets by converting to the requested
    target. */
   ...

 /* Handle requests for standard targets by calling XmeStandardConvert. */
   XmeStandardConvert(w, NULL, ConvertCallbackStruct);

 /* Return the appropriate status. */
```

```
   if (conversion was not successful)
     cs->status = XmCONVERT_REFUSE;
   else if (conversion was successful)
     cs->status = XmCONVERT_DONE;
}
```

10.8 Writing a Destination Routine

A **destinationProc** trait method is usually a lot simpler to write than a **convertProc** trait method. A **destinationProc** need only do the following:

- Accept or reject the user's request.
- If the request is accepted, call **XmTransferValue**. For certain situations, the **destinationProc** may do a little analysis prior to calling **XmTransferValue**.
- If the request is rejected, call **XmTransferDone**.

The remainder of this section examines how **ExmStringTransfer** implemented its **destinationProc** trait method.

The prototype of a **destinationProc** trait method is the same as for any Motif callback. In the *call_data* argument, Motif passes an **XmDestinationCallbackStruct**. Therefore, the opening lines of the **DestinationProc** of **ExmStringTransfer** are as follows:

```
static void
DestinationProc(
     Widget w,
     XtPointer client_data,
     XtPointer call_data)
{
 XmDestinationCallbackStruct *ds = (XmDestinationCallbackStruct *) call_data;
```

As in a **convertProc** trait method, a **destinationProc** trait method must declare **Atom** variables for all the targets it needs, for example:

```
Atom XA_MOTIF_DROP = XInternAtom(XtDisplay(w), "_MOTIF_DROP", False);
Atom TARGETS = XInternAtom(XtDisplay(w), "TARGETS", False);
```

The **ds->operation** field contains the name of the requested operation (link, move, or copy). **ExmStringTransfer** does not support the link operation, so **DestinationProc** uses the following code to reject a link request:

```
if (ds -> operation == XmLINK) {
  /* We don’t support links. */
  XmTransferDone(ds -> transfer_id, XmTRANSFER_DONE_FAIL);
  return;
}
```

ExmStringTransfer can be a drop site. If the user has requested a drop, UTM will pass a pointer to an **XmDropProcCallbackStruct** in the *destination_data* member of the **XmDestinationCallbackStruct**; for example:

```
if (ds -> selection == XA_MOTIF_DROP) {
  XmDropProcCallbackStruct *cb =
          (XmDropProcCallbackStruct *) ds -> destination_data;
```

For most transfer mechanisms, a **destinationProc** needs to ask the source widget for its list of targets. However, this is not always necessary in a drag and drop operation because the DragContext holds this list in its **XmNexportTargets** resource. Therefore, **DestinationProc** gets the target list as follows:

```
n = 0;
XtSetArg(args[n], XmNexportTargets, &targets); n++;
XtSetArg(args[n], XmNnumExportTargets, &num_targets); n++;
XtGetValues(cb -> dragContext, args, n);
```

DestinationProc examines the returned list and picks the most desirable target. (See the next section for more details.) Then **DestinationProc** calls **XmTransferValue** to ask the source widget to convert the target.

```
XmTransferValue(ds -> transfer_id, target,
                (XtCallbackProc) TransferProc, NULL, 0);
```

If the user has requested a transfer other than drag and drop, **DestinationProc** needs to ask the source widget for its list of targets. **DestinationProc** does this by calling **XmTransferValue** as follows:

```
XmTransferValue(ds -> transfer_id, TARGETS,
                (XtCallbackProc) TransferProc, NULL, 0);
```

10.9 Writing a Transfer Procedure

The destination procedure is rather short; most of the serious work on the destination side is done in the widget's transfer procedure. The transfer procedure of **ExmStringTransfer** is called **TransferProc**. UTM calls **TransferProc** when the source has finished converting a target. **TransferProc** must do the following:

- If the source widget has returned a list of *TARGETS*, then **TransferProc** must pick the most desirable target and then request it.
- If the source widget has returned the specified target, then **TransferProc** must paste the new string into the *ExmStringTransfer* widget.

A transfer procedure has the same function prototype as any Motif callback. Motif passes the transfer procedure an **XmSelectionCallbackStruct** pointer as the *client_data* argument. The *targets* member of this callback structure holds the name of the target converted by the source widget. For example, **TransferProc** calls **XmTransferValue** as follows to request a particular target:

```
XmTransferValue(ss -> transfer_id, target,
                (XtCallbackProc) TransferProc, NULL, 0);
```

When the source widget finishes the conversion, UTM calls **TransferProc** again.

When the source widget finishes converting the desired target, **TransferProc** must paste the text into the **ExmStringTransfer** widget. To simplify things, **TransferProc** overwrites whatever text was previously displayed. For more sophisticated widgets, you will have to determine how to insert the transferred text without overriding the previous text.

ExmStringTransfer displays the contents of the *ExmNcompoundString* resource. This resource holds an **XmString** value. Therefore, **TransferProc** must take the text that the source widget transferred and convert it into **XmString** format. If the source widget transferred the text as an _MOTIF_COMPOUND_STRING target, then **TransferProc** will have an easy time converting to **XmString**. If the source widget has transferred the text as some other target, then the conversion will be more difficult. For example, the following code from **TransferProc** handles the situation where the source widget transferred the text as *COMPOUND_TEXT*:

```
if (ss -> type == COMPOUND_TEXT) {
    /* Convert compound text to a compound string.
```

```
     * Note that XmCvtCTToXmString does not convert a list of compound text
     * strings, so we will get only the first if there’s more than one.
     * XmCvtCTToXmString expects a NULL-terminated compound text string,
     * so add a trailing NULL. */
    char *string;
    string = XtMalloc(ss -> length + 1);
   (void) memcpy(string, ss -> value, ss -> length);
    string[ss -> length] = ’ ’;
    cstring = XmCvtCTToXmString(string);
    XtFree(string);
    transferred = True;
}
```

TransferProc contains similar code that converts text in other targets to **XmString** format. Eventually, **TransferProc** uses **XtSetValues** to assign the transferred **XmString** to the **ExmNcompoundString** resource as follows:

```
if (transferred) {
    /* We have a compound string. Use it as the new value of
     * ExmNcompoundString. */
    Arg args[1];
    Cardinal n;
    n = 0;
    XtSetArg(args[n], ExmNcompoundString, cstring);  n++;
    XtSetValues(w, args, n);
}
```

10.9.1 Preferred Textual Targets

In a typical text transfer, the source widget is capable of converting multiple textual targets. The destination widget in a textual transfer needs to pick the richest of these textual targets. The routine in **ExmStringTransfer** that does this is called **PreferredTarget**. **DestinationProc** and **TransferProc** call **PreferredTarget** when the source widget returns its list of supported targets.

PreferredTarget uses the following algorithm to determine which textual target to pick. If the locale atom is present, the precedence order is as follows:

1. _MOTIF_COMPOUND_STRING

2. *TEXT*
3. *COMPOUND_TEXT*
4. locale atom
5. *STRING*

If the locale atom is not present, **PreferredTargets** uses this order:

1. _MOTIF_COMPOUND_STRING
2. *COMPOUND_TEXT*
3. *STRING*

The code in **PreferredTarget** that implements the precedence is as follows:

```
static Atom
PreferredTarget(
     Widget w,
     Atom *targets,
     Cardinal num_targets)
{
  ...
  int n;
  int cs_index = -1;
  int ct_index = -1;
  int locale_index = -1;
  int string_index = -1;
  int text_index = -1;
 /* Which targets can the source convert? Examine the returned targets. */
  for (n = 0; n < num_targets; n++) {
    if (targets[n] == MOTIF_C_S) cs_index = n;
    if (targets[n] == COMPOUND_TEXT) ct_index = n;
    if (targets[n] == TEXT) text_index = n;
    if (targets[n] == LOCALE_ATOM) locale_index = n;
    if (targets[n] == XA_STRING) string_index = n;
  }

 /* If the source supports the locale atom, specify the
    precedence order as follows. */
  if (locale_index >= 0) {
    if (cs_index >= 0) return targets[cs_index];
    if (text_index >= 0) return targets[text_index];
    if (ct_index >= 0) return targets[ct_index];
    if (locale_index >= 0) return targets[locale_index];
    if (string_index >= 0) return targets[string_index];
  } else {
 /* If the source does not support the locale atom, specify the
    precedence order as follows. */
    if (cs_index >= 0) return targets[cs_index];
    if (ct_index >= 0) return targets[ct_index];
    if (string_index >= 0) return targets[string_index];
  }

  return None;
}
```

10.10 Timestamps

Most UTM routines expect a *time* argument. To get that *time* argument, your widget should call **XtLastTimestampProcessed**. If you specify either **CurrentTime** or **0**, UTM will automatically change the call to **XtLastTimestampProcessed**.

With one exception, each UTM conversion request must contain a different timestamp. The one exception is that conversion requests for a multiple transfer can share the same timestamp.

10.11 Transferring Multiple Targets

The data transfers we have looked at so far have all assumed that the destination widget wanted only one conversion from the source widget. For example, the destination might ask for the selection to be converted to *TEXT* or to _MOTIF_COMPOUND_STRING but not to both. However, there are some cases where the destination widget might want the selection converted to several different targets and then transferred. If the user tries to transfer a pixmap, the destination might want the source to transfer both a *PIXMAP* and a *COLORMAP*. For such cases, the destination could call **XmTransferValue** several times, each time asking for a new conversion. However, UTM does provide the following routines to implement a faster multiple transfer:

- **XmTransferStartRequest**
- **XmTransferSetParameters**
- **XmTransferSendRequest**

The calling sequence to transfer multiple values is as follows:

1. Call **XmTransferStartRequest** once to initiate the multiple transfer.
2. Call **XmTransferSetParameters** if a subsequent call to **XmTransferValue** will transfer a value containing a parameter.
3. Call **XmTransferValue** every time you need to transfer a value. For example, call **XmTransferValue** twice to transfer two values.
4. Call **XmTransferSendRequest** once to mark the end of the multiple transfer.

The following code fragment demonstrates how a transfer procedure might request both a *PIXMAP* and a *COLORMAP*:

```
XmSelectionCallbackStruct  *scs;
...
XmTransferStartRequest(scs->transfer_id);
XmTransferValue(scs->transfer_id, PIXMAP,   TransferProc,
                     NULL, XtLastTimestampProcessed);
XmTransferValue(scs->transfer_id, COLORMAP, TransferProc,
                     NULL, XtLastTimestampProcessed);
XmTransferSendRequest(scs->transfer_id, XtLastTimestampProcessed);
...
```

Chapter 11
How to Write a Motif Button Widget

Button widgets, like label widgets, display some noneditable visual such as text or a pixmap. Unlike label widgets, button widgets can be activated by a user. When a user activates a button widget, an application callback is typically triggered.

This chapter explains how to write a Motif button widget. It begins by examining the kinds of buttons we expect you to write, and the kinds of buttons we recommend that you not write. Then, this chapter examines the following three button demonstration widgets:

- The **ExmCommandButton** widget acts as a button in a DialogBox.
- The **ExmMenuButton** widget acts as a menu button. It can be the child of any **XmRowColumn** widget whose **XmNrowColumnType** is set to some value other than **XmWORK_AREA**.
- The **ExmTabButton** widget is meant to serve as a tab child of an **XmNotebook** widget.

11.1 Buttons to Write and Buttons to Avoid Writing

The standard Motif widget set includes several different kinds of button widgets. Since writing a button widget is not a simple task, you should first determine if one of the standard button widgets already does what you need. For example, if you need a button widget to display interesting visuals, you should consider using the **XmDrawnButton** widget rather than writing your own widget.

XmPushButton can serve as a DialogBox button widget. However, you may want to write your own DialogBox button widget if you require features not available in **XmPushButton**. For example, you will need to write your own widget if you want a DialogBox button widget that has a nonrectangular shape. When a button widget becomes the default choice of a DialogBox, the button widget needs to visually alter itself. If you do not like the way that **XmPushButton** visually alters itself, you can write your own DialogBox button widget.

None of the standard Motif button widgets install the *XmQTjoinSide* trait. So, if you want a button widget that knows how to visually merge itself with its manager, you are going to have to write your own.

If you do decide to write your own menu button widget, we do not recommend writing it "from scratch." Rather, we strongly recommend that you do it by modifying the **ExmMenuButton** widget. Note, however, that OSF does not support the **ExmMenuButton** widget in any way.

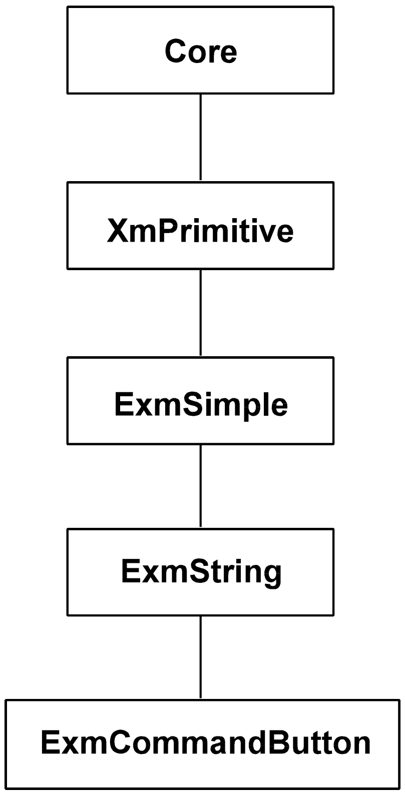
Menu button widgets must be managed by a widget holding the *XmQTmenuSystem* trait. The only standard Motif widget that holds this trait is **XmRowColumn**. Do not write your own *XmQTmenuSystem* widget; always use the **XmRowColumn** widget that comes with the Motif toolkit.

We strongly caution you against writing your own CascadeButton-style widget. If you need a CascadeButton, use the **XmCascadeButton** widget that comes with the Motif toolkit.

11.2 Writing Your Own DialogBox Button

The *ExmCommandButton* widget demonstrates how to code a button widget for a DialogBox. As Figure 11-1 shows, **ExmCommandButton** is a subclass of **ExmString**. **ExmCommandButton** therefore inherits **ExmString**'s ability to display a compound string. **ExmCommandButton** layers on the additional ability to function as an activatable button inside a DialogBox, much the way an **XmPushButton** is used as an OK button inside a standard selection box.

Figure 11–1. ExmCommandButton Position in Hierarchy



In the following subsections, we examine a few features of the *ExmCommandButton*.

11.2.1 Activation

Every DialogBox button widget must be activatable. In other words, when the user explicitly or implicitly chooses your button, your button must call the appropriate activate callbacks. The **ExmCommandButton** widget does the following in order to become activatable:

- Provides an activate callback.
- Installs the *XmQTactivatable* trait on the widget

11.2.1.1 Providing an Activate Callback

ExmCommandButton provides an activate callback resource named **XmNactivateCallback**. The **XmNactivateCallback** resource is defined in the resources array of **CommandB.c** as follows:

```
{
  XmNactivateCallback,
  XmCCallback,
  XmRCallback,
  sizeof(XtCallbackList),
  XtOffsetOf( ExmCommandButtonRec, command_button.activate_callback),
  XmRPointer,
  (XtPointer) NULL
},
```

When an **ExmCommandButton** is activated, it must initialize a callback structure and then call **XtCallCallbackList**. Both of these things are done in several different action methods of *ExmCommandButton*. For example, following is the *ExmCommandButtonActivate* action method:

```
ExmCommandButtonActivate (
        Widget w,
        XEvent *event,
        String *params,
        Cardinal *num_params
)
{
 ExmCommandButtonWidget cw = (ExmCommandButtonWidget)w;
 XmAnyCallbackStruct cb;

  if (cw->command_button.activate_callback) {
 /* Initialize a callback structure. */
    cb.reason = XmCR_ACTIVATE;
    cb.event = event;
    XFlush (XtDisplay(cw));
 /* Call XtCallCallbackList. */
    XtCallCallbackList (w, cw->command_button.activate_callback, &cb);
  }}
```

The **XmNactivateCallback** resource of **ExmCommandButton** uses the generic callback structure **XmAnyCallbackStruct**. Your own DialogBox button widget may define a more elaborate activation callback structure.

11.2.1.2 Installing the XmQTactivatable Trait

The *XmQTactivatable* trait tells a DialogBox that the **ExmCommandButton** widget wishes to be treated as a DialogBox button. In other words, Motif DialogBox widgets check their children for this trait. Only those children holding this trait are eligible to become DialogBox buttons.

The *XmQTactivatable* trait defines a trait method called **changeCB**. A DialogBox calls this trait method to add or remove a callback from the list of activate callbacks held by the DialogBox button widget.

The following shows how **ExmCommandButton** codes the **changeCB** trait method:

```
ChangeCB(
         Widget widget,
         XtCallbackProc activCB,
         XtPointer clientData,
         Boolean setUnset)
{
  if (setUnset)
    XtAddCallback (widget, XmNactivateCallback, activCB, clientData);
  else
    XtRemoveCallback (widget, XmNactivateCallback, activCB, clientData);
}
```

11.2.2 The Default DialogBox Button

The *CDE 2.1/Motif 2.1—Style Guide and Glossary* mandates that a DialogBox widget have a default action associated with it at all times. For example, consider an **XmSelectionBox** having three DialogBox buttons: an OK button, a Cancel button, and an Apply button. Of these three buttons, suppose the OK button is the current default button. When the user activates the default action (typically by pressing Return anywhere in the DialogBox), the DialogBox must act as if the OK button was explicitly pushed.

The default button must provide some visual cue so that users will know that it is, in fact, the default button. Typically, the default button does this by displaying a border around itself; however, the widget writer is somewhat free to pick a different kind of visual cue. (See the *CDE 2.1/Motif 2.1—Style Guide and Glossary* for the exact constraints regarding this visual cue.)

To support default activation, Motif provides the *XmQTtakesDefault* trait.All dialog buttons must install this trait. This trait announces to DialogBox widgets that the dialog button is capable of changing its appearance to show that it is the default button.

The *XmQTtakesDefault* trait provides only one trait method, **showAsDefault**. The DialogBox managing the button will typically call this trait method one or more times, each time asking the button to get into a different state. The DialogBox will probably ask each of its buttons to get into the **XmDEFAULT_READY** state first. Each **ExmCommandButton** responds to this by expanding its margins. The DialogBox widget will then ask one of its buttons to become the default button by asking it to go into the **XmDEFAULT_ON** state. **ExmCommandButton** responds to this state by displaying distinctive highlighting. If a different button becomes the default button, the DialogBox widget asks the former default button to go into the **XmDEFAULT_OFF** state.**ExmCommandButton** responds to this request by removing its distinctive highlighting.What follows is the implementation of the **showAsDefault** trait method of **ExmCommandButton**:

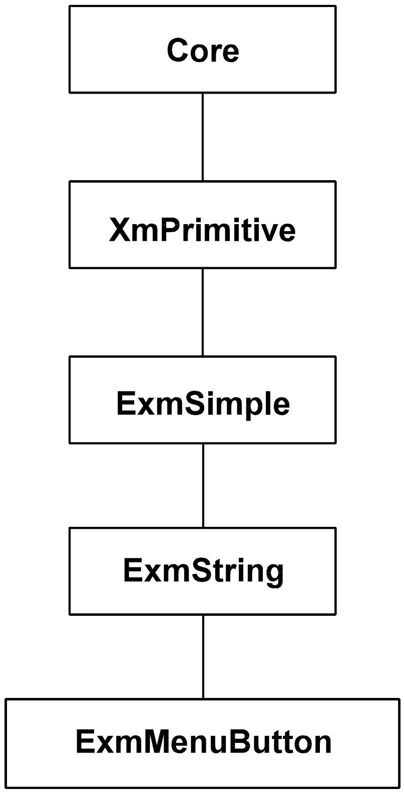
```
ShowAsDefault(Widget w,
              XtEnum state)
{
 ExmCommandButtonWidgetClass cbwc = (ExmCommandButtonWidgetClass)XtClass(w);
 ExmCommandButtonWidget cbw = (ExmCommandButtonWidget)w;
 Position   start_x_of_outer_shadow,  start_y_of_outer_shadow;
 Dimension  margin_push_out;
 Dimension  width_of_outer_shadow, height_of_outer_shadow;
 int   dx, dy, width, height;
 GC    top_GC, bottom_GC;
 Dimension outer_shadow_thickness;
 int       outer_shadow_type;
 int       margins_were_pushed_out=0;
#define MARGIN_BETWEEN_HIGHLIGHT_AND_OUTER_SHADOW 2

  start_x_of_outer_shadow = cbw->primitive.highlight_thickness +
                            MARGIN_BETWEEN_HIGHLIGHT_AND_OUTER_SHADOW;
  start_y_of_outer_shadow = cbw->primitive.highlight_thickness +
                            MARGIN_BETWEEN_HIGHLIGHT_AND_OUTER_SHADOW;
  width_of_outer_shadow  = cbw->core.width - (2 * start_x_of_outer_shadow);
  height_of_outer_shadow = cbw->core.height - (2 * start_y_of_outer_shadow);
  outer_shadow_thickness = 3;

   switch (state) {
     case XmDEFAULT_READY:
       /* Push out the margins to make room for subsequent increases in
          the shadow thickness. The request to push out the margins will
          increase the size of the CommandButton widget assuming that its
          manager has the space to spare. */

          if (cbw->primitive.shadow_thickness < 5)
            margin_push_out = 5;
          else
            margin_push_out = cbw->primitive.shadow_thickness;
          margins_were_pushed_out = 1;
          XtVaSetValues((Widget)cbw,
             XmNmarginWidth,  cbw->simple.margin_width + margin_push_out,
             XmNmarginHeight, cbw->simple.margin_height + margin_push_out,
             NULL);
          break;
     case XmDEFAULT_ON:
       /* Draw an outer shadow. The outer shadow is drawn outside the
          widget’s margins but inside the border highlight.
          The inner shadow is drawn by the DrawShadow method. */
          top_GC = cbw->primitive.top_shadow_GC;
          bottom_GC = cbw->primitive.bottom_shadow_GC;
          outer_shadow_type = cbw->command_button.visual_armed ?
```

```
                                      XmSHADOW_ETCHED_IN:
                                      XmSHADOW_ETCHED_OUT;
              XmeDrawShadows(XtDisplay(w), XtWindow(w),
                             top_GC,
                             bottom_GC,
                             start_x_of_outer_shadow,
                             start_y_of_outer_shadow,
                             width_of_outer_shadow,
                             height_of_outer_shadow,
                             outer_shadow_thickness,
                             outer_shadow_type);
              break;
         case XmDEFAULT_OFF:
           /* Erase the outer shadow when the widget is no longer the
              default. */
              XmeClearBorder(XtDisplay(w), XtWindow(w),
                             start_x_of_outer_shadow,
                             start_y_of_outer_shadow,
                             width_of_outer_shadow,
                             height_of_outer_shadow,
                             outer_shadow_thickness);
              break;
         case XmDEFAULT_FORGET:
         default:
           /* The widget is not a potential default button. If XmDEFAULT_FORGET
              is called at some point after XmDEFAULT_READY was called, then
              we have to restore the margins back to their original size. */
              if (margins_were_pushed_out)
                XtVaSetValues((Widget)cbw,
                  XmNmarginWidth,  cbw->simple.margin_width - margin_push_out,
                  XmNmarginHeight, cbw->simple.margin_height - margin_push_out,
                  NULL);
              break;
        }
    }
```

11.3 Writing Your Own Menu Button Widget

The **ExmMenuButton** demonstration widget is a subclass of **ExmString**. Figure 11-2 shows its position in the hierarchy.

Figure 11–2. ExmMenuButton Position in Hierarchy



Like an **ExmString**, an **ExmMenuButton** can display one compound string. However, **ExmMenuButton** provides the additional ability to function as a menu child widget. That is, an **ExmMenuButton** can be a child of an **XmRowColumn** whose **XmNrowColumnType** resource is set to something other than **XmWORK_AREA**.

Before getting too deeply into menu buttons, we want to remind you that you should not write an original menu button widget. If you feel the standard Motif menu button widgets and gadgets do not meet your needs, then you should modify **ExmMenuButton**. For example, you might provide a *realize* method for **ExmMenuButton** that creates circular-shaped menu buttons. Another possibility is to extend the visuals of **ExmMenuButton** so that each menu button simultaneously displays both a pixmap and some text.

The following subsections detail selected features of **ExmMenuButton**.

11.3.1 Traversal Translations

ExmMenuButton provides two different translation strings: **defaultTranslations** and **traversalTranslations**.

The **defaultTranslations** string is specified in the **tm_table** field of the **Core** class record. These translations handle activation events.

The **traversalTranslations** string is used in the *translations* field of the Primitive class record. These translations allow a user to move between menu choices. None of the action routines associated with the **traversalTranslations** string are defined by the **ExmMenuButton** widget. Therefore, the Intrinsics automatically search for the action routines in the parent of **ExmMenuButton**. The only possible parent of an **ExmMenuButton** widget is an **XmRowColumn** widget. As it happens, **XmRowColumn** defines all the **MenuTraverse** action routines.

11.3.2 The XmQTmenuSavvy Trait

Every menu button widget must install the *XmQTmenuSavvy* trait. The menu system widget (**XmRowColumn**) examines its children for this trait. If a child does not hold this trait, the child cannot be a child of the Motif menu system widget.

ExmMenuButton installs the *XmQTmenuSavvy* trait as part of its **ClassInitialize** method. The following call to **XmeTraitSet** does the installation:

```
XmeTraitSet(exmMenuButtonWidgetClass, XmQTmenuSavvy,
            (XtPointer) &menuSavvyTraitRec);
```

The third argument to **XmeTraitSet**, **menuSavvyTraitRec**, is defined earlier in the **MenuB.c** file as follows:

```
static XmConst XmMenuSavvyTraitRec menuSavvyTraitRec = {
  0,                                       /* Version */
  (XmMenuSavvyDisableProc) DisableCallback, /* disableCallback   */
  GetAccelerator,                          /* getAccelerator    */
  GetMnemonic,                             /* getMnemonic       */
  GetActivateCBName,                    /* getActivateCBName */
};
```

11.3.3 Installing Accelerators and Mnemonics

The **ExmMenuButton** widget supports accelerators and mnemonics. (See the *Motif 2.1—Programmer's Guide* for details on accelerators and mnemonics.) In brief, mnemonics and accelerators each provide a way for a user to activate a menu button without actually clicking on it. For example, the **AllExmDemo.c** demonstration program places both an accelerator and a mnemonic on the **ExmMenuButton** Quit button. The accelerator is **"Alt-q"**. Therefore, a user can activate the **ExmMenuButton** Quit button by pressing Alt-q even when the button is not visible. The mnemonic is "Q". Therefore, a user can activate the **ExmMenuButton** Quit button by pressing Q whenever the button is visible.

To support accelerators and mnemonics, **ExmMenuButton** does the following:

1. Provides appropriate resources; manages the resource values in **initialize** and **set_values**
2. Provides appropriate *XmQTmenuSavvy* trait methods
3. Provides appropriate visual support for accelerators and mnemonics in its **DrawVisual** method

The following subsections examine each of the preceding items.

11.3.3.1 Step 1: Provide Appropriate Resources

ExmMenuButton provides the following resources definitions in its *resources* array to support accelerators and mnemonics:

```
...
{
    XmNmnemonic,
    XmCMnemonic,
    XmRKeySym,
    sizeof(KeySym),
    XtOffsetOf( ExmMenuButtonRec, menu_button.mnemonic),
    XmRImmediate,
    (XtPointer) XK_VoidSymbol
},

{
    XmNaccelerator,
    XmCAccelerator,
    XmRString,
    sizeof(char *),
    XtOffsetOf(ExmMenuButtonRec, menu_button.accelerator),
    XmRImmediate,
    (XtPointer) NULL
},

{
    XmNacceleratorText,
    XmCAcceleratorText,
    XmRXmString,
    sizeof(XmString),
    XtOffsetOf(ExmMenuButtonRec, menu_button.accelerator_text),
    XmRImmediate,
    (XtPointer) NULL
},

{
    XmNmnemonicCharSet,
    XmCMnemonicCharSet,
    XmRString,
    sizeof(XmStringCharSet),
    XtOffsetOf(ExmMenuButtonRec, menu_button.mnemonic_charset),
    XmRImmediate,
    (XtPointer) XmFONTLIST_DEFAULT_TAG
}
...
```

11.3.3.2 Step 2: Provide XmQTmenuSavvy Trait Methods

The *XmQTmenuSavvy* trait contains two relevant trait methods: **getAccelerator** and **getMnemonic**. What follows is how *ExmMenuButton* implements these two trait methods:

```
static char*
GetAccelerator(Widget w)
{
  ExmMenuButtonWidget mw = (ExmMenuButtonWidget)w;

  return(mw -> menu_button.accelerator);
}


static KeySym
GetMnemonic(Widget w)
{
  ExmMenuButtonWidget mw = (ExmMenuButtonWidget)w;

  return(mw -> menu_button.mnemonic);
}
```

The menu system widget (**XmRowColumn**) calls these two trait methods to determine what the accelerator and mnemonic are.

11.3.3.3 Step 3: Provide Appropriate Visuals

ExmMenuButton must provide visuals that tell the user what the accelerator and mnemonic are. The visuals are rendered by the **DrawVisual** method of **ExmMenuButton**.

It is customary to identify the mnemonic by underlining one of the characters in the regular button text. The **DrawVisual** method of **ExmMenuButton** calls **XmStringDrawUnderline** to do the underlining.

If a button has an accelerator, the accelerator must be shown following the label of the button. After **DrawVisual** draws the regular button text, **DrawVisual** calls **XmStringDraw** to write the accelerator text.

11.3.4 The XmQTmenuSystem Trait

ExmMenuButton does not install the *XmQTmenuSystem* trait. Rather, **XmRowColumn** installs this trait, and **ExmMenuButton** calls many of its trait methods. We now consider several of those calls to *XmQTmenuSystem* trait methods.

When the cursor enters the window associated with an *ExmMenuButton*, the **ExmMenuButton** method is called. This method is responsible for giving focus to the **ExmMenuButton**, but only if the menu system is in drag mode. (When the menu system is in drag mode, each menu button child will automatically be highlighted as it gains keyboard focus.)

In order to do this, the **MenuButtonEnter** method must ask its parent if it holds the *XmQTmenuSystem* trait. The following code illustrates this:

```
{
ExmMenuButtonWidgetClass wc = (ExmMenuButtonWidgetClass)XtClass(w);
ExmMenuButtonWidget mw = (ExmMenuButtonWidget)w;
XmMenuSystemTrait menuSTrait;
int status;

menuSTrait = (XmMenuSystemTrait)
  XmeTraitGet((XtPointer) XtClass(XtParent(w)), XmQTmenuSystem);
```

```
if (! menuSTrait)
  return;
```

Assuming that the parent is a menu system widget, the **MenuButtonEnter** method must now find out what state the widget is in. To do so, the **MenuButtonEnter** method must call the *status* trait method as follows:

```
status = menuSTrait->status(w);
```

The *status* trait method returns a bit mask into the *status* variable. The **Xm/MenuT.h** header file provides several macros for probing the returned bit mask. The **MenuButtonEnter** method calls the **XmIsInDragMode** macro to determine if the menu system is in drag mode as follows:

```
if ((((ShellWidget) XtParent(XtParent(mw)))->shell.popped_up) &&
  XmIsInDragMode(status)) {
```

If the menu system is in drag mode, then the **ExmMenuButton** must grab the keyboard focus by calling the **childFocus** trait method of *XmQTmenuSystem* as follows:

```
menuSTrait -> childFocus(w);
```

11.4 Writing Your Own Tab Button

An application program can attach one or more tab children to an **XmNotebook** widget. In most applications, the tab children are usually **XmLabel** widgets, but nothing prevents widget writers from creating their own tab children widgets. The **ExmTabButton** demonstration widget is one possible tab child widget.

All tab children widgets should install the *XmQTjoinSide* trait. Widgets holding this trait can affix themselves directly onto the side of another widget. Widgets with this trait look almost as if they have been melded onto the side of another widget. This appearance is especially useful for tab buttons, so the **ExmTabButton** widget installs the *XmQTjoinSide* trait.

Widgets holding the *XmQTjoinSide* trait usually have a somewhat irregular shape. This irregular shape makes it much harder for a widget writer to draw suitable window decorations (border highlights and shadows). In other words, since the window is not a standard rectangular shape, the border highlights and shadows cannot be rectangular either. See the **BorderHighlight**, **BorderUnhighlight**, and **DrawShadows** routines of **ExmTabButton** for details.

Chapter 12
Geometry Management

The Intrinsics provide the basic mechanisms and policies that underlay geometry management in Motif. This chapter describes the Intrinsics support for managing geometry within widgets. This chapter also explains the geometry management tools that Motif provides for widget writers.

The fundamental principle of geometry management is that parent widgets control the geometry of their children. Despite this principle, child widgets are not as powerless as it might seem. For example, a child widget may request a geometry change from its parent, or a parent widget may ask a child widget for its opinion on an impending geometry change.

12.1 Overview

Your widget needs to manage geometry when it encounters any of the following situations:

- Your widget is initially realized.

- An application calls **XtSetValues** to change one of the geometry resources in your widget.
- The user resizes a shell containing one of your widgets.
- An application resizes one of the parents of your widget.
- An application resizes one of the children managed by your widget.

In addition to these typical cases, an application may also generate a geometry request by calling one of the Xt geometry management routines. Most of these routines require a response from your widget.

The Intrinsics provide the five *geometry resources* shown in the following table. The Intrinsics monitor the values of these resources in a variety of ways. For example, the **XtSetValues** call automatically monitors the five geometry resources and takes certain actions when they change.

Table 12–1. Geometry Resources of Core

Field	**Resource**	**Meaning**
core.border_width	**XmNborderWidth**	Specifies the width of the border that surrounds the widget’s window on all four sides. The width is specified in pixels. A width of 0 means that no border shows.
core.height	**XmNheight**	Specifies the inside height (excluding the border) of the widget’s window.
core.width	**XmNwidth**	Specifies the inside width (excluding the border) of the widget’s window.
core.x	**XmNx**	Specifies the x-coordinate of the upper-left outside corner of the widget’s window. The value is relative to the upper-left inside corner of the parent’s window.
core.y	**XmNy**	Specifies the y-coordinate of the upper-left outside corner of the widget’s window. The value is relative to the upper-left inside corner of the parent’s window.

The following seven **Core** class methods handle most of the work of geometry management:

- **change_managed** (of the parent widget)
- **geometry_manager** (of the parent widget)
- **initialize** (of the child widget)
- **set_values** (of the child widget)
- **set_values_almost** (of the child widget)
- **resize** (of the child widget)
- **query_geometry** (of the child widget)

Each of these methods is examined in the following subsections.

12.2 The Parent's changed_managed Method

The Intrinsics call a parent widget's **change_managed** method when either of the following happens:

- One of its unmanaged children becomes managed.
- One of its managed children becomes unmanaged.

When one of these things happen, the parent widget often must move or resize some of its children. The parent can

- Move a child by calling **XtMoveWidget**
- Resize a child by calling **XtResizeWidget**
- Move and/or resize a child by calling either **XtConfigureWidget** or **XmeConfigureObject**

Of these four routines, we recommend using **XmeConfigureObject** because it is the only routine that knows how to update the appropriate drag and drop fields. Each of the four functions update the appropriate geometry resources of the child and, if the child is realized, reconfigure the child's window.

The **change_managed** method of a Motif manager widget must call the **XmeNavigChangeManaged** routine. The **XmeNavigChangeManaged** function establishes the correct keyboard traversal policy for all the children of the manager. Typically, your widget calls **XmeNavigChangeManaged** at the end of the **change_managed** method.

The following code from the **ExmGrid** demonstration widget illustrates how to write a **change_managed** method:

```
static void
ChangeManaged(
        Widget w
)
{
 Dimension gridWidth, gridHeight;
 ExmGridWidgetClass gwc = (ExmGridWidgetClass) XtClass(w);

    if (!XtIsRealized(w))  {
  /* If the user or application has set an initial (creation)
     size, honor it. If the user or application has not set
     an initial size, XtWidth and XtHeight will return 0. */
        gridWidth = XtWidth(w);
        gridHeight = XtHeight(w);
    } else {
  /* Otherwise, just force width and height to 0 so that CalcSize
     will recalculate the appropriate size. */
        gridWidth = 0;
        gridHeight = 0;
    }

  /* The CalcSize method determines the ideal size of Grid. */
    if (gwc->grid_class.calc_size)
        (*(gwc->grid_class.calc_size))(w, NULL, &gridWidth, &gridHeight);
    else
        CalcSize (w, NULL, &gridWidth, &gridHeight);

 /* Ask parent of Grid if Grid’s new size is acceptable. Keep asking until
    parent returns either XtGeometryYes or XtGeometryNo. */
    while (XtMakeResizeRequest (w, gridWidth, gridHeight,
                                &gridWidth, &gridHeight) == XtGeometryAlmost);
    /* Now that we have a size for the Grid, we can layout the children
       of the grid. */
    if (gwc->grid_class.layout)
        (*(gwc->grid_class.layout))(w, NULL);
    else
        Layout (w, NULL);

    /* Update keyboard traversal. */
    XmeNavigChangeManaged (w);
}
```

12.3 The Parent's geometry_manager Method

A managed child widget can make a geometry request of a realized parent by calling **XtMakeGeometryRequest**. The Intrinsics automatically call **XtMakeGeometryRequest** on a parent when an application changes a geometry resource of one of its children. Any call to **XtMakeGeometryRequest** automatically invokes the parent's **geometry_manager** method. The arguments of the **geometry_manager** method are the same as those to **XtMakeGeometryRequest**.

The child passes the specifics of its geometry request inside the second argument to **XtMakeGeometryRequest**. The **geometry_manager** method must respond to this geometry request in one of the following three ways:

- Grant the child's request by returning **XtGeometryYes**.
- Deny the child's request by returning **XtGeometryNo**.
- Suggest a compromise to the child by returning **XtGeometryAlmost**.

If the **geometry_manager** method can grant the request, the **geometry_manager** method must do the following:

- Update the appropriate geometry resources of the child.
- Return **XtGeometryYes**.

If **geometry_manager** returns **XtGeometryYes**, then **XtMakeGeometryRequest** will take care of changing the geometry of the child's window. (In other words, the

geometry_manager method does not have to do it.) The child is responsible for calling its own *resize* method to handle the new size.

The **geometry_manager** methods for all standard Motif widgets return **XtGeometryYes**, not **XtGeometryDone**. All **geometry_manager** methods of widgets that you write should return **XtGeometryYes**.

The **geometry_manager** method may be able to satisfy some, but not all, of a child’s request. For example, it may be able to grant the requested width, but not the requested height. In this case, the **geometry_manager** method may offer the child a compromise geometry. It fills in the reply *XtWidgetGeometry* structure with the parameters it intends to allow, and it sets the corresponding bit in the reply bitmask for any parameter it intends to change from the value requested. If the child immediately makes another geometry request by using the compromise parameters, the **geometry_manager** method must grant the request.

Following is the first part of the **GeometryManager** method of the **ExmGrid** demonstration widget:

```
static XtGeometryResult
GeometryManager (
        Widget w,  /* instigator */
        XtWidgetGeometry *request,
        XtWidgetGeometry *reply
)
{
 ExmGridWidget gw = (ExmGridWidget) XtParent(w);
 XtWidgetGeometry parentRequest;
 XtGeometryResult result;
 Dimension curWidth, curHeight, curBW;
 ExmGridWidgetClass gwc = (ExmGridWidgetClass) XtClass((Widget)gw);

  /* If the request was caused by ConstraintSetValues reset the flag */
    if (gw->grid.processing_constraints) {
      gw->grid.processing_constraints = False;
        /* The ConstraintSetValues added one to border_width;
           This is the Xt trick used to fire the GM when a non core
           geometry resource (like a constraint) changes.
           now take it away. */
        request->border_width -= 1;
    }

  /* Save the original child resources. */
    curWidth = w->core.width;
    curHeight = w->core.height;
    curBW = w->core.border_width;

  /* Deny any requests for a new position. */
    if ((request->request_mode & CWX) || (request->request_mode & CWY))
        return XtGeometryNo;

   if (request->request_mode & CWWidth)
     w->core.width = request->width;
   if (request->request_mode & CWHeight)
     w->core.height = request->height;
   if (request->request_mode & CWBorderWidth)
     w->core.border_width = request->border_width;

  /* Calculate a new ideal size based on these requests. */
  /* Setting width and height to 0 tells CalcSize to override these
     fields with the calculated width and height. */
    parentRequest.width = 0;
    parentRequest.height = 0;
    if (gwc->grid_class.calc_size)
        (*(gwc->grid_class.calc_size))((Widget)gw, w,
                                   &parentRequest.width,
                                   &parentRequest.height);
    else
        CalcSize ((Widget)gw, w, &parentRequest.width,
                                   &parentRequest.height);
```

Often a parent widget must change its own geometry in order to satisfy a child's request. The parent's **geometry_manager** method uses **XtMakeGeometryRequest** to ask its own parent for a geometry change. If an **XtMakeGeometryRequest** request to the grandparent returns **XtGeometryYes**, the **geometry_manager** method of the grandparent must update the requesting parent's geometry fields and may resize other child, but it should not call the requesting parent's *resize* method. The parent may call its own *resize* method as long as that routine does not call the requesting child's *resize* method. The parent's **geometry_manager** method then returns **XtGeometryYes**.

Sometimes the parent needs to make a geometry request to its own parent just to find out whether the grandparent will accept a proposed change. For example, the parent may intend to offer a compromise geometry to the child but it must first determine whether the grandparent will allow the parent to change its own geometry in order to offer the compromise. In this case, the parent does not want the grandparent actually to make the proposed change; it just wants the grandparent to tell the parent whether the change is acceptable.

In making its own geometry request to the grandparent, the parent sets the **XtCWQueryOnly** bit in the request bitmask. The grandparent can return **XtGeometryYes**, but it must not actually change any of its children. The parent then returns **XtGeometryAlmost** to the child, along with its compromise parameters. If the child accepts the compromise, the parent repeats its request to the grandparent without setting **XtCWQueryOnly**. The grandparent should grant the parent’s request, and the parent can then grant the child’s request.

If the grandparent’s response is **XtGeometryAlmost** and the parent still wishes to offer a compromise to the child, it caches the grandparent’s reply and returns **XtGeometryAlmost** to the child. If the child accepts this compromise, the parent then makes another request of the grandparent, using the cached compromise parameters from the grandparent and without setting **XtCWQueryOnly** . The grandparent should grant the parent’s request, and the parent can then grant the child’s request.

The following code forms the second half of the **GeometryManager** method of **ExmGrid**:

```
/* Ask the Grid’s parent if new calculated size is acceptable. */
  parentRequest.request_mode = CWWidth | CWHeight;
  if (request->request_mode & XtCWQueryOnly)
      parentRequest.request_mode |= XtCWQueryOnly;
  result = XtMakeGeometryRequest ((Widget)gw, &parentRequest, NULL);

/*  Turn XtGeometryAlmost into XtGeometryNo. */
  if (result == XtGeometryAlmost)
    result = XtGeometryNo;

  if (result == XtGeometryNo ||
      request->request_mode & XtCWQueryOnly) {
      /* Restore original geometry. */
      w->core.width = curWidth;
```

```
        w->core.height = curHeight;
        w->core.border_width = curBW;
   } else {
       /* result == XtGeometryYes and this wasn’t just a query */
       if (gwc->grid_class.layout)
           (*(gwc->grid_class.layout))((Widget)gw, w);
       else
           Layout ((Widget)gw, w); /* Layout with this child as
                                      instigator, so that we don’t
                                      resize this child. */
   }

   return (result);
}
```

12.4 The initialize Method

The way you write the geometry management sections of an *initialize* method depends heavily on whether you are writing a primitive widget or a manager widget.

12.4.1 Primitive Widgets

If the value of **core.width** or **core.height** is **0**, then the **initialize** method of your primitive widget should generate a starting widget size. If the user or application has set a starting value for **XmNwidth** or **XmNheight** other than **0**, then the initialize method should honor the user’s or application’s preference for its starting size.

A parent will treat the child’s starting size as a suggestion. In other words, the parent may choose to honor or to ignore the child’s starting size, or possibly to compromise on the starting size.

For consistency with other Motif widgets, your widget should set the **core.width** field to **0** when you want the widget to calculate a new preferred width and the **core.height** field to **0** when you want the widget to calculate a new preferred height.

12.4.2 Manager Widgets

The**initialize** methods of most manager widgets seldom perform any geometry management. In general, the *initialize* method should not specify values for any of the **Core** geometry resources shown in Table 12-1. Typically, the **change_managed** method of your manager's parent will establish the starting position and dimensions of your manager.

12.5 The Child's set_values and set_values_almost Method

When a user or application invokes **XtSetValues** on a geometry resource, **XtSetValues** makes a geometry request. (The five geometry resources are shown in Table 12-1.) After invoking all the widget's **set_values** methods, **XtSetValues** checks for changes to any geometry resources. If any of those resources have changed, it sets their values to those in effect before **XtSetValues** was called and then makes a geometry request with the new values as the requested geometry parameters. If the geometry request returns **XtGeometryYes**, **XtSetValues** calls the widget's **resize** method. If the parent's **geometry_manager** method returns **XtGeometryDone**, **XtSetValues** does not call the widget's **resize** method.

If the geometry request returns **XtGeometryNo** or **XtGeometryAlmost**, **XtSetValues** calls the widget's **set_values_almost** method. The child's **set_values_almost** method determines whether to accept the compromise, reject the compromise, or request an alternate geometry change.

XtSetValues passes **set_values_almost** the request and reply *XtWidgetGeometry* structures. If the request returns **XtGeometryNo**, the bitmask in the reply structure is 0. The **set_values_almost** method can accept a compromise geometry by copying the reply parameters into the request structure. It can also construct another request by altering the request structure, or it can end the negotiation by setting the request bitmask to **0**. If the request bitmask is nonzero when the **set_values_almost** method returns, **XtSetValues** makes another geometry request and treats the result in the same way as for the original request.

A widget's **set_values** method can initiate a geometry request by changing any of the geometry resources. For example, if **XtSetValues** is invoked on a Label's text, the

set_values method can calculate how large the widget should be to contain the new text and then set the relevant geometry fields accordingly. The **set_values** method should not do any resizing itself; in particular, it should not resize any child widgets because the geometry request might be denied. Resizing is usually done in the widget's **resize** method. The widget's **set_values_almost** method may need to restore some widget state in the event the geometry request is denied.

12.6 The Child's resize Method

When a child's size changes, the Intrinsics automatically call the child's *resize* method. This call informs the child that its size has been changed. The child's **resize** method must make any internal changes necessary to conform to the new dimensions. If the child is itself a composite widget, its **resize** method can move or resize its own children.

A widget's **resize** method is invoked in the following circumstances:

- Whenever a parent calls **XtConfigureWidget**, **XtResizeWidget**, or **XmeConfigureObject** to resize the child
- By **XtSetValues** when the widget's geometry resources are changed and the resulting geometry request returns **XtGeometryYes**

In addition, a shell's **resize** method is invoked when the size of the shell is changed, often by a user through the window manager.

When the Intrinsics call the **resize** method, the widget's **Core** geometry resources contain the new position and dimensions. The **resize** method must take these resource values as given; the **resize** method cannot issue a geometry request. The **resize** method uses these resource values to recalculate the widget's layout.

The **resize** method of a manager widget should not do layout itself. Rather, a manager widget should provide a separate layout routine. This layout routine should take as an argument the child that is making the request (if any) so that the layout routine can avoid resizing that child.

12.6.1 What to Cut When Space is Limited

One of the responsibilities of a **resize** method is to determine what should be displayed when there is not enough space to display everything. Motif provides the following precedence recommendations for handling this situation:

1. Try to reduce or eliminate any unused space (white space) inside the widget. In many widgets, this unused space is caused by the widget margins. The purpose of the margins is to provide a break between the widget's visual and the widget's border decorations. When there is plenty of space inside the widget, the margins are helpful. However, when space becomes tight, the margins must be reduced or eliminated.
2. Try to reduce the size of the widget's visual. For example, if the widget's visual is some sort of geometric shape, your widget should maintain the shape but scale down its size. If the widget's visual is text, then portions of the text will have to be clipped.
3. Try to reduce the size of the widget's shadows.
4. Try to reduce the size of the border highlight. (This is a last resort; always try to maintain at least the border highlight.)

12.7 The child's query_geometry Method

When calculating its layout, a parent widget may take account of a child's preferred size and location. The parent uses **XtQueryGeometry** to inquire about a child's preferred geometry. The parent passes to **XtQueryGeometry** pointers to two **XtWidgetGeometry** structures, one containing the parameters that the parent intends to impose and the other containing the preferred parameters returned by the child.

When the parent calls **XtQueryGeometry**, the Intrinsics automatically call the **geometry_manager** method of the child. The child's **query_geometry** method is responsible for determining the widget's preferred geometry. The method stores the parameters into the return **XtWidgetGeometry** structure, setting corresponding bits in the bitmask for fields that it cares about. The **query_geometry** method must return one of the following values:

- If the parent's intended geometry is acceptable, the child's **query_geometry** method returns **XtGeometryYes**.

- If the parent's and child's parameters differ for some field that both widgets care about, or if the child has expressed interest in a field that the parent does not care about, it returns **XtGeometryAlmost**.
- If the child's preferred geometry is the same as its current geometry, it returns **XtGeometryNo**.

After the **query_geometry** method returns, **XtQueryGeometry** fills in any fields in the return **XtWidgetGeometry** structure that the child does not care about with the current values of the resources in the child widget. **XtQueryGeometry** returns the value returned by the **query_geometry** method.

Most composite widgets should call **XtQueryGeometry** whenever they intend to change the geometry of a child that is not in the process of making a geometry request. A **geometry_manager** method should not call **XtQueryGeometry** for the child making the request. For a widget making a geometry request, the requested geometry is the preferred geometry.

12.7.1 Handling Easy Query Geometry Requests

In many cases, the **query_geometry** method needs to consist of only two parts. The first part should calculate the widget's preferred width and height. The second part should pass the preferred dimensions to the **XmeReplyToQueryGeometry** convenience function.

Later on in this chapter, we'll see a sample **query_geometry** method that calls **XmeReplyToQueryGeometry**.

12.8 Exposure and Redisplay

A widget can recompute its layout in its **resize**, **set_values**, or **geometry_manager** method, but usually it does not actually generate the window contents in those methods. A widget usually regenerates its window contents in response to an **Expose** event, which causes the widget's **expose** method to be invoked. This method takes as arguments the widget, the event, and the set of rectangles to be redisplayed. Using the current state of the widget (including its geometry resources), the **expose** method generates the contents of either the affected rectangles or the window as a whole.

XmeConfigureObject, **XtConfigureWidget**, **XtResizeWidget**, and **XtMoveWidget** cause the server to generate **Expose** events when necessary. **XtMakeGeometryRequest** also causes the server to generate **Expose** events when the parent’s **geometry_manager** method returns **XtGeometryYes**.

The **expose** method of all Motif manager widgets should call the **XmeRedisplayGadgets** routine. This routine passes exposure events down to all the gadget children of the manager. For example, following is the entire **Redisplay** method of **ExmGrid**:

```
static void
Redisplay (
        Widget w,
        XEvent *event,
        Region region)
{
  /* Pass exposure event down to gadget children. */
    XmeRedisplayGadgets (w, event, region);
}
```

12.9 A Widget Case Study: ExmSimple

This section examines the geometry management portions of the **ExmSimple** widget. **ExmSimple** only has the capability of being a child widget; it cannot serve as a parent. Therefore, **ExmSimple** allows us to focus solely on the child side of geometry management.

The **ExmSimple** widget displays a simple visual that is either a rectangle or an oval.

12.9.1 Variables Influencing Geometry

In addition to the standard geometry resources of **Core**, the **ExmSimple** widget provides a few geometry fields of its own, as described in the following table:

Table 12–2. Geometry Variables of ExmSimple

Field	Meaning
simple.pref_width	Holds an integral value representing the widget's current preferred width
simple.pref_height	Holds an integral value representing the widget's current preferred height
simple.need_to_compute_width	Holds a Boolean value; if **True**, then the widget needs to renegotiate its width
simple.need_to_compute_height	Holds a Boolean value; if **True**, then the widget needs to renegotiate its height
need_to_reconfigure	Holds a Boolean value; if True, the widget needs to call the **Reconfigure** method (typically to calculate a new preferred width or height)

12.9.2 Widget Start Up

The Intrinsics automatically call the **Initialize** method of **ExmSimple** when **ExmSimple** is instantiated. The **Initialize** method determines whether or not the user or application has specified an initial value for width or height. The following code fragment examines the widget's height:

```
if (rw->core.height == FIND_NATURAL_SIZE) {
  nw->simple.need_to_compute_height = True;
}
else {
  nw->simple.need_to_compute_height = False;
  nw->simple.pref_height = rw->core.height;
  nw->core.height = rw->core.height;
}
```

(**Initialize** also contains parallel code for width.)

If the height is 0 (*FIND_NATURAL_SIZE* is a constant 0), then the user or application has not specified an initial value for height. In this case, **ExmSimple** will need to calculate its preferred height. To do so, **ExmSimple** sets its **need_to_compute_height** flag to True.

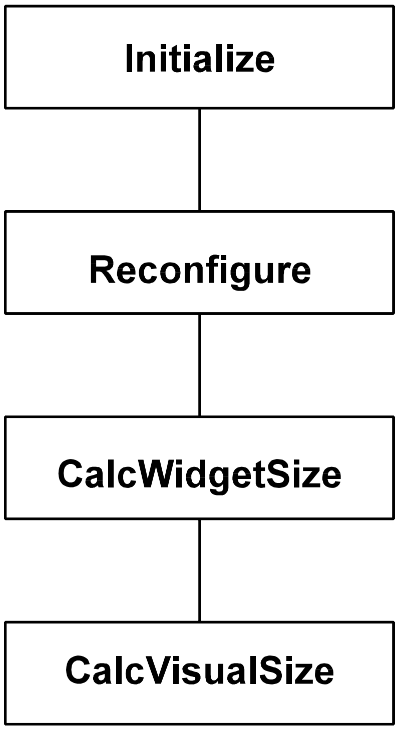
On the other hand, if the user or application has specified an initial value for height, then the Initialize method should honor those requests. The initialize method does this by setting both **simple.pref_height** and **core.height** to the user’s or application’s request. Note that changing the value of **core.height** will ultimately cause the Intrinsics to alert the parent of **ExmSimple**. In fact, the user or application’s requested starting size may not be honored by the parent of **ExmSimple**.

The **Initialize** method concludes by calling the **Reconfigure** method of **ExmSimple**.

12.9.3 Calculating Preferred Size

The **Reconfigure** method of **ExmSimple** calls the **CalcWidgetSize** method. The **CalcWidgetSize** method begins by calling the **CalcVisualSize** method. Therefore, the sequence of calls to this point is as follows:

Figure 12–1. Initialization Geometry Management Call Sequence



The **CalcVisualSize** method of **ExmSimple** is responsible for calculating the ideal size of the widget's visual. Note that **ExmSimple** provides no resources permitting the user any direct control over the size of the visual. For some widgets, the ideal visual size is fairly obvious. For example, the ideal size of the visual in *ExmString* is the extent of the string that is to be displayed. The ideal visual size of a widget that displays a pixmap would be the dimensions of the pixmap. For **ExmSimple**, though, the ideal visual size is not so obvious. After all, what is the ideal size of a rectangle or an oval? Does a 10-pixel-wide rectangle look better than a 12-pixel-wide rectangle? Since there is no true ideal size for a rectangle or oval, we have arbitrarily picked an ideal size of 30 pixels. The code in **CalcVisualSize** is as follows:

```
sw->simple.visual.height = IDEAL_SHAPE_SIZE;
```

After calculating visual size, the flow of control returns to the **CalcWidgetSize** method of **ExmSimple**. The **CalcWidgetSize** examines the **need_to_compute_height** flag.

This flag was set to either **True** or **False** back in the **Initialize** method. As you may recall, a value of **True** means that the user or application has not specified a starting height for **ExmSimple**.

Therefore, if **need_to_compute_height** is **True**, then **CalcWidgetSize** will need to calculate the preferred widget size.

The preferred widget size must take into account not only the preferred visual size but also the widget’s margins, shadows, and highlights. The following code does just that:

```
if (sw->simple.need_to_compute_height == True)
 sw->core.height = sw->simple.visual.height +
                    (2 * (sw->simple.margin_height +
                    sw->primitive.shadow_thickness +
                    sw->primitive.highlight_thickness));
```

If the **need_to_compute_height** flag is **False**, then the user or application has specified a starting height. **ExmSimple** needs to honor that preference by setting the **core.height** variable to the user’s preference:

```
else
  sw->core.height = sw->simple.pref_height;
```

Upon completing the calculations in **CalcWidgetSize**, flow of control returns to the **Reconfigure** method. The widget dimensions calculated by **CalcWidgetSize** become the new preferred dimensions of **ExmSimple**, as follows:

```
nw->simple.pref_height = nw->core.height
```

Then, **Reconfigure** conditionally calls the **Resize** method. For performance reasons, **Reconfigure** is careful to call **Resize** only once. Why is this important? Well, consider that **ExmSimple** is a superclass for several other widgets (including **ExmString**). The **initialize** method is chained in superclass-to-subclass order. Therefore, when **ExmString** is instantiated, the Intrinsics call the **Initialize** method of **ExmSimple** prior to calling the **Initialize** method of **ExmString**. Consequently, the **Reconfigure** method will be called twice, once by the **Initialize** method of **ExmSimple** and once again by the **Initialize** method of **ExmString**. Therefore, if **Reconfigure** were not careful, it would end up calling the **Resize** method twice, once by **ExmSimple** and once again by **ExmString**. The first call to **Resize** will be a waste of time. Therefore, in this case, it was more efficient to make sure that the only time **ExmSimple** called

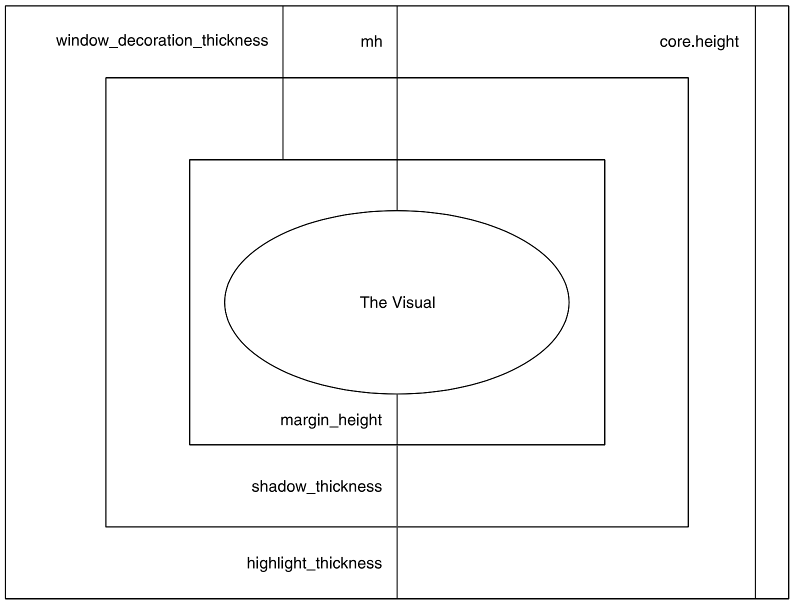
Resize was when **Reconfigure** had been called by the instantiated class itself and not by one of its chained superclasses.

12.9.4 Resize

The **Resize** method makes no attempts to change the size of the widget. Rather, the goal of **Resize** is to fit everything into the amount of space that has been allocated for it.

To help you visualize the relevant portions of the **ExmSimple** widget, we provide Figure 12-2.

Figure 12–2. A Labeled ExmSimple Widget



The following portion of the **Resize** method of **ExmSimple** implements the rules described in Section 12.6.1. That is, this code figures out whether or not there is enough room in the widget to display everything. If there is not enough room, something will have to be trimmed back or removed altogether.

```
window_decoration_thickness = sw->primitive.highlight_thickness  +
                              sw->primitive.shadow_thickness;
mh = window_decoration_thickness + sw->simple.margin_height;
total_target_widget_height = (2 * mh) + sw->simple.visual.height;

if (sw->core.height >= total_target_widget_height) {
  /* We have enough space to display everything (the visual, the margins,
     and the border decorations). */
  sw->simple.visual.y = mh;
  sw->simple.visual.height = sw->core.height - (2 * mh);
}
else if (sw->core.height >
          ((2 * window_decoration_thickness) + sw->simple.visual.height)) {
  /* We do not have enough space to display everything, but we do have
     enough space to display the visual and the border decorations.
     The top and bottom margins will have to be reduced. */
  sw->simple.visual.y = (sw->core.height - sw->simple.visual.height)/2;
}
else if (sw->core.height > 2 * window_decoration_thickness) {
 /* Space is very tight. We will eliminate the top and right margins
    all together. Furthermore, we will reduce the size of the visual. */
  sw->simple.visual.y = window_decoration_thickness;
  sw->simple.visual.height = sw->core.height -
                             (2 * window_decoration_thickness);
}
  else {
 /* We do not have enough space to display even one pixel of the visual. */
  sw->simple.visual.height = 0;
}
```

Notice that the **resize** method does not actually render any pixels onto the screen. Instead, **resize** calculates the dimensions of the widget's visuals. Another routine (**DrawVisual**) will do the rendering.

12.9.5 Handling Size Changes

When an application calls **XtSetValues** to change a resource value in **ExmSimple**, the **set_values** method of **ExmSimple** must handle the change request. The geometry management code in the **set_values** method of **ExmSimple** is similar to the geometry management code in its **initialize** method. In both methods, the goal is to examine the value of **core.width** and **core.height**. If either is **0**, then a preferred value for that dimension will have to be calculated. On the other hand, if the application has set the value of **core.width** or **core.height** to some value other than 0, then **set_values** must make that new value the widget's preferred size.

The **set_values** method of **ExmSimple** calls the **Reconfigure** method, just as the **initialize** method of **ExmSimple** does. The only difference is that the **initialize** method always calls **Reconfigure**, but the **set_values** method only calls **Reconfigure** if a reconfiguration is needed.

12.9.6 Handling Geometry Queries

The **ExmSimple** widget must supply a **query_geometry** method to handle geometry queries from its parents. This method must return **XtGeometryYes**, **XtGeometryNo**, or **XtGeometryAlmost**.

ExmSimple is not concerned how its parents might change **x**, **y**, and **border_width**. In fact, of the geometry resources, the only two that concern **ExmSimple** are *width* and *height*. Therefore, we can provide a **query_geometry** method that relies on **XmeReplyToQueryGeometry**. Prior to calling **XmeReplyToQueryGeometry**, the widget has to have calculated its preferred size.

Here is the entire **query_geometry** method of **ExmSimple**:

```
QueryGeometry (
        Widget widget,
        XtWidgetGeometry *parent_request,
        XtWidgetGeometry *child_reply)
{
 ExmSimpleWidget sw = (ExmSimpleWidget) widget;

   if (!XtIsRealized(widget)) {   /* Simple has not yet been realized. */
     child_reply->width  = XtWidth(widget);   /* might be 0 */
     child_reply->height = XtHeight(widget);  /* might be 0 */
   } else {                       /* Simple has been realized. */
     child_reply->width  = sw->simple.pref_width;
     child_reply->height = sw->simple.pref_height;
   }

 /* Return ExmSimple’s preferred size */
   return XmeReplyToQueryGeometry(widget, parent_request, child_reply);
}
```

12.9.7 Redisplay

The **Redisplay** method of **ExmSimple** triggers the redisplay (or original display) of all visuals components in the widget. **ExmSimple** widgets consist of three visual components:

- A rectangle or arc
- The shadow
- The border highlights

ExmSimple calls its **DrawVisual** method to draw the rectangle or arc. **ExmSimple** then calls its **DrawShadow** method to render its shadows. Finally, **ExmSimple** calls the *expose* method of **XmPrimitive** to draw the border highlight.

Chapter 13

UIL and WML Compatibility

You must provide a way for Motif applications programmers to instantiate your new widget. Chapter 2 explained how to make widgets accessible to C applications. We now explain how to make your new widgets accessible to a User Interface Language (UIL) application. If you have no need of making your widgets accessible to a UIL application, you can skip this chapter.

UIL is a programming language designed for rapid prototyping of Motif interfaces. A UIL program consists mainly of definitions of the application's widget hierarchy. That is, a UIL program tells the Motif Resource Manager (MRM) which widgets to instantiate. (For details on writing UIL applications, see the *Motif 2.1—Programmer's Reference*.)

Motif vendors typically provide a UIL compiler that "understands" only those widgets in the Motif toolkit. Fortunately, UIL is an extensible language. This means that, when you write a new widget, you can expand the UIL language to make the compiler understand your new widget.

13.1 Strategies

There are three ways to make your new widget accessible to UIL programmers:

- You can provide your UIL customers with two small header files and one MRM initialization file. These files will contain sufficient information for the UIL compiler to compile your customer’s UIL programs. Of the three ways, this is the easiest to implement. However, this way has some liabilities when it comes to type checking. That is, the UIL compiler will not detect certain coding problems in your customers’ UIL programs.
- You can provide your customers with a Widget Meta-Language Database (WMD) file. The UIL compiler reads the WMD file at runtime and processes the new or modified widget definitions dynamically. This is somewhat harder than the previous solution; however, an updated WMD file will allow UIL to do type checking.
- You can provide your customers with a new UIL compiler that understands your new widget(s) This solution is the hardest of the three ways to implement. Furthermore, it provides no real advantages in performance over the second solution. Therefore, we do not recommend doing this. See the *Motif Release Notes* for information on building a UIL compiler.

Even if you choose the second or third mechanism, you still have to provide your customers with most of the files described in the first mechanism. For example, if you provide your customers with a WMD file, you must also provide them with one of the header files and an MRM initialization file.

13.2 Providing UIL Access Through Three Small Files

Probably the simplest way to make your new widget accessible to UIL applications is to provide the following three files:

- A UIL creation header file
- An MRM initialization file
- A header file for the MRM initialization file

Providing these files permits users to instantiate your new widget from their UIL applications.

One important advantage of this mechanism is that it is portable. That is, the header file should work without modification on any UIL platform. One big disadvantage is that the UIL compiler will not be able to do complete data type checking on UIL applications.

The following subsections detail these files and then explain how a UIL application can access them.

13.2.1 The UIL Creation Header File

The UIL creation header file specifies the names of the new widget(s) and new resources in a format that MRM will be able to read. Typically, you create one UIL creation header file that describes all the widgets that you have created. You can find a sample UIL creation header file for the Exm sample widget set in the **demos/programs/Exm** directory under the filename **Exm.uil**. Following are the contents of this file:

```
!*****************************************************************************
!*
!*  Exm.uil -  Exm widgets UIL creation header file
!*
!*****************************************************************************


!* Exm Creation API

procedure
    ExmCreateSimple();
    ExmCreateString();
    ExmCreateStringTransfer();
    ExmCreateCommandButton();
    ExmCreateMenuButton();
    ExmCreateGrid();
    ExmCreateTabButton();
    ExmCreatePanner();
```

```
!* Exm Resources

value
    ExmNsimpleShape: private argument (’simpleShape’, integer);
    ExmNcompoundString: private argument (’compoundString’,
                                             compound_string);
    ExmNgridMarginWidthWithinCell: private argument
                         (’gridMarginWidthWithinCell’, integer);
    ExmNgridMarginHeightWithinCell: private argument
                         (’gridMarginHeightWithinCell’, integer);
    ExmNopenSide: private argument (’openSide’, integer);
    ExmNreportCallback: private argument (’reportCallback’, callback);
    ExmNrubberBand: private argument (’rubberBand’, boolean);
    ExmNcanvasWidth: private argument (’canvasWidth’, integer);
    ExmNcanvasHeight: private argument (’canvasHeight’, integer);
    ExmNsliderX: private argument (’sliderX’, integer);
    ExmNsliderY: private argument (’sliderY’, integer);
    ExmNsliderWidth: private argument (’sliderWidth’, integer);
    ExmNsliderHeight: private argument (’sliderHeight’, integer);

value
    ExmSHAPE_OVAL: 0; ExmSHAPE_RECTANGLE: 1;
```

The procedure section of the header file lists the convenience creation functions of all eight Exm widgets. The first *value* section describes the new resources of the Exm widgets. (Resource names already used in the Motif toolkit need not be listed in the *value* section.) The second *value* section associates numerical constants with the enumerated constants of the **ExmNsimpleShape** resource.

If you create your own WMD file, you do not need to create the UIL file.

See the reference page for **UIL(5X)**for more information.

13.2.2 MRM Initialization File

Before a UIL application can use one of your new widgets, the widgets must be registered with the MRM. You register a widget by calling the **MrmRegisterClass** function. The easiest way to register a group of widgets is to pack all the

MrmRegisterClass calls into one convenience function. For example, the MRM initialization file for all the Exm widgets is stored online in file **demos/lib/Exm/ExmMrm.c**. Following are its contents:

```
#include <stdio.h>
#include <Xm/Xm.h>
#include <Mrm/MrmPublic.h>
#include <Exm/Simple.h>
#include <Exm/String.h>
#include <Exm/StringTrans.h>
#include <Exm/CommandB.h>
#include <Exm/MenuB.h>
#include <Exm/Grid.h>
#include <Exm/TabB.h>
#include <Exm/Panner.h>

/**********************************************************************
 *
 * ExmMrmInitialize - register Exm widget classes with Mrm
 *
 **********************************************************************/

int ExmMrmInitialize()
{
    MrmRegisterClass (MrmwcUnknown, "ExmSimple",
                        "ExmCreateSimple", ExmCreateSimple,
                        exmSimpleWidgetClass);
    MrmRegisterClass (MrmwcUnknown, "ExmString",
                        "ExmCreateString", ExmCreateString,
                        exmStringWidgetClass);
    MrmRegisterClass (MrmwcUnknown, "ExmStringTransfer",
                        "ExmCreateStringTransfer", ExmCreateStringTransfer,
                        exmStringTransferWidgetClass);
    MrmRegisterClass (MrmwcUnknown, "ExmGrid",
                        "ExmCreateGrid", ExmCreateGrid,
                        exmGridWidgetClass);
    MrmRegisterClass (MrmwcUnknown, "ExmCommandButton",
                        "ExmCreateCommandButton", ExmCreateCommandButton,
                        exmCommandButtonWidgetClass);
    MrmRegisterClass (MrmwcUnknown, "ExmMenuButton",
                        "ExmCreateMenuButton", ExmCreateMenuButton,
                        exmMenuButtonWidgetClass);
    MrmRegisterClass (MrmwcUnknown, "ExmTabButton",
                        "ExmCreateTabButton", ExmCreateTabButton,
                        exmTabButtonWidgetClass);
    MrmRegisterClass (MrmwcUnknown, "ExmPanner",
                        "ExmCreatePanner", ExmCreatePanner,
                        exmPannerWidgetClass);
    return (0);
}
```

This file includes the widget public header files of all eight Exm widgets.

Creating an MRM initialization file is not a requirement; it is only a convenience for UIL applications programmers. The only requirement for a UIL application is that **MrmRegisterClass** gets called for each new widget.

13.2.3 Header File for the MRM Initialization File

If you create an MRM initialization file, then you should also create a header file for it. The header file will provide a convenient handle for UIL applications programs to access your MRM initialization file. We provide such a header file in file **ExmMrm.h**. The only code in the file is as follows:

```
int ExmMrmInitialize(void);
```

13.2.4 Accessing Widgets from a UIL Application

After creating the three files described earlier in this section, UIL applications programs can access the new widgets. We provide a sample UIL application in the directory **demos/programs/Exm/app_in_uil**. This code looks very much like any other UIL application. In fact, there are only two differences between this UIL application and a UIL application that uses the standard Motif widget set.

First, the UIL application must include the UIL creation header file; for example:

```
include file ("Exm.uil");
```

Second, the UIL callback file must invoke the registration function defined in the MRM initialization file. To make this work, the UIL callback file must also include the appropriate header file. For example, the UIL callback file **app_in_uil.c** contains the following code:

```
#include <ExmMrm.h>

MrmInitialize ();  /* standard Motif widget set */
ExmMrmInitialize(); /* Exm widget set */
```

13.3 What is WML?

Another way to make your new widgets accessible to UIL applications is to create a new UIL compiler that understands your new widgets. Motif provides the Widget Meta-Language (WML) facility to help you do this.You can use WML in two ways:

- Compile a customized WML file into a Widget Meta-Language Database (WMD) file. The UIL compiler reads this file at runtime and processes the new or modified widget definitions dynamically.
- Build a new UIL compiler by running the WML facility with a customized WML file.

This section explains how to write an appropriate WML file. Later sections of this chapter will explain how to build a new UIL or WMD from your WML file.

The WML facility generates the components of the UIL compiler that can change depending on the widget set. WML adds support in UIL for additional widgets that are not in the Motif standard widget set or for a totally new widget set.

UIL is made up of the following:

- Static syntax
- Dynamic syntax
- Data types

13.3.1 Static Syntax Elements

The static syntax elements are the basic syntax and keywords of UIL. These elements do not change when the widget set is modified. The static syntax elements of UIL are defined in the file **Uil.y** in the WML source directory.

13.3.2 Dynamic Syntax Elements

The dynamic syntax elements are the parts of UIL that change with the widget set. These elements describe the widget and gadget classes supported by UIL, including their resources and hierarchy. The dynamic elements of UIL are defined in WML files. The **motif.wml** file (stored in the WML source directory) defines the dynamic elements of the standard Motif widget set. The **Exm.wml** file (in the **demos/lib/Exm/ wml** directory) defines the dynamic elements of the Exm demonstration widget set.

13.3.3 Data Types

The data type elements describe the allowable data types for each widget and gadget resource. Although the data types do not change, the resources that they are assigned to change with the widget set. The allowable data types for each resource are defined in the same file as the dynamic syntax elements.

The WML facility combines the static syntax, dynamic syntax, and data type elements to produce new source code for UIL. This allows a developer to modify the dynamic elements of UIL, adding resources, widgets, gadgets, or even new widget sets.

For more information on the syntax of WML files, see the **WML(5X)** reference page in the *Motif 2.1—Programmer's Reference*. The next section explores a sample WML file.

13.4 A WML File Example

The WML file that describes the standard widgets of the Motif toolkit is stored in the **tools/wml** directory in filename **motif.wml**. We also provide an example that demonstrates how to write your own WML file. This example WML file describes the Exm demonstration widget set. You can find this example WML file in the **demos/lib/Exm/wml** directory as **Exm.wml**.

WML filenames must have the **.wml** suffix.

13.4.1 A Sample #include Directive

Our goal is to produce a UIL compiler that can compile requests for two kinds of widgets:

- The standard widgets and gadgets of the Motif toolkit
- The Exm widget set

The easiest way to ensure that our new UIL compiler will understand all the standard widgets and gadgets is to specify the following line:

```
#include "motif.wml"
```

This line includes the descriptions of all the standard Motif widgets in the Motif toolkit.

13.4.2 A Sample ControlList

Use the **ControlList** directive to add the names of your widgets to various categories. Although you can specify any categories you want, you should at least place the appropriate widgets in the following four categories:

- **AllWidgetsAndGadgets** contains the name of every widget or gadget.
- **AllWidgets** contains the name of every widget. (It does not include the name of any gadget.)
- **MenuWidgetsAndGadgets** contains the name of any widget or gadget that can be in a menu. In the standard widget set, this includes obvious choices like **XmPushButton** and somewhat less obvious widgets like **XmSeparator**. Your widget should go into this list if it can be a child of a RowColumn that has an

XmNrowColumnType resource of anything except **XmWORK_AREA**. In the Exm demonstration widget set, only **ExmMenuButton** meets this criteria.

- **ManagerWidgets** contains the name of every manager widget. In other words, any subclass of **XmManager** should be in this category.

For example, here is the complete **ControlList** for **Exm.wml**:

```
ControlList
        AllWidgetsAndGadgets
                {
                ExmSimple;
                ExmString;
                ExmCommandButton;
                ExmMenuButton;
                ExmStringTransfer;
                ExmPanner;
                ExmGrid;
                ExmTabButton;
                };
        AllWidgets
                {
                ExmSimple;
                ExmString;
                ExmCommandButton;
                ExmMenuButton;
                ExmStringTransfer;
                ExmPanner;
                ExmGrid;
                ExmTabButton;
                };
        MenuWidgetsAndGadgets
                {
                ExmMenuButton;
                }
        ManagerWidgets
                {
                ExmGrid;
                };
```

13.4.3 An EnumerationSet Example

If any of your new widget resources require an enumerated value, you must specify them within an **EnumerationSet** section. The **EnumerationSet** section defines the set of legal enumerated constants for enumerated resources. For example, the Exm demonstration widget contains a resource, *ExmNsimpleShape*, which requires enumerated constants. Therefore, the **EnumerationSet** section appearing in **Exm.uil** looks as follows:

```
EnumerationSet
        SimpleShape: integer
                { ExmSHAPE_OVAL; ExmSHAPE_RECTANGLE; };
```

Resource names should be stripped of their prefix. Therefore, the *ExmNsimpleShape* resource is specified as **SimpleShape**. The data type will always be **integer**. The values inside the braces are the set of legal enumerated constants for the resource.

13.4.4 A Resource Example

Your WML file must contain one or more **Resource** sections to describe the "new" resources supported by your widget. A new resource is one that is not defined by another **Resource** section. If your WML file includes the standard Motif WML file (**motif.wml**), then all the standard Motif resources will already have been defined for you. In this case, you should not specify any standard Motif resources within the **Resource** sections of your own WML file.

Consider the resource set of the **ExmString** widget as defined within the **String.c** file. Although the *resources* array of **String.c** defines five resources, four of these resources are part of the standard widget set. Therefore, the only resource that needs to be described in a **Resource** section is **ExmNcompoundString**. Following is the appropriate code:

```
Resource
        ExmNcompoundString: Argument
                { Type = compound_string; };
```

Constraints must be labeled with the keyword **Constraint**. For example, the description of the two constraints of the **ExmGrid** widget follows:

```
Resource
        ExmNgridMarginWidthWithinCell: Constraint
                { Type = integer; };
        ExmNgridMarginHeightWithinCell: Constraint
                { Type = integer; };
```

You should beware of a few resource and constraint naming rules. WML will issue an error message if you attempt to use the same name as both a resource and a constraint. For example, the previous code fragment specified *ExmNgridMarginWidthWithinCell* as a **Constraint**. If a subsequent **Resource** section lists *ExmNgridMarginWidthWithinCell* as an **Argument** (that is, as a resource), then WML will issue an error message. You should also be careful when reusing a name. If you do reuse a name, you must ensure that the **Type** is the same for both definitions. For example, given that you have already declared *ExmNcompoundString* as a **compound_string**, WML will issue an error message if you try to redefine *ExmNcompoundString* as an **integer**..

13.4.5 Class Example

The **Class** definition can supply a rich variety of information about your widget class. At the very least, each **Class** definition should probably contain definitions for **SuperClass**, **ConvenienceFunctions**, **WidgetClass**, and **Resources**. For example, following is the **Class** definition for *ExmString*:

```
Class
        ExmString: Widget
                {
                SuperClass = ExmSimple;
                ConvenienceFunction = ExmCreateString;
                WidgetClass = ExmString;
                Resources {
                        XmNtraversalOn;
                        ! New
                        ExmNcompoundString;
                        XmNrenderTable;
                        XmNalignment;
                        XmNrecomputeSize;
                        };
                };
```

The **Resources** modifier contains the names of all resources defined in the *XtResource* array of **ExmString**. If a resource appears in the *XtResource* array in order to override the default value established by a superclass, then the resource should appear above the "**! New**" comment. All other resources should appear below the "**! New** "comment. For example, of the five resources in the *XtResource* array of **ExmString**, only **XmNtraversalOn** appears above " **! New**" since it is a resource of a superclass (**XmPrimitive**).

13.5 Building a WMD File from Your WML File

This section explains how to create a WMD file from your WML file. The easiest way to build a new WMD file is to make minor modifications within the example directory that we provide. This example directory is at pathname **demos/lib/Exm/wml**. Inside this directory, you will find an **Imakefile**, our sample WML file (**Exm.wml**), and several other necessary files.

The **Imakefile** we provide creates a WMD file named **Exm.wmd** from the **Exm.wml** file. If you want to create **Exm.wmd**, you must issue the following sequence of commands:

```
cd demos/widgets/Exm/wml
make Makefile
make includes
make depend
make
```

The resulting **Exm.wmd** file can now be used by a UIL compiler. (See the next section for details.)

To build a WMD file from your own WML file, you should first copy your WML file into the **demos/lib/Exm/wml** directory.

Second, change two lines of the **Imakefile** so that it will build a WMD file from your WML file rather than from **Exm.wml**. The two lines to change are the definitions of the *TABLE* and *WMDTABLE* variables. Before you edit the **Imakefile**, these definitions look as follows:

```
TABLE = Exm.wml
WMDTABLE = Exm.wmd
```

You should change these variable assignments to reflect the names of your WML and WMD files. For example, suppose your WML file was stored in file **My.wml**. In this case, you would change the *TABLE* and *WMDTABLE* assignments as follows:

```
TABLE = My.wml
WMDTABLE = My.wmd
```

By convention, the *WMDTABLE* variable should have the same root name as the *TABLE* variable, but it should have the suffix *wmd* instead of *wml*.

Next, you must modify the **wmldbcreate.c** file stored inside the **demos/lib/Exm/wml** directory. The only part of the **wmldbcreate.c** file that you need to modify is the section containing the names of the user-supplied public header files for your widget. This section currently contains the names of all eight public header files for the Exm widget set, as follows:

```
#include <Exm/Simple.h>
#include <Exm/String.h>
#include <Exm/StringTrans.h>
#include <Exm/CommandB.h>
#include <Exm/MenuB.h>
#include <Exm/Panner.h>
#include <Exm/Grid.h>
#include <Exm/TabB.h>
```

All you have to do is to replace these lines with the names of your widget’s public header files.

You build a WMD file by issuing the following command sequence:

```
cd demos/lib/Exm/wml
make Makefile
make includes
make depend
make
```

The resulting **My.wmd** file can now be used by a UIL compiler.

13.6 UIL Compiling with Your WMD File

By default, the UIL compiler bases its compilation on the default WMD file stored in **motif.wmd**. An application programmer needing to use your WMD file instead of the default must do one of the following:

- Specify the **−wmd** option to the **uil** command
- Supply values for the *database* and **database_flag** members of the **Uil_command_type** structure whose address is the first argument to the UIL function

For example, given a WMD file named **My.wmd** and a UIL application name **app.uil**, the application programmer can issue a compilation command like the following:

```
uil -wmd My.wmd -o app.uid app.uil
```

Note that the application programmer must also have access to the header files and MRM initialization files described earlier in this chapter.

Chapter 14

Miscellany

This chapter describes miscellaneous topics regarding Motif widget writing.

14.1 Internationalization

When writing a widget, you should consider how the presented information will be perceived by different users around the world. Writing a truly internationalized widget requires considerable expertise.

One of the most important parts of writing an internationalized widget is in allowing for different layout directions. The desired layout direction is held by the **XmNlayoutDirection** resource of **XmPrimitive** and **XmManager**.

XmNlayoutDirection also affects the layout direction of text. In some languages, text must be displayed from left to right; other languages display text from right to left; and in some Asian languages, text is displayed vertically (from top to bottom). For this reason, all Motif widgets that display text need to base text direction on the value of the **XmNlayoutDirection** resource. The **XmPrimitive** and **XmManager** widgets will set an appropriate default value for **XmNlayoutDirection** based on the locale. Your task as a widget writer is to have your widget read the value of **XmNlayoutDirection** and render it appropriately.

14.2 Binary Compatibility

In order to make a widget binary compatible with future releases of **XmPrimitive** and **XmManager** that have expanded their instance records, you must do the following:

1. Define an index macro in the widget private header file
2. Define **XmField** access macros in the widget source code file
3. Use the proper offset macros and data types when you declare resources and constraints in the widget source code file
4. Set the **version** field of the **CoreClassPart** to **XtVersionDontCheck**.
5. Declare an **XmOffsetPtr** variable in the widget source code file
6. Call **XmeResolvePartOffsets** when you initialize the widget, probably in the **class_initialize** method
7. Access all members of the widget instance record through the **XmField** macros you created in Step 2

By doing all these steps, a widget written and compiled at the current release of Motif should continue to work without recoding or recompiling at future releases. If you do not do these steps when writing a widget, it is probable that your widgets will have to be recompiled for future releases.

The only Exm widget that is binary compatible with future releases is the **ExmTabButton** widget.

14.2.1 Step 1: Define an Index Macro

The index macro represents your widget’s place in the hierarchy. The index macro is defined in the widget private header file. Use the following syntax to define the index macro:

```
#define YourWidgetNameIndex (SuperclassIndexMacro + 1)
```

For example, **XmPrimitive** is the superclass of **ExmSimple**. The **SuperclassIndexMacro** associated with **XmPrimitive** is called **XmPrimitiveIndex**. Therefore, the index macro of **ExmSimple** is defined as follows:

```
#define ExmSimpleIndex (XmPrimitiveIndex + 1)
```

Similarly, the index macro of **ExmString** is defined as follows:

```
#define ExmStringIndex (ExmSimpleIndex + 1)
```

Note that, although **ExmTabButton** is the only binary-compatible Exm widget, all the widgets that precede it in the widget hierarchy must define index macros.

14.2.2 Step 2: Define XmField Access Macros

To ensure binary compatibility, your widget source code file must define an **XmField** access macro for each field that your source code file accesses. In this context, a "field" means a member of a widget instance record in any subclass of **XmPrimitive** and **XmManager**. For example, **ExmTabButton** access the following three members of widget instance records of subclasses of **XmPrimitive**:

- **tab_button.open_side** (from **ExmTabButton**)
- **tab_button.draw_bevel** (from **ExmTabButton**)
- **command_button.visual_armed** (from **ExmCommandButton**)

Therefore, the source code file of **ExmTabButton** contains the following 5 macro definitions:

```
#define OpenSide(w)
    XmField(w, offsets, ExmTabButton, open_side, XtEnum)
#define JoinShadowThickness(w)
    XmField(w, offsets, ExmTabButton, join_shadow_thickness, Dimension)
#define VisualArmed(w)
    XmField(w, offsets, ExmCommandButton, visual_armed, Boolean)
#define SimpleShape(w)
    XmField(w, offsets, ExmSimple, simple_shape, unsigned char)
#define NeedToReconfigure(w)
```

```
    XmField(w, offsets, ExmSimple, need_to_reconfigure, Boolean)
```

14.2.3 Step 3: Declare Resources Properly

To ensure binary compatibility, you must define the *resources* array as an array of **XmPartResource** instead of an array of *XtResource*. In addition, the fifth field of each resource record should access offsets through the **XmPartOffset** macro instead of the **XtOffsetOf** macro.

For example, following is the *resources* array of **ExmTabButton**:

```
static XmPartResource resources[] = {
    {
     ExmNopenSide,
     ExmCOpenSide,
     ExmROpenSide,
     sizeof (XtEnum),
     XmPartOffset (ExmTabButton, open_side),
     XmRImmediate,
     (XtPointer) XmLEFT
   },
};
```

14.2.4 Step 4: Set version to XtVersionDontCheck

To ensure binary compatibility, you must set the **version** field of the **CoreClassPart** to **XtVersionDontCheck** instead of **XtVersion**. For example, the **CoreClassPart** of **ExmTabButton** contains the following **version** field:

```
/* version */                XtVersionDontCheck,
```

14.2.5 Step 5: Declare an XmOffsetPtr Variable

For binary compatibility, a widget must declare an **XmOffsetPtr** variable. By convention, the **XmOffsetPtr** is declared immediately following the declaration of trait record variables in the widget source code file.

For example, following is the declaration of the **XmOffsetPtr** variable that appears in **ExmTabButton**:

```
/* Part Offset table for XmResolvePartOffsets */
static XmOffsetPtr offsets;
```

14.2.6 Step 6: Call XmeResolvePartOffsets

Your widget must call **XmeResolvePartOffsets** prior to the first time that an **XmField** macro (defined in Step 2) is used in a widget method. In other words, in order for an **XmField** macro to work properly, **XmeResolvePartOffsets** must already have been called.

For example, following is the call to **XmeResolvePartOffsets** that appears in the **ClassInitialize** method of **ExmTabButton**:

```
XmeResolvePartOffsets(exmTabButtonWidgetClass, &offsets, NULL);
```

14.2.7 Step 7: Access Fields Through Macros

Instead of using any fields from widget instance parts directly, you must access them through the **XmField** access macros you created back in Step 2.

For example, the **Initialize** method of **ExmStringTransfer** accesses the **draw_bevel** field as follows:

```
DrawBevel(new) = False;  /* right */
```

rather than as follows:

```
nw->tab_button.draw_bevel = False;  /* wrong */
```

The latter way of accessing **draw_bevel** is adequate for creating a working widget but does not create a binary-compatible widget.

Chapter 15
Widget Printing

Because only a subset of the X protocol is supported by most X print servers (some GC functions, like **GXxor**, or things like **CopyArea**, **GetImage**, may not be supported), a widget implementation may run into problems when it is connected to a print server.

The list of requests that are guaranteed to be supported are documented in the X Print documentation. In addition, a fallback behavior is provided for those requests that are not fully supported (**GXor** drawing can resolve to **GXcopy**, *XGetImage* can return a white image, and so on). At the widget method level, one simple way to check whether or not a particular instance is running on a print server is to check if its shell root is an **XmPrintShell**.

Widget writers should bear these remarks in mind when writing or adapting their code. They should provide new resources to turn off visual characteristics that are appropriate for the screen representation of the widget, but usually inappropriate when the widget is printed on paper (for example, blinking and highlighting). Alternatively, they may turn off such resources automatically in the case where the widget is to be printed. (For information on scaling bitmaps and pixmaps for printing purposes, refer to the discussion of the *XmRBitmap* and *XmRDynamicPixmap* converters in Chapter 6.)

The knowledge of the **XmPrintShell** can also be used to optimize the widget code for printing only. In such cases, there is no need for drag-and-drop contexts, scrollbars, and the like.

The application printing model assumes that the application developer duplicates some widget instances for use in displaying them on the screen and printing them on paper (refer to *Motif 2.1—Programmer’s Guide* for a discussion of this subject). Widget writers are therefore encouraged to provide and document such resources as best define the content of their widgets for the benefit of application developers who use them.

In the following example, a simple widget Initialize method wants to know it is creating a widget for printing or video.

```
Initialize(widget...)
/*-------------*/
{
            Widget parent = widget;

    do {
                if (XmIsPrintShell(parent)) break;
    } while (parent = XtParent(parent));

    if (!parent) {
              /* not printing case, create our DragIcon */
      XmCreateDragIcon(...);
    } else {
      /* use GXcopy instad of GXxor in the graphics */
      XCreateGC(...);
}
```

Part 2

Widget-Writing Reference Pages

Chapter 16

Xme Reference Pages

This chapter contains reference pages for the internal Motif functions useful for writing Motif widgets.

XmeAddFocusChangeCallback

Purpose

Registers a callback for focus changes

Synopsis

#include <Xm/VendorSEP.h>

void XmeAddFocusChangeCallback(
  Widget *widget***,**
  XtCallbackProc *callback_procedure***,**
  XtPointer *data***);**

Description

XmeAddFocusChangeCallback registers a *callback_procedure* to be called whenever there is a focus change to any widget in the widget tree managed by a specified **VendorShell** or subclass of **VendorShell**. This *callback_procedure* is called the focus change callback. Motif will not call the focus change callbacks when the **VendorShell** (or subclass) is in implicit mode. In explicit mode, Motif automatically calls the focus change callbacks whenever the user or the application attempts to change focus. Your focus change callback procedure has the option of accepting or rejecting the attempted focus change.

Registering a focus change callback can cause an entire application to run more slowly because the focus change callbacks might be called fairly frequently.

A focus change callback can be removed by calling **XmeRemoveFocusChangeCallback**.

widget Specifies the widget whose children are to be monitored for focus changes. The specified *widget* must be a **VendorShell** or a subclass of **VendorShell**.

callback_procedure
Specifies the callback procedure to be called whenever there is a focus change.

data Specifies the call data to be passed as the *call_data* argument to the callback procedure.

Motif passes a pointer to an **XmFocusMovedCallbackStruct** to *callback_procedure*. When *callback_procedure* returns, Motif examines the *cont* field only.

```
typedef struct {
        int reason ;
        XEvent *event;
        Boolean cont;
        Widget old_focus;
        Widget new_focus;
        unsigned char focus_policy;
        XmTraversalDirection direction;
} XmFocusMovedCallbackStruct;
```

reason Indicates why the callback was invoked. Motif always sets this field to **XmCR_FOCUS_MOVED**.

event Points to the event that triggered the callback.

cont Indicates whether an attempted focus change will be allowed or rejected. A focus change callback should set *cont* to True (the default) to permit the focus change. A focus change callback should set *cont* to False to reject the focus change. Therefore, if you set *cont* to False, Motif will ensure that the focus stays at widget *old_focus*.

old_focus Indicates the widget ID of the widget that had keyboard focus immediately prior to the most recent traversal.

new_focus Indicates the widget ID of the widget that has just gotten keyboard focus.

focus_policy Indicates the **VendorShell**'s keyboard focus policy; this will always be **XmEXPLICIT** since *callback_procedure* only gets called in explicit mode.

direction Specifies the direction of the traversal. (See **XmProcessTraversal**(3) in the *Motif 2.1—Programmer's Reference*for details on possible values of the **XmTraversalDirection** enumerated type.)

Related Information

XmeRemoveFocusChangeCallback(3).

XmeClearBorder

Purpose

Clears the window decorations that border a given widget

Synopsis

#include <Xm/DrawP.h>

```
void XmeClearBorder(
        Display *display,
        Window window,
        Position x,
        Position y,
        Dimension width,
        Dimension height,
        Dimension thickness);
```

Description

XmeClearBorder clears the border highlight and/or shadows bordering a rectangular widget. To clear these window decorations, **XmeClearBorder** paints them in the widget’s background color.

The border highlight surrounds the perimeter of a widget. When calling **XmeClearBorder** to clear the border highlight, your widget should do the following:

- Set *x* and *y* to 0.
- Set *width* and *height* to the width and height of *window*.
- Set *thickness* to the highlight thickness. (If you are subclassing a primitive widget, then the highlight thickness is stored in the field **primitive.highlight_thickness**.)

The widget’s shadows lie within the border highlights. When calling **XmeClearBorder** to clear the shadows, your widget should do the following:

- Set *x* and *y* to the highlight thickness.
- Set *width* so that it equals the width of the window minus twice the highlight thickness.

- Set *height* so that it equals the height of the window minus twice the highlight thickness.
- Set *thickness* to the shadow thickness of the widget. (If you are subclassing a primitive widget, the shadow thickness is stored in the field **primitive.shadow_thickness**.)

display	Specifies the display on which *window* is rendered.
window	Specifies the window whose decorations are to be erased.
x	Specifies the x-coordinate in pixels of the left edge of whatever decoration is to be cleared.
y	Specifies the y-coordinate in pixels of the top edge of whatever decoration is to be cleared.
width	Specifies the width in pixels of the top edge of whatever decoration is to be cleared.
height	Specifies the height in pixels of the left edge of whatever decoration is to be cleared.
thickness	Specifies the thickness in pixels of whatever decoration is to be cleared.

Related Information

XmeDrawShadows(3) and **XmeDrawHighlight**(3).

XmeClipboardSink

Purpose

A toolkit function that transfers data from the clipboard to a widget

Synopsis

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

Boolean XmeClipboardSink(
        Widget widget,
        XtEnum op,
        XtPointer location_data);
```

Description

XmeClipboardSink transfers data from the clipboard to a widget.

This routine initializes an **XmDestinationCallbackStruct** as follows:

- Sets the *selection* member to *CLIPBOARD*
- Sets the *operation* member to the value of the *op* argument
- Sets the *location_data* member to the value of the *location_data* argument

The routine then makes the following sequence of calls:

1. Calls the destination widget’s **destinationPreHookProc** trait method, if any. **destinationPreHookProc** is one of the trait methods of the *XmQTtransfer* trait. **XmeClipboardSink** passes the initialized **XmDestinationCallbackStruct** as the *call_data* argument.
2. Calls any **XmNdestinationCallback** procedures that the application has attached to the destination widget.
3. Calls the destination widget’s **destinationProc** trait method, if any, after all transfers initiated by **XmNdestinationCallback** procedures have finished. However, if an **XmNdestinationCallback** procedure has called **XmTransferDone** with a status of **XmTRANSFER_DONE_DEFAULT**, **XmePrimarySink** does not call the **destinationProc** trait method.

It is the responsibility of the **XmNdestinationCallback** procedures and the **destinationProc** trait method to transfer any data to the widget.

widget Specifies the widget that is the destination for the data.

op Specifies the transfer operation. Possible values are **XmCOPY** and **XmLINK**.

location_data
Specifies information about the location where data is to be transferred. If the value is *NULL*, the data is to be inserted at the widget's cursor position. If *location_data* cannot fit inside an **XtPointer**, *location_data* must either be a static variable or be allocated. If *location_data* is allocated, a call must be made to **XmeTransferAddDoneProc** to establish a procedure to free the allocated memory. The value of *location_data* is only valid for the duration of a transfer. Once the transfer done procedures start to be called, *location_data* will no longer be stable.

Return Values

This function returns **False** if no transfers take place. Otherwise, it returns **True**.

Related Information

XmQTtransfer(3), **XmTransferDone**(3), **XmTransferValue**(3), **XmeClipboardSource**(3), **XmeConvertMerge**(3), **XmeDragSource**(3), **XmeDropSink**(3), **XmeGetEncodingAtom**(3), **XmePrimarySink**(3), **XmePrimarySource**(3), **XmeSecondarySink**(3), **XmeSecondarySource**(3), **XmeSecondaryTransfer**(3), **XmeStandardConvert**(3), **XmeStandardTargets**(3), and **XmeTransferAddDoneProc**(3).

XmeClipboardSource

Purpose

A toolkit function that places data on the clipboard

Synopsis

#include <Xm/Xm.h>

Boolean XmeClipboardSource(
    Widget *widget*,
    XtEnum *op*,
    Time *time*);

Description

XmeClipboardSource places widget data on the clipboard, using the Motif clipboard interface. If there is an owner of *CLIPBOARD_MANAGER*, calling **XmeClipboardSource** produces undefined results.

This reference page uses the term "the conversion routines associated with widget *widget*." Whenever you see that term, it means that **XmeClipboardSource** is making the following sequence of calls:

1. Calls the application's **XmNconvertCallback** procedures, if any.
2. Calls the source widget's **convertProc** trait method. (**convertProc** is one of the trait methods of the *XmQTtransfer* trait.) However, if any **XmNconvertCallback** procedure returns **XmCONVERT_DONE** or **XmCONVERT_REFUSE**, then **XmePrimarySource** will not call **convertProc**.

If the *selection* member is *CLIPBOARD* and the *target* member is _MOTIF_CLIPBOARD_TARGETS or _MOTIF_DEFERRED_CLIPBOARD_TARGETS, then **XmeClipboardSource** sets the value of the *parm* member of the **XmConvertCallbackStruct** to the value of the *op* argument.

XmeClipboardSource makes the following sequence of calls:

1. **XmeClipboardSource** asks the conversion routines associated with widget *widget* to convert the *CLIPBOARD* selection to _MOTIF_CLIPBOARD_TARGETS.

2. For each returned target, **XmeClipboardSource** asks the conversion routines associated with widget *widget* to convert the *CLIPBOARD* selection to that target. **XmeClipboardSource** then calls **XmClipboardCopy** to copy the converted data to the clipboard.
3. **XmeClipboardSource** asks the conversion routines associated with widget *widget* to convert the *CLIPBOARD* selection to _MOTIF_DEFERRED_CLIPBOARD_TARGETS.
4. If any of the conversion routines associated with widget *widget* return a target, **XmeClipboardSource** asks the widget to convert the *CLIPBOARD* selection to the _MOTIF_SNAPSHOT target. The responding conversion routine is expected to save a snapshot of the data and to return a distinguisher atom that uniquely identifies the snapshot.
5. **XmeClipboardSource** places each deferred target on the clipboard by name, using **XmClipboardCopy**.
6. If the *op* argument is **XmMOVE** and the data is successfully transferred, **XmeClipboardSource** asks the conversion routines associated with widget *widget* to convert the *CLIPBOARD* selection to the *DELETE* target.

In addition, **XmeClipboardSource** establishes a callback that is automatically called whenever a request is made to convert data to a deferred target. This callback is responsible for converting snapshot data to a particular target. Here is the sequence of calls that the callback makes:

1. The callback asks the conversion routines associated with widget *widget* to convert the snapshot data to that target, using as the *selection* the distinguisher atom returned from the _MOTIF_SNAPSHOT conversion.
2. The callback copies the converted data to the clipboard, using **XmClipboardCopyByName**.
3. When the snapshot data is no longer needed, the callback asks each of the conversion routines associated with widget *widget* to convert the snapshot to the *DONE* target, using the distinguisher atom as the selection.

The widget responding to _MOTIF_SNAPSHOT must own the snapshot atom.

The _MOTIF_SNAPSHOT target takes an optional parameter which should be used to distinguish the current selection. If the parameter is *NULL*, the conversion routine must allocate a distinguisher atom and assign it to the *selection* member of the **XmConvertCallbackStruct**.

widget	Specifies the widget that is the source for the data.
op	Specifies the transfer operation. Possible values are **XmMOVE**, **XmCOPY**, and **XmLINK**.
time	Specifies the time of the transfer. This is usually the timestamp from the event passed to an action routine. You should call *XtLastTimeStampProcessed* to generate the *time* value. If you set *time* to **CurrentTime** or 0, UTM will automatically change the call to *XtLastTimeStampProcessed*.

Return Values

This function returns **False** if the clipboard is locked or if no data is placed on the clipboard. Otherwise, it returns **True**.

Related Information

XmQTtransfer(3), **XmTransferDone**(3), **XmTransferValue**(3), **XmeClipboardSink**(3), **XmeConvertMerge**(3), **XmeDragSource**(3), **XmeDropSink**(3), **XmeGetEncodingAtom**(3), **XmePrimarySink**(3), **XmePrimarySource**(3), **XmeSecondarySink**(3), **XmeSecondarySource**(3), **XmeSecondaryTransfer**(3), **XmeStandardConvert**(3), **XmeStandardTargets**(3), and **XmeTransferAddDoneProc**(3).

XmeGetDesktopColorCells

Purpose

return desktop pixel data in **XColor[]** format

Synopsis

#include <Xm/ColorObjP.h>

Boolean XmeGetDesktopColorCells(
  Screen **screen*,
  Colormap *colormap*,
  XColor **colors*,
  int *n_colors*,
  int **n_colors_ret***);**

Description

The Motif library creates and maintains a **ColorObject** at application initialization time (first *VendorShell* creation) which holds color information coming from a Color Server running on the desktop (see Color Server protocol). An application that needs to use a private colormap should include in its colormap the desktop pixels returned by the Color Server protocol, plus the pixels used for the desktop icons, so that the rest of the desktop doesn't go "technicolor" when the application colormap is installed (and the desktop default colormap is de-installed). This function calls **XmeGetColorObjData(screen...)**, in order to get the raw pixel information and fill out the **XColor[]** color array for as many pixels as specified by the **colorUse** attributes, and then determine the RGB components for these pixel in the desktop colormap.

After it has retrieved the pixels of this screen's color object using *XmeGetColorObjData*, and added the pixels to the Color calculation cache in Motif (so that widgets created with a colormap set to share pixels get the right pixels for derived colors like shadows) this function calls *XQueryColors* on the default colormap to get the RGB intensity values for the desktop pixels in the XColor elements. It also sets all the flags to **(DoRed|DoGreen|DoBlue)** in the XColor array.

For the icon pixels, the function use *XParseColor* and *XAllocColor* on the default colormap using the 16 standard desktop icon color names (**"black", "white", "red", "green", "blue", "yellow", "cyan", "magenta", "#dedededede",**

"#bdbdbdbdbdbd", "#adadadadadad", "#949494949494", "#737373737373", "#636363636363", "#424242424242", "#212121212121") to find out the pixel ids that need to be shared.

XmeGetDesktopColorCells returns most-interesting-pixels first, and guarantees no duplicate pixel entries (which is not the same as duplicate RGB entries)

The order in the *XColor* array on return is:

- Foreground and background for all 8 palettes (primary, secondary, text, active, inactive, front-panel, ws buttons, in that order)
- Select color for primary, secondary
- The icon pixels (2 for *LOW_COLOR* and **B_W**, 16 for *HIGH* and *MEDIUM_COLOR*)
- **Topshadow** color for all 8 palettes
- **Bottomshadow** color for all 8 palettes (same order as above)
- Rest of select colors

The idea is that if an application has only, say, 12 pixels to spare in its colormap for the desktop, it should get shared pixels that allow most of the desktop visual to be usable: foreground and background colors first, mainly.

screen	X screen passed in.
colormap	Colormap for which the pixel are fetched
colors	X color structures returned (allocated by called).
n_colors	Size available in colors array.
n_colors_ret	Number of X colors elements filled by the function

Return Values

False if *XmeGetColorObjData* returns False, or if **colorUse** is *XmCO_BLACK_WHITE*.

Structures

XColor is defined in Xlib.

Environment

The information returned by this function depends on the presence of an active Color Server.

Resources

This function is affected by the resources set on the Color Server and useColorObj on XmScreen.

Actions/Messages

None.

Errors/Warnings

None.

Examples

After calling this function, the application can then add more entries to the **XColor[]** array (it can do that before in fact) and call *XStoreColors* with this **XColor[]** on the new colormap (note that if a program wants to use a private colormap and share the desktop pixels, a **ReadWrite** colormap must be used, since pixel have to be allocated at specific location, which is not possible with read-only cells.)

```
static void
SetColormap(Widget widget,
            int pixel_for_desktop)
{
    Colormap my_colormap;
    int ncolors_ret, i, p;
    XColor    colors[256];

    if (!XtIsRealized(widget)) return;

    /* create a readwrite colormap */
    my_colormap = XCreateColormap(XtDisplay(widget),
                                  XtWindow(widget),
                                  XDefaultVisual(XtDisplay(widget), 0),
                                  AllocAll);

    /* we need to keep track of which pixel are used, mark
       them all free before calling XmeGet... */
    for(i=0; i<256; i++) colors[i].pixel = -1;

    /* get the desktop pixels in the array */
    XmeGetDesktopColorCells (XtScreen(widget), my_colormap,
                             colors, pixel_for_desktop,
                             &ncolors_ret);

    /* then fill in the rest of our private colormap */
    for(i=ncolors_ret, p = 0; i<256; i++) {
                /* get a free pixel id */
                while (p <= 255 && colors[p].pixel != -1) p ++;
                colors[i].pixel = p;
                color[i].flags = DoRed|DoGreen|DoBlue;
        color[i].red = color[i].green = color[i].blue = i * 256;
    }

    XStoreColors(XtDisplay(widget), my_colormap, colors, 256);
    XtVaSetValues(widget, XmNcolormap, my_colormap, NULL);
}

main () {
    toplevel = XtAppInitialize(...);
    XtRealizeWidget(toplevel);

    SetColormap (toplevel, 40);
    ...
}
```

Related Information

dtsession(1), **dtstyle**(1), **XmeGetColorObjData**(3)

XmeGetColorObjData

Purpose

access ColorObject desktop and pixel data

Synopsis

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

Boolean XmeGetColorObjData(
        int *screen,
        int *coloruse,
        XmPixelSet *pixel_set,
        unsigned short pixel_set_size,
        short *active, *inactive, *primary, *secondary, *text);

typedef struct {
  Pixel fg;
  Pixel bg;
  Pixel ts;
  Pixel bs;
  Pixel sc;
} XmPixelSet;
```

Description

The Motif library creates and maintains a **ColorObject** at application initialization time (first **VendorShell** creation) which holds color information coming from a Color Server running on the desktop (see Color Server protocol, documented as part of the new revision of the X/Open XCSA specification).

This **ColorObject** is not directly available to programmers, but an API is provided to access most of the information it contains.

The **ColorObject** itself makes use of this information internally to default most color setting for the Motif widgets (e.g. it will add ***background: <PIXEL>** in the in memory resource database for the application)

This function returns color information maintained by the **ColorObject** in Motif, which comes from the color server using the Color Server protocol (see above sections) The *color_use* possible returned values are:

```
enum { XmCO_BLACK_WHITE,
       XmCO_LOW_COLOR,
       XmCO_MEDIUM_COLOR,
       XmCO_HIGH_COLOR };
```

which correspond to the type of monitor in use by the desktop.

There is a maximum of 8 pixel sets returned. Each pixel set consists of the five fundamental motif colors maintained by the **ColorObject** for this screen: **background**, **foreground**, **top_shadow_color**, **bottom_shadow_color**, and **select_color**.

The function also returns the color set id number used by the **ColorObject**.

screen	X screen passed in
color_use	colorUse type enum returned
pixel_set	Pixel sets returned (allocated by caller)
pixel_set_size	Size of pixel sets array (max used by function)
active, inactive, primary, secondary, text	Returned color set ids.

Return Values

False if the color server is not running, if the **useColorObj XmScreen** resource is set to False, or if the screen number is out of the range managed by the color server; otherwise True.

Structures

XmPixelSet is used to specify the pixel allocated by the Color Server.

Environment

The information returned by this function depends on the presence of an active Color Server.

Resources

This function is affected by the resources set on the Color Server and **useColorObj** on *XmScreen*.

Actions/Messages

None.

Errors/Warnings

None.

Examples

On return, one can use:

```
primary_background = pixelSet[primary_id].bg;
```

Related Information

dtsession(1), **dtstyle**(1)

XmeConfigureObject

Purpose

Changes a child’s position, size, or border width

Synopsis

#include <Xm/XmP.h>

void XmeConfigureObject(
    Widget *widget***,**
    Position *x***,**
    Position *y***,**
    Dimension *width***,**
    Dimension *height***,**
    Dimension *border_width***);**

Description

XmeConfigureObject is a Motif wrapper around the **XtConfigureWidget** call. Motif manager widgets should call **XmeConfigureObject** instead of *XtConfigureObject*. The **XmeConfigureObject** function can reconfigure Motif widgets or Motif gadgets. Furthermore, **XmeConfigureObject** automatically updates drop site information.

widget	Specifies the child widget or gadget to be reconfigured.
x	Specifies the starting x-coordinate of *widget* relative to its parent.
y	Specifies the starting y-coordinate of *widget* relative to its parent.
width	Specifies the new width of *widget*.
height	Specifies the new height of *widget*.
border_width	Specifies the border width of *widget*.

Related Information

XtConfigureWidget(3).

XmeConvertMerge

Purpose

A toolkit function that merges data converted during a transfer operation

Synopsis

#include <Xm/Xm.h>

void XmeConvertMerge(
  XtPointer *data***,**
  Atom *type***,**
  int *format***,**
  unsigned long *length***,**
  XmConvertCallbackStruct ****call_data*);**

Description

XmeConvertMerge merges data converted in the course of a transfer operation. The data to be merged is typically two list of targets. An **XmNconvertCallback** procedure in an application may supply some elements of the list, and the **convertProc** trait method of a widget may supply others. Therefore, the **convertProc** trait method will need to call **XmeConvertMerge** to merge the two target lists.

XmeConvertMerge can only be called from an **XmNconvertCallback** procedure or from a **convertProc** trait method.

A **convertProc** trait method usually calls **XmeConvertMerge** when an **XmNconvertCallback** procedure returns an **XmConvertCallbackStruct** structure containing a *status* of **XmCONVERT_MERGE**.

XmeConvertMerge uses **XtRealloc** to increase the allocated storage for the *value* member of the **XmConvertCallbackStruct** passed in the *call_data* argument. It then appends *data* to the data already present in the *value* member. The *type* and *format* passed as arguments to **XmeConvertMerge** must match the *type* and *format* fields passed in the **XmConvertCallbackStruct**.

data	Specifies the data to be added to the *value* member of the callback struct.
type	Indicates the type of *data*.

format	Specifies how the call should interpret *data*. You must specify 8, 16, or 32. A value of **8** implies that *data* is an array of *char*. A value of **16** implies that *data* is an array of *short*. A value of **32** implies that *data* is an array of *long*. It is possible that the specified number may not match the actual number of bits passed in *data*. For example, a value of 32 may actually correspond to a 64-bit data structure on some machines.
length	Specifies the number of elements in *data*, where each element has the number of bits specified by *format*.
call_data	Specifies a pointer to the **XmConvertCallbackStruct** that is to be modified.

Related Information

XmQTtransfer(3), **XmTransferDone**(3), **XmTransferValue**(3), **XmeClipboardSink**(3), **XmeClipboardSource**(3), **XmeDragSource**(3), **XmeDropSink**(3), **XmeGetEncodingAtom**(3), **XmePrimarySink**(3), **XmePrimarySource**(3), **XmeSecondarySink**(3), **XmeSecondarySource**(3), **XmeSecondaryTransfer**(3), **XmeStandardConvert**(3), **XmeStandardTargets**(3), and **XmeTransferAddDoneProc**(3).

XmeCreateClassDialog

Purpose

Creates a DialogShell containing a widget of the specified class

Synopsis

#include <Xm/DrawP.h>

Widget XmeCreateClassDialog(
  WidgetClass *widget_class***,**
  Widget *parent***,**
  String *name***,**
  ArgList *args***,**
  Cardinal *arg_count***);**

Description

XmeCreateClassDialog is a convenience function that creates a DialogShell and then instantiates widget *widget_class* inside the DialogShell. **XmeCreateClassDialog** names the created DialogShell *name_popup*.

XmeCreateClassDialog forces the value of the **XmNallowShellResize** resource of **Shell** to True.

Call **XtManageChild** to pop up the created DialogShell. You should set the *child* argument of **XtManageChild** to the **Widget** returned by **XmeCreateClassDialog**. Call **XtUnmanageChild** to pop down the DialogShell.

XmeCreateClassDialog registers a **destroyCallback** for the created widget so that its DialogShell parent can be automatically destroyed.

widget_class	Specifies the widget class you want to instantiate as a child of the DialogShell. The specified *widget_class* does not have to hold the *XmQTdialogShellSavvy* trait although, typically, it will hold this trait.
parent	Specifies the parent widget of the DialogShell.
name	Specifies the name of the created widget.

args	Specifies the argument list (*args*) passed to the DialogShell and the created widget.
arg_count	Specifies the number of attribute/value pairs in the argument list (*args*).

Return Values

Returns the child widget of the DialogShell.

Related Information

XmCreateDialogShell(3) and **XtCreateWidget**(3).

XmeDragSource

Purpose

A toolkit function that starts a drag and drop operation

Synopsis

#include <Xm/TransferP.h>

Widget XmeDragSource(
    Widget *widget***,**
    XtPointer *location_data***,**
    XEvent ***event*,**
    ArgList *args***,**
    Cardinal *arg_count***);**

Description

XmeDragSource begins a drag and drop operation from the specified widget.

This routine first asks the widget to convert the _MOTIF_DROP selection to _MOTIF_EXPORT_TARGETS. The returned list of targets becomes the initial value of the DragContext's **XmNexportTargets**, and the number of targets becomes the initial value of the DragContext's **XmNnumExportTargets**.

The *location_data* argument contains information about the location of the elements being dragged. If these consist of the widget's current selection, the value is *NULL*. Otherwise, the type and interpretation of the value are specific to the widget class. The *location_data* argument becomes the value of the DragContext's **XmNclientData** and of the *location_data* member of the **XmConvertCallbackStruct** for subsequent conversions of the data being dragged.

XmeDragSource sets the DragContext's **XmNconvertProc** to a function that asks the widget to convert data.

This routine calls **XmDragStart** with the widget specified in *widget*, the event specified in *event*, and the arguments specified in *args*. It overrides any settings of **XmNexportTargets**, **XmNnumExportTargets**, **XmNconvertProc**, and **XmNclientData** in *args*.

Whenever this routine or a subsequent drop operation asks a widget to convert data, it makes the following sequence of calls:

1. Calls the application's **XmNconvertCallback** procedures, if any.
2. Calls the source widget's **convertProc** trait method. However, if any **XmNconvertCallback** procedure returns **XmCONVERT_DONE** or **XmCONVERT_REFUSE**, then **XmePrimarySource** will not call **convertProc**.

widget Specifies the widget that is the source for the data.

location_data
Specifies the location of the elements being dragged. If these consist of the widget's current selection, the value is *NULL*. Otherwise, the type and interpretation of the value are specific to the widget class.

event Specifies the event that began the drag operation.

args Specifies arguments to be passed to **XmDragStart**.

arg_count Specifies the number of arguments in *args*.

Return Values

This function returns the widget ID of the **XmDragContext** returned by **XmDragStart**.

Related Information

XmDragStart(3), **XmQTtransfer**(3), **XmTransferDone**(3), **XmTransferValue**(3), **XmeClipboardSink**(3), **XmeClipboardSource**(3), **XmeConvertMerge**(3), **XmeDropSink**(3), **XmeGetEncodingAtom**(3), **XmePrimarySink**(3), **XmePrimarySource**(3), **XmeSecondarySink**(3), **XmeSecondarySource**(3), **XmeSecondaryTransfer**(3), **XmeStandardConvert**(3), **XmeStandardTargets**(3), and **XmeTransferAddDoneProc**(3).

XmeDrawArrow

Purpose

Draws a Motif-style, three-dimensional arrow

Synopsis

#include <Xm/DrawP.h>

void XmeDrawArrow(
  Display ******display***,**
  Drawable *drawable***,**
  GC *top_gc***,**
  GC *bottom_gc***,**
  GC *center_gc***,**
  Position *x***,**
  Position *y***,**
  Dimension *width***,**
  Dimension *height***,**
  Dimension *shadow_thickness***,**
  unsigned char *direction***);**

Description

XmeDrawArrow draws a Motif-style arrowhead surrounded by Motif-style shadows. The routine always draws the arrowhead as an equilateral triangle.

This function does not draw the tail of the arrow. Furthermore, this function does not allow you to draw a broad range of arrow shapes (for example, you cannot draw a curved arrow). You can, however, control the direction that the arrowhead faces.

Four of the arguments to **XmeDrawArrow** (*x*, *y*, *width*, and *height*) describe a bounding box. The bounding box encloses not only the arrowhead but its shadows as well. **XmeDrawArrow** centers the arrowhead within the bounding box.

display	Specifies the display on which the arrow is to be rendered.
drawable	Specifies the drawable in which the arrow is to be rendered. (Typically, this drawable is a widget window.)

top_gc	Specifies the graphics context of the shadow that is generally above and to the left of the triangular arrowhead.
bottom_gc	Specifies the graphics context of the shadow that is generally below and to the right of the triangular arrowhead.
center_gc	Specifies the graphics context of the arrow itself. If *center_gc* is *NULL*, the diamond will not be filled, but the shadow will still be drawn.
x	Specifies the leftmost point of the bounding box.
y	Specifies the top of the bounding box.
width	Specifies the width of the bounding box.
height	Specifies the height of the bounding box.
shadow_thickness	Specifies the thickness of the arrow's shadows in pixels. At Motif Release 2.0, the only supported values of *shadow_thickness* are 0 (no shadow), 1, or 2.
direction	Specifies the direction of the arrow. The direction must be one of the following values: **XmARROW_RIGHT**, **XmARROW_LEFT**, **XmARROW_UP**, or **XmARROW_DOWN**.

Related Information

XmArrowButton(3).

XmeDrawCircle

Purpose

Draws a Motif-style, three-dimensional circle

Synopsis

#include <Xm/DrawP.h>

void XmeDrawCircle(
    Display ***display**,
    Drawable *drawable***,**
    GC *top_gc***,**
    GC *bottom_gc***,**
    GC *center_gc***,**
    Position *x***,**
    Position *y***,**
    Dimension *width***,**
    Dimension *height***,**
    Dimension *shadow_thickness***,**
    Dimension *margin***);**

Description

XmeDrawCircle draws a Motif-style circle and the shadows around it. You specify the position of the circle by describing a bounding rectangle. The bounding rectangle encompasses not only the circle, but its shadows as well.

You may optionally specify a margin between the circle and its shadows.

display	Specifies the display on which the circle is to be rendered.
drawable	Specifies the drawable in which the circle is to be rendered.
top_gc	Specifies the graphics context of the circle's top shadow.
bottom_gc	Specifies the graphics context of the circle's bottom shadow.
center_gc	Specifes the graphics context of the circle itself. If you specify *NULL*, the circle will not be drawn, but the shadow will still be drawn.

x	Specifies the leftmost point of the bounding rectangle.
y	Specifies the top point of the bounding rectangle.
width	Specifies the width of the bounding rectangle.
height	Specifies the height of the bounding rectangle.
shadow_thickness	Specifies the thickness, in pixels, of the shadow surrounding the circle.
margin	Specifies the gap, in pixels, between the circle and its shadow. Without this margin, the boundary between a circle and its shadow may appear somewhat ragged on a monochrome screen.

Related Information

XmeDrawArrow(3), **XmeDrawDiamond**(3), **XmeDrawIndicator**(3), and **XmeDrawSeparator**(3)

XmeDrawDiamond

Purpose

Draws a Motif-style, three-dimensional diamond

Synopsis

#include <Xm/DrawP.h>

void XmeDrawDiamond(
    **Display ****display*,
    Drawable *drawable*,
    GC *top_gc*,
    GC *bottom_gc*,
    GC *center_gc*,
    Position *x*,
    Position *y*,
    Dimension *width*,
    Dimension *height*,
    Dimension *shadow_thickness*,
    Dimension *margin***);**

Description

XmeDrawDiamond draws a Motif-style diamond and the shadows around it. You specify the position of the diamond by describing a bounding rectangle. The bounding rectangle encompasses not only the diamond but its shadows as well.

You may optionally specify a margin between the diamond and its shadows.

display	Specifies the display on which the diamond is to be rendered.
drawable	Specifies the drawable in which the diamond is to be rendered.
top_gc	Specifies the graphics context of the diamond's top shadow.
bottom_gc	Specifies the graphics context of the diamond's bottom shadow.
center_gc	Specifes the graphics context of the diamond itself.

If *center_gc* is *NULL*, the diamond will not be filled, but the shadow will still be drawn.

x	Specifies the leftmost point of the bounding rectangle.
y	Specifies the top point of the bounding rectangle.
width	Specifies the width of the bounding rectangle.
height	Specifies the height of the bounding rectangle.
shadow_thickness	Specifies the thickness, in pixels, of the shadow surrounding the diamond.
margin	Specifies the gap, in pixels, between the diamond and its shadow. Without this margin, the boundary between a diamond and its shadow may appear somewhat ragged on a monochrome screen.

Related Information

XmeDrawArrow(3).

XmeDrawHighlight

Purpose

Draws a Motif-style highlight around a given widget to show that it has been selected

Synopsis

#include <Xm/DrawP.h>

void XmeDrawHighlight(
    Display ****display*,**
    Drawable *drawable***,**
    GC *gc***,**
    Position *x***,**
    Position *y***,**
    Dimension *width***,**
    Dimension *height***,**
    Dimension *highlight_thickness***);**

Description

XmeDrawHighlight draws a highlight rectangle immediately inside the boundaries of the specified drawable (which is typically a widget window). The highlight rectangle around a widget symbolizes that the widget has the keyboard focus.

display	Specifies the display on which *d* is rendered.
drawable	Specifies the drawable on which the border highlight is to be rendered. (Typically, this is a widget window.)
gc	Specifies the graphics context for the border highlight.
x	Specifies the x-coordinate in pixels of the leftmost point of the highlight box. Typically, the value of *x* will be 0.
y	Specifies the y-coordinate in pixels of the top point of the highlight box. Typically, the value of *y* will be 0.
width	Specifies the width in pixels of the highlight box. Typically, *width* will be the width of the widget.

height Specifies the height in pixels of the highlight box. Typically, *height* will be the height of the widget.

highlight_thickness
Specifies the thickness in pixels of the highlight box.

Related Information

XmeClearBorder(3).

XmeDrawIndicator

Purpose

Draws a Motif-style indicator (either a check mark or a cross) at the specified location in the drawable

Synopsis

#include <Xm/DrawP.h>

void XmeDrawIndicator(
  Display ****display*,**
  Drawable *drawable***,**
  GC *gc***,**
  Position *x***,**
  Position *y***,**
  Dimension *width***,**
  Dimension *height***,**
  Dimension *margin***,**
  XtEnum *indicator_type***);**

Description

XmeDrawIndicator draws an indicator inside the specified drawable. The *indicator_type* argument determines whether the indicator will be a check mark or a cross. The arguments *x*, *y*, *width*, and *height* define a bounding box; the indicator is rendered inside this bounding box. The *margin* argument determines the offset of the indicator from the sides of the bounding box.

display	Specifies the display.
drawable	Specifies the drawable on which the indicator is to be drawn. Typically, the drawable will be a widget window.
gc	Specifies the graphics context used to render the indicator.
x	Specifies the x-coordinate, in pixels, of the leftmost point of the bounding box.

y Specifies the y-coordinate, in pixels, of the top point of the bounding box.

width Specifies the width, in pixels, of the bounding box.

height Specifies the height, in pixels, of the bounding box.

margin Specifies the size of the blank space between the sides of the bounding box and the indicator. For example, a *margin* of 0 would tell **XmeDrawIndicator** to draw the indicator so that its sides touched the sides of the bounding box.

indicator_type
Specifies the kind of indicator to render. You must specify one of the following:

XmINDICATOR_CHECK
Draws a check mark.

XmINDICATOR_CROSS
Draws a cross.

Related Information

XmeDrawSeparator(3).

XmeDrawPolygonShadow

Purpose

Draws a Motif-style, three-dimensional shadow around a polygon

Synopsis

#include <Xm/DrawP.h>

void XmeDrawPolygonShadow(
    Display ****display*,**
    Drawable *drawable***,**
    GC *top_gc***,**
    GC *bottom_gc***,**
    XPoint ****points*,**
    int *point_count***,**
    Dimension *shadow_thickness***,**
    unsigned char *shadow_type***);**

Description

XmeDrawPolygonShadow places a Motif-style shadow around a drawn polygon or around a polygon-shaped widget.

display	Specifies the display on which *drawable* is rendered.
drawable	Specifies the drawable (typically, a widget window) around which the shadow will be drawn.
top_gc	Specifies the graphics context for the shadows appearing on the upper-left portions of the polygon.
bottom_gc	Specifies the graphics context for the shadows appearing on the bottom-right portions of the polygon.
points	Specifies the coordinates of each point in the bounding polygon.
point_count	Specifies the number of points in the bounding polygon.
shadow_thickness	Specifies the shadow thickness, in pixels.

shadow_type Specifies the kind of shadow to render. The possible values are as follows:

XmSHADOW_IN
Draws a shadow so that it appears inset. This means that the bottom shadow visuals and top shadow visuals are reversed.

XmSHADOW_OUT
Draws a shadow so that it appears outset.

XmSHADOW_ETCHED_IN
Draws a shadow using a double line giving the effect of a line etched into the window. The thickness of the double line is equal to the value of *shadow_thick*.

XmSHADOW_ETCHED_OUT
Draws a frame by using a double line to give the effect of a line coming out of the window. The thickness of the double line is equal to the value of *shadow_thick*.

Related Information

XmeDrawShadows(3).

XmeDrawSeparator

Purpose

Draws several different types of line-based separators

Synopsis

#include <Xm/DrawP.h>

void XmeDrawSeparator(
    Display ****display*,**
    Drawable *drawable***,**
    GC *top_gc***,**
    GC *bottom_gc***,**
    GC *separator_gc***,**
    Position *x***,**
    Position *y***,**
    Dimension *width***,**
    Dimension *height***,**
    Dimension *shadow_thickness***,**
    Dimension *margin***,**
    unsigned char *orientation***,**
    unsigned char *separator_type***);**

Description

XmeDrawSeparator draws a Motif-style, three-dimensional line to separate various components.

display	Specifies the display.
drawable	Specifies the drawable on which the separator will be rendered. Typically, the drawable will be a widget window.
top_gc	Specifies the graphics context of the top shadow on those separators that have shadows. (The separators have shadows when the *separator_type* is **XmSHADOW_ETCHED_IN**, **XmSHADOW_ETCHED_OUT**, **XmSHADOW_ETCHED_IN_DASH**, or **XmSHADOW_ETCHED_OUT_DASH**.) The definition of "top" depends on

the values of both the *separator_type* and the *orientation*. Suppose the *separator_type* is **XmSHADOW_ETCHED_IN** or **XmSHADOW_ETCHED_IN_DASH**. If the orientation is **XmHORIZONTAL**, then the top shadow is the shadow above the separator and the bottom shadow is the shadow below the separator. However, if the orientation is **XmVERTICAL**, then the top shadow is immediately to the left of the separator and the bottom shadow is immediately to the right of the separator. Now consider the results if the *separator_type* is **XmSHADOW_ETCHED_OUT** or **XmSHADOW_ETCHED_OUT_DASH**. If the orientation is **XmHORIZONTAL**, the top shadow is the shadow below the separator, and the bottom shadow is the shadow above the separator. However, if the orientation is **XmVERTICAL**, then the top shadow is immediately to the right of the separator, and the bottom shadow is immediately to the left of the separator.

bottom_gc Specifies the graphics context of the bottom shadow on those separators that have shadows. (See the description of *top_gc* for more information.)

separator_gc
Specifies the graphics context of the separator itself when the *separator_type* is **XmSINGLE_LINE**, **XmDOUBLE_LINE**, **XmSINGLE_DASHED_LINE**, or **XmDOUBLE_DASHED_LINE**.

x Specifies the x-coordinate of the start of the bounding box, in pixels. The **XmeDrawSeparator** function will draw the separator centered within the bounding box. However, the separator’s exact position depends not only on the bounding box but also on the *margin*. (See *margin* for details.)

y Specifies the y-coordinate of the start of the bounding box, in pixels. The separator will be drawn centered within the bounding box. However, the separator’s exact position depends not only on the bounding box but also on the *margin*. (See *margin* for details.)

width Specifies the width of the bounding box, in pixels.

height Specifies the height of the bounding box, in pixels.

shadow_thickness
Specifies the shadow thickness when *separator_type* is **XmSHADOW_ETCHED_IN_DASH**, **XmSHADOW_ETCHED_OUT_DASH**, **XmSHADOW_ETCHED_IN**, or

XmSHADOW_ETCHED_OUT. The value of *shadow_thickness* has no effect when the separator_type is **XmSINGLE_LINE**, **XmDOUBLE_LINE**, **XmSINGLE_DASHED_LINE**, **XmDOUBLE_DASHED_LINE**, or **XmNO_LINE**.

margin If the orientation is **XmHORIZONTAL**, *margin* specifies the gap between the left side of the bounding box and the left endpoint of the separator. The *margin* also specifies the gap, in pixels, between the right side of the bounding box and the right endpoint of the separator. If the orientation is **XmVERTICAL**, *margin* specifies the gap between the top side of the bounding box and the top endpoint of the separator. The *margin* also specifies the gap, in pixels, between the bottom side of the bounding box and the bottom endpoint of the separator.

orientation Specifies the orientation of the line, either horizontal (specify **XmHORIZONTAL**) or vertical (specify **XmVERTICAL**).

separator_type
Specifies the type of line and shadow to be used as a separator. The possible values are as follows:

XmSINGLE_LINE
Single line

XmDOUBLE_LINE
Double line

XmSINGLE_DASHED_LINE
Single-dashed line

XmDOUBLE_DASHED_LINE
Double-dashed line

XmNO_LINE
No line

XmSHADOW_ETCHED_IN
A line whose shadows give the effect of a line etched into the window. The thickness of the line is equal to the value of *shadow_thickness*.

XmSHADOW_ETCHED_OUT
A line whose shadows give the effect of an etched line coming out of the window. The thickness of the line is equal to the value of *shadow_thickness*.

XmSHADOW_ETCHED_IN_DASH
Identical to **XmSHADOW_ETCHED_IN**, except that a series of lines creates a dashed line.

XmSHADOW_ETCHED_OUT_DASH
Identical to **XmSHADOW_ETCHED_OUT**, except that a series of lines creates a dashed line.

Related Information

XmSeparator(3).

XmeDrawShadows

Purpose

Draws a Motif-style, three-dimensional shadow around a widget

Synopsis

#include <Xm/DrawP.h>

void XmeDrawShadows(
    **Display ****display*,
    Drawable *drawable*,
    GC *top_gc*,
    GC *bottom_gc*,
    Position *x*,
    Position *y*,
    Dimension *width*,
    Dimension *height*,
    Dimension *shadow_thickness*,
    unsigned char *shadow_type*);

Description

XmeDrawShadows places a three-dimensional, Motif-style shadow around a widget. This shadow is one of the fundamental visuals that gives Motif widgets their characteristic look and feel.

display	Specifies the display.
drawable	Specifies the drawable to draw the shadow around.
top_gc	Specifies the graphics context for the upper-left portion of the shadow.
bottom_gc	Specifies the graphics context for the bottom-right portion of the shadow.
x	Specifies the x-coordinate of the leftmost point of the shadow.
y	Specifies the y-coordinate of the top of the shadow.
width	Specifies the width of the shadow rectangle, in pixels.
height	Specifies the height of the shadow rectangle, in pixels.

shadow_thickness
Specifies the shadow thickness in pixels. The shadow is drawn from the outside in. The dimensions of the outside of the shadow are described by *x*, *yx*, *width*, and *height*.

shadow_type Specifies the kind of shadow to render. The possible values are as follows:

XmSHADOW_IN
Draws shadow so that it appears inset. This means that the bottom shadow visuals and top shadow visuals are reversed.

XmSHADOW_OUT
Draws shadow so that it appears outset.

XmSHADOW_ETCHED_IN
Draws shadow by using a double line to give the effect of a line etched into the window. The thickness of the double line is equal to the value of *shadow_thickness*.

XmSHADOW_ETCHED_OUT
Draws shadow by using a double line to give the effect of a line coming out of the window. The thickness of the double line is equal to the value of *shadow_thickness*.

Related Information

XmeDrawPolygonShadow(3).

XmeDropSink

Purpose

A toolkit function that establishes a widget as a drop site

Synopsis

#include <Xm/TransferP.h>

void XmeDropSink(
  Widget *widget***,**
  ArgList *args***,**
  Cardinal *arg_count***);**

Description

XmeDropSink establishes a widget as a drop site for drag and drop operations. This routine calls **XmDropSiteRegister** with the widget specified in *widget* and the arguments specified in *args*. It overrides any setting of **XmNdropProc** in *args*.

This routine sets the DropSite's **XmNdropProc** to a function that initializes an **XmDestinationCallbackStruct** as follows:

- Sets the *selection* member to _MOTIF_DROP.
- Sets the *operation* member to the following value:
  - **XmCOPY**, if the *operation* member of the **XmDropProcCallbackStruct** passed to the **XmNdropProc** is **XmDROP_COPY** and the **dropAction** member of the **XmDropProcCallbackStruct** is not **XmDROP_HELP**.
  - **XmMOVE**, if the *operation* member of the **XmDropProcCallbackStruct** passed to the **XmNdropProc** is **XmDROP_MOVE** and the **dropAction** member of the **XmDropProcCallbackStruct** is not **XmDROP_HELP**.
  - **XmLINK**, if the *operation* member of the **XmDropProcCallbackStruct** passed to the **XmNdropProc** is **XmDROP_LINK** and the **dropAction** member of the **XmDropProcCallbackStruct** is not **XmDROP_HELP**.

— **XmOTHER**, if the *operation* member of the **XmDropProcCallbackStruct** passed to the **XmNdropProc** is **XmDROP_NOOP** or if the **dropAction** member of the **XmDropProcCallbackStruct** is **XmDROP_HELP**.

- Sets the *destination_data* member to a pointer to the **XmDropProcCallbackStruct** passed to the **XmNdropProc**.
- Sets the *time* member to the **timeStamp** member of the **XmDropProcCallbackStruct** passed to the **XmNdropProc**.

XmeDropSink makes the following sequence of calls:

1. Calls the destination widget's **destinationPreHookProc** trait method, if any; **destinationPreHookProc** is one of the trait methods of the *XmQTtransfer* trait. **XmeDropSink** passes the initialized **XmDestinationCallbackStruct** as the *call_data* argument.
2. Calls any **XmNdestinationCallback** procedures that the application has attached to the destination widget.
3. Calls the destination widget's **destinationProc** trait method, if any, after all transfers initiated by **XmNdestinationCallback** procedures have finished. However, if an **XmNdestinationCallback** procedure has called *XmTransferDone* with a status of **XmTRANSFER_DONE_DEFAULT**, **XmePrimarySink** does not call the **destinationProc** trait method.

widget	Specifies the widget that is the drop site.
args	Specifies arguments to be passed to **XmDropSiteRegister**.
arg_count	Specifies the number of arguments in *args*.

Related Information

XmQTtransfer(3), **XmTransferDone**(3), **XmTransferValue**(3), **XmeClipboardSink**(3), **XmeClipboardSource**(3), **XmeConvertMerge**(3), **XmeDragSource**(3), **XmeGetEncodingAtom**(3), **XmePrimarySink**(3), **XmePrimarySource**(3), **XmeSecondarySink**(3), **XmeSecondarySource**(3), **XmeSecondaryTransfer**(3), **XmeStandardConvert**(3), **XmeStandardTargets**(3), and **XmeTransferAddDoneProc**(3).

XmeFocusIsInShell

Purpose

Returns a Boolean value to indicate whether the parent shell of a specified widget has the keyboard focus.

Synopsis

#include <Xm/XmP.h>

Boolean XmeFocusIsInShell(
    Widget *widget***);**

Description

XmeFocusIsInShell returns a Boolean value to indicate whether the parent shell of *widget* has the keyboard focus. This function is particularly useful for those programmers writing menu button widgets. (See the *ExmMenuButton* demonstration widget in the **demos/widgets/Exm/lib** directory for a sample usage of this function.)

widget	Specifies any widget.

Return Values

Returns True if the parent shell of *widget* has the keyboard focus. Otherwise, it returns False.

Related Information

ExmMenuButton(3).

XmeFromHorizontalPixels

Purpose

Converts from pixels to real-world dimensions based on horizontal resolution of the screen

Synopsis

#include <Xm/XmP.h>

void XmeFromHorizontalPixels(
    Widget *widget***,**
    int *offset***,**
    XtArgVal ****value*);**

Description

XmeFromHorizontalPixels converts a horizontal pixel value to a real-world dimensional unit. The real-world dimensional unit depends on the value of the **XmNunitType** resource associated with the widget. For example, suppose *widget* has an **XmNunitType** resource value of **Xm1000TH_INCHES**. If the input *value* were 200 pixels, the output *value* might be 2500 (2.5 inches) on one screen and 2000 (2.0 inches) on another screen.

XmeFromHorizontalPixels is an **XmExportProc**.

Typically, your widgets will call this function in a synthetic resource record.

widget	Specifies the widget.
offset	Specifies the offset (in bytes) of a synthetic resource field in the widget record.
value	Specifies a value in pixels and returns a value in the dimensional units used by *widget*.

Related Information

XmeToHorizontalPixels(3), **XmeFromVerticalPixels**(3), and **XmeToVerticalPixels**(3).

XmeFromVerticalPixels

Purpose

Converts from pixels to real-world dimensions based on vertical resolution of the screen

Synopsis

#include <Xm/XmP.h>

void XmeFromVerticalPixels(
  Widget *widget***,**
  int *offset***,**
  XtArgVal ****value*);**

Description

XmeFromVerticalPixels converts a vertical pixel value to a real-world dimensional unit. The real-world dimensional unit depends on the value of the **XmNunitType** resource associated with the widget. For example, suppose *widget* has an **XmNunitType** resource value of **Xm1000TH_INCHES**. If the input *value* were 200 pixels, the output *value* might be 2500 (2.5 inches) on one screen and 2000 (2.0 inches) on another screen.

XmeFromVerticalPixels is an **XmExportProc**.

Typically, your widgets will call this function in a synthetic resource record.

widget	Specifies the widget.
offset	Specifies the offset in bytes of a synthetic resource field in the widget record.
value	Specifies a value in pixels and returns a value in the dimensional units used by *widget*.

Related Information

XmeToHorizontalPixels(3), **XmeFromHorizontalPixels**(3), and **XmeToVerticalPixels**(3).

XmeGetDefaultPixel

Purpose

Generates colors based on the screen background color

Synopsis

#include <Xm/XmP.h>

void XmeGetDefaultPixel(
    Widget *widget***,**
    int *type***,**
    int *offset***,**
    XrmValue ****value*);**

Description

XmeGetDefaultPixel generates a color associated with a logical component of a widget. For example, you can use **XmeGetDefaultPixel** to find the foreground color of *widget*.

XmeGetDefaultPixel uses the screen’s background color to calculate the foreground, top shadow, bottom shadow, or select color.

widget Specifies the widget.

type Specifies the component of *widget* for which you are requesting information. Specify one of the following values:

XmBACKGROUND
Specifies a request for the default background color of the screen.

XmFOREGOUND
Specifies a request for the generated foreground color.

XmTOP_SHADOW
Specifies a request for the generated top shadow color. The top shadow is the portion of the shadow that is generally above and to the left of the widget.

XmBOTTOM_SHADOW
Specifies a request for the generated bottom shadow color. The bottom shadow is the portion of the shadow that is generally below and to the right of the widget.

XmSELECT
Specifies a request for the generated select color.

If you do not specify one of the preceding values, **XmeGetDefaultPixel** returns a warning message.

offset Specifies the byte offset of this field from the beginning of the widget class. (**XmeGetDefaultPixel** currently ignores this field; this field is provided solely for compatibility with the Intrinsics.)

value Returns the requested color.

Related Information

None.

XmeGetDefaultRenderTable

Purpose

Returns the default render table associated with a specified widget

Synopsis

#include <Xm/XmP.h>

XmRenderTable XmeGetDefaultRenderTable(
    Widget *widget***,**
    unsigned char *renderTableType***);**

Description

XmeGetDefaultRenderTable returns the default render table associated with widget *widget*.

To determine the default render table, **XmeGetDefaultRenderTable** scans the parent hierarchy of *widget*, searching for the first ancestor to hold the *XmQTspecifyRenderTable* trait. If it finds one, **XmeGetDefaultRenderTable** calls the **getRenderTable** trait method of the *XmQTspecifyRenderTable* trait. This trait method returns the appropriate default render table based on *renderTableType*.

In the standard Motif widget set, the following widgets hold the *XmQTspecifyRenderTable* trait:

- **XmBulletinBoard**
- **XmMenuShell**
- **VendorShell**

If **XmeGetDefaultRenderTable** cannot find an ancestor holding the *XmQTspecifyRenderTable* trait, **XmeGetDefaultRenderTable** creates a render table from the default system font.

widget The name of the widget whose render table you are seeking.

renderTableType
You must specify one of the following:

XmLABEL_RENDER_TABLE
Specifies a request for the default render table associated with labels. Users or applications typically specify this default through the **XmNlabelRenderTable** resource of **XmBulletinBoard**, **VendorShell**, or **XmMenuShell**.

XmTEXT_RENDER_TABLE
Specifies a request for the default render table associated with text. Users or applications typically specify this default through the **XmNtextRenderTable** resource of **XmBulletinBoard**, **VendorShell**, or **XmMenuShell**.

XmBUTTON_RENDER_TABLE
Specifies a request for the default render table associated with buttons. Users or applications typically specify this default through the **XmNbuttonRenderTable** resource of **XmBulletinBoard**, **VendorShell**, or **XmMenuShell**.

Return Values

Returns the default render table associated with widget *widget*. Your widget should always make a copy of the returned render table and use the copy rather than the original.

Related Information

XmeRenderTableGetDefaultFont(3).

XmeGetDirection

Purpose

A compound string parse procedure to insert a direction component

Synopsis

#include <Xm/XmP.h>

XmIncludeStatus XmeGetDirection(
> **XtPointer** ***text_in_out*,**
> **XtPointer** *text_end***,**
> **XmTextType** *type***,**
> **XmStringTag** *tag***,**
> **XmParseMapping** *entry***,**
> **int** *pattern_length***,**
> **XmString** ***str_include*,**
> **XtPointer** *call_data***);**

Description

XmeGetDirection is a compound string parse procedure (a function of type **XmParseProc**). It is suitable for use as an **XmNinvokeParseProc** procedure in an **XmParseMapping** data type.

XmeGetDirection is triggered when the implicit direction of the current character in text being parsed is different from the direction of the previous character. The function assumes that *text_in_out* points to the character whose direction has changed. It creates a compound string with a component of type **XmSTRING_COMPONENT_DIRECTION** that specifies the direction of the character at the *text_in_out* position. The argument *text_in_out* remains unchanged.

text_in_out Specifies the text being parsed. The value is a pointer to the first byte of a character whose implicit direction is different from that of the previous character. When the parse procedure returns, this argument is not set to a value different from the input value.

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. For locale text, *type* has a value of either **XmMULTIBYTE_TEXT** or **XmWIDECHAR_TEXT**. For charset text, *type* has a value of **XmCHARSET_TEXT**.
tag	Specifies the tag to be used in determining the character direction.
entry	Specifies the parse mapping that triggered the call to the parse procedure.
pattern_length	Specifies the number of bytes in the input text 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.

Return Values

This function returns **XmINSERT**, indicating that the parsing routine should concatenate the result to the compound string being constructed and continue parsing.

Related Information

XmParseMapping(3), **XmParseTable**(3), **XmString**(3), and **XmeGetNextCharacter**(3).

XmeGetEncodingAtom

Purpose

A toolkit function that returns the encoding of the locale

Synopsis

#include <Xm/TransferP.h>

Atom XmeGetEncodingAtom(
Widget *widget***);**

Description

XmeGetEncodingAtom returns an **Atom** that represents the encoding of the current locale. This routine calls **XmbTextListToTextProperty** with the display of the specified widget, a string of characters in the X Portable Character Set, and an encoding style of *XTextProperty*. The routine returns the **Atom** returned by **XmbTextListToTextProperty** as the *encoding* member of the *XTextProperty* structure.

widget	Specifies the widget whose display is to be used.

Return Values

If **XmbTextListToTextProperty** returns **Success**, this function returns an **Atom** representing the encoding of the locale. Otherwise, the function returns **None**.

Related Information

XmQTtransfer(3), **XmTransferDone**(3), **XmTransferValue**(3), **XmeClipboardSink**(3), **XmeClipboardSource**(3), **XmeConvertMerge**(3), **XmeDragSource**(3), **XmeDropSink**(3), **XmePrimarySink**(3), **XmePrimarySource**(3), **XmeSecondarySink**(3), **XmeSecondarySource**(3), **XmeSecondaryTransfer**(3), **XmeStandardConvert**(3), **XmeStandardTargets**(3), and **XmeTransferAddDoneProc**(3).

XmeGetHomeDirName

Purpose

Returns the pathname of the user’s home directory

Synopsis

#include <Xm/XmosP.h>

```
String XmeGetHomeDirName(
        void);
```

Description

XmeGetHomeDirName returns the pathname of the user’s home directory. The definition of the user’s home directory varies depending on the operating system.

Do not deallocate (for example, with **XtFree**) the returned **String**.

Do not modify the returned **String** holding the pathname of the user’s home directory.

Return Values

Returns a **String** holding the pathname of the user’s home directory. If **XmeGetHomeDirName** is unable to determine the home directory, the returned **String** will be zero length.

Related Information

None.

XmeGetLocalizedString

Purpose

Returns a localized version of the input string

Synopsis

#include <Xm/XmosP.h>

XmString XmeGetLocalizedString(
  char ****reserved*,**
  Widget *widget***,**
  char ****resource*,**
  String *string***);**

Description

XmeGetLocalizedString returns a localized version of *string*. On most implementations of Motif, **XmeGetLocalizedString** ignores the values of *reserved*, *widget*, and *resource*. For these implementations, a call to **XmeGetLocalizedString** is equivalent to a call to **XmStringCreateLocalized**.

However, Motif vendors may use the values of *widget* and/or *resource* in an implementation-specific way. For example, if *string* is *NULL*, some Motif implementations return the localized version of the name of *widget*.

reserved	Reserved for future use.
widget	Specifies a widget ID.
resource	Specifies a resource name.
string	Specifies a string in ISO 8859-1 string format.

Return Values

Returns a localized version of *string* in **XmString** format.

Related Information

XmStringCreateLocalized(3).

XmeGetNextCharacter

Purpose

A compound string parse procedure to insert a character

Synopsis

#include <Xm/XmP.h>

XmIncludeStatus XmeGetNextCharacter(
  XtPointer ****text_in_out*,**
  XtPointer *text_end***,**
  XmTextType *type***,**
  XmStringTag *tag***,**
  XmParseMapping *entry***,**
  int *pattern_length***,**
  XmString ****str_include*,**
  XtPointer *call_data***);**

Description

XmeGetNextCharacter is a compound string parse procedure (a function of type **XmParseProc**). It is suitable for use as an **XmNinvokeParseProc** procedure in an **XmParseMapping** data structure.

XmeGetNextCharacter causes the character following the pattern characters to be used as output to the compound string under construction. The function assumes that *text_in_out* points to the first byte of the first character in the matched pattern. It creates a compound string consisting of a component that contains the next character following the matching characters. It sets *text_in_out* to a pointer to the first byte following the character parsed.

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 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 type of compound string component the function creates. If *type* is **XmCHARSET_TEXT** and *tag* is other than **XmFONTLIST_DEFAULT_TAG**, the function creates a component of type **XmSTRING_COMPONENT_TEXT**. If *type* is **XmCHARSET_TEXT** and *tag* is **XmFONTLIST_DEFAULT_TAG**, or if *type* is **XmMULTIBYTE_TEXT**, the function creates a component of type **XmSTRING_COMPONENT_LOCALE_TEXT**. If *type* is **XmWIDECHAR_TEXT**, the function creates a component of type **XmSTRING_COMPONENT_WIDECHAR_TEXT**.

tag Specifies the tag to be used along with a *type* of **XmCHARSET_TEXT** in determining the type of compound string component the function creates.

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.

Return Values

This function returns **XmINSERT**, indicating that the parsing routine should concatenate the result to the compound string being constructed and continue parsing.

Related Information

XmParseMapping(3), **XmParseTable**(3), **XmString**(3), and **XmeGetDirection**(3).

XmeGetNullCursor

Purpose

Returns the null cursor associated with a given display

Synopsis

#include <Xm/ScreenP.h>

Cursor XmeGetNullCursor(
Widget *widget***);**

Description

Use **XmeGetNullCursor** to get a handle to a null cursor. The "null" cursor is transparent. A null cursor maintains cursor state information like the cursor's current position and hotspot. However, a null cursor is invisible to the user.

For example, you might want to use a null cursor instead of a visible cursor during certain drag operations.

widget	Specifies any instantiated widget on the target screen.

Return Values

Returns a handle to a null cursor.

Related Information

XmeQueryBestCursorSize(3).

XmeGetPixmapData

Purpose

Returns details about a cached pixmap

Synopsis

#include <Xm/XmP.h>

Boolean XmeGetPixmapData(
  Screen ****screen*,**
  Pixmap *pixmap***,**
  char *****image_name*,**
  int ****depth*,**
  Pixel ****foreground*,**
  Pixel ****background*,**
  int ****hot_x*,**
  int ****hot_y*,**
  unsigned int ****width*,**
  unsigned int ****height*);**

Description

Motif maintains a pixmap cache. This cache holds information on the pixmaps currently in use. Use **XmeGetPixmapData** to determine if a specified *pixmap* is currently in the pixmap cache. If it is, **XmeGetPixmapData** returns the value True and returns information for all eight of its output arguments.

If *pixmap* is not in the pixmap cache, **XmeGetPixmapData** fetches *pixmap* from the server, caches *pixmap*, and then returns False. When **XmeGetPixmapData** returns False, only *depth*, *width*, and *height* are valid.

XmeGetPixmapData is relatively efficient because it gets the data through a table lookup rather than through a server request.

screen	Specifies a pointer to the screen.
pixmap	Specifies the desired pixmap.
image_name	Returns the image name associated with *pixmap*.

	The caller must not modify or free the returned data.
depth	Returns the depth of the *pixmap*.
foreground	Returns the foreground color associated with *pixmap*.
background	Returns the background color associated with *pixmap*.
hot_x	Returns the x-coordinate of the hotspot of *pixmap* if hotspot information was stored with the *pixmap*.
hot_y	Returns the y-coordinate of the hotspot of *pixmap* if hotspot information was stored with the *pixmap*.
width	Returns the width of *pixmap*.
height	Returns the height of *pixmap*.

Return Values

Returns True if *pixmap* is in the image cache. Otherwise, it returns False.

Related Information

XmGetPixmap(3).

XmeGetTextualDragIcon

Purpose

Returns an icon widget symbolizing a textual drag operation in progress

Synopsis

#include <Xm/DragIconP.h>

Widget XmeGetTextualDragIcon(
Widget *widget***);**

Description

The *CDE 2.1/Motif 2.1—Style Guide and Glossary* requires an application to display a drag icon to symbolize a drag operation in progress. Each drag icon consists of a source icon, a state icon, and an operation icon. The source icon symbolizes the object being dragged. If the object being dragged is text, the standard Motif text source icon should be displayed. To get the standard Motif text source icon, call **XmeGetTextualDragIcon**. For more information about drag icons, see the *Motif 2.1—Programmer's Guide*.

The **XmeGetTextualDragIcon** function returns the standard text source icon widget. Typically, you assign this returned widget as the value of the **XmNsourceCursorIcon** resource of the **XmDragContext** widget; for example:

```
drag_icon = XmeGetTextualDragIcon(w);
XtSetArg(args[n], XmNsourceCursorIcon, drag_icon), n++;
(void) XmeDragSource(w, event, args, n);
```

widget	Specifies any widget displayed on the screen on which you want the source icon to appear.

Return Values

Returns a text source icon widget. The returned icon depends on the value of the **XmNenableDragIcon** resource of **XmScreen**.

Related Information

XmDragStart(3), **XmDragContext**(3), and **XmScreen**(3).

XmeMicroSleep

Purpose

Suspends execution for a specified number of microseconds

Synopsis

#include <Xm/XmosP.h>

int XmeMicroSleep(
  long *microsecs***);**

Description

XmeMicroSleep suspends execution of a client for a specified number of microseconds.

microsecs Specifies the number of microseconds.

Return Values

XmeMicroSleep returns an integer value symbolizing the success or failure of the function. A return value of 0 (zero) indicates successful completion of the function. A return value of -1 indicates that something went wrong.

Related Information

None.

XmeNamedSink

Purpose

A toolkit function that transfers data from the named selection to a widget

Synopsis

#include <Xm/TransferP.h>

Boolean XmeNamedSink(
  Widget *widget***,**
  Atom *named_selection***,**
  XtEnum *op***,**
  XtPointer *location_data***,**
  Time *time***);**

Description

XmeNamedSink transfers data from the specified selection to a widget.

This routine initializes an **XmDestinationCallbackStruct** with the *selection* member set to *named_selection*, the *operation* member set to the value of the *op* argument, the *location_data* member set to the value of the *location_data* argument, and the *time* member set to the value of the *time* argument. The *location_data* value contains information about the location where data is to be transferred. If the value is *NULL*, the data is to be inserted at the widget's cursor position. Otherwise, the type and interpretation of the value are specific to the widget class.

This routine calls the widget's **destinationPreHookProc** *XmQTtransfer* trait method with this **XmDestinationCallbackStruct**. It then calls the widget's **XmNdestinationCallback** procedures, if any. Unless an **XmNdestinationCallback** procedure has called *XmTransferDone* with a status other than **XmTRANSFER_DONE_DEFAULT**, this routine calls the widget's **destinationProc** *XmQTtransfer* trait method after all transfers initiated by callback procedures have finished. It is the responsibility of the **XmNdestinationCallback** procedures and the **destinationProc** method to transfer any data to the widget.

If *op* is **XmMOVE** and the data is successfully transferred, **XmeNamedSink** asks the owner of the selection named by *named_selection* to convert that selection to the *DELETE* target. If the selection owner has called **XmeNamedSource** to take ownership of the selection, this conversion request first calls the owner's **XmNconvertCallback** procedures, if any. If no **XmNconvertCallback** procedures exist or if these procedures return a status of **XmCONVERT_DEFAULT** or **XmCONVERT_MERGE**, this request then calls the owner's **convertProc** *XmQTtransfer* trait method.

widget Specifies the widget that is the destination for the data.

named_selection
Specifies the desired selection from which to obtain the data.

op Specifies the transfer operation. Possible values are **XmCOPY**, **XmMOVE**, and **XmLINK**.

location_data
Specifies information about the location where data is to be transferred. If the value is *NULL*, the data is to be inserted at the widget's cursor position. Otherwise, the type and interpretation of the value are specific to the widget class. If *location_data* cannot fit inside an *XtPointer*, *location_data* must either be a static variable or be allocated. If *location_data* is allocated, a call must be made to **XmeTransferAddDoneProc** to establish a procedure to free the allocated memory. The value of *location_data* is only valid for the duration of a transfer. Once transfer done procedures start to be called, *location_data* will no longer be stable.

time Specifies the time of the transfer. This is usually the timestamp from the event passed to an action routine. You should call *XtLastTimeStampProcessed* to generate the *time* value. If you set *time* to **CurrentTime** or 0, UTM will automatically change the call to *XtLastTimeStampProcessed*.

Return Values

This function returns False if no transfers take place. Otherwise, it returns True.

Related Information

XmQTtransfer(3), **XmTransferDone**(3), **XmTransferValue**(3), **XmeClipboardSink**(3), **XmeClipboardSource**(3), **XmeConvertMerge**(3),

XmeDragSource(3), **XmeDropSink**(3), **XmeGetEncodingAtom**(3), **XmePrimarySource**(3), **XmeNamedSource**(3), **XmeSecondarySink**(3), **XmeSecondarySource**(3), **XmeSecondaryTransfer**(3), **XmeStandardConvert**(3), **XmeStandardTargets**(3), and **XmeTransferAddDoneProc**(3).

XmeNamedSource

Purpose

A toolkit function that takes ownership of a named selection

Synopsis

#include <Xm/TransferP.h>

```
Boolean XmeNamedSource(
        Widget widget,
        Atom named_selection,
        Time time);
```

Description

XmeNamedSource takes ownership of the specified selection.

This routine establishes a function that is called when the widget is asked to convert the selection named *named_selection*. That function makes the following sequence of calls:

1. Calls the application's **XmNconvertCallback** procedures, if any. **XmeNamedSource** passes an **XmConvertCallbackStruct** (with the *selection* member set to *named_selection*) to each of these **XmNconvertCallback** procedures.
2. Calls the source widget's **convertProc** trait method; **convertProc** is one of the trait methods of the *XmQTtransfer* trait. However, if any **XmNconvertCallback** procedure returns **XmCONVERT_DONE** or **XmCONVERT_REFUSE**, then **XmeNamedSource** will not call **convertProc**.

This routine also establishes a function that is called when the widget loses the selection. That function makes the following sequence of calls:

1. Calls the application's **XmNconvertCallback** procedures, if any. **XmeNamedSource** passes an **XmConvertCallbackStruct** to each of these **XmNconvertCallback** procedures. **XmeNamedSource** initializes the **XmConvertCallbackStruct** as follows:
   - Sets the *selection* member to the *named_selection* argument

- Sets the *target* member to _MOTIF_LOSE_SELECTION

2. Calls the source widget's **convertProc** trait method. However, if any **XmNconvertCallback** procedure returns **XmCONVERT_DONE** or **XmCONVERT_REFUSE**, then **XmeNamedSource** will not call **convertProc**.

widget Specifies the widget that is to take ownership of the selection.

named_selection
Specifies the **Atom** that names the selection to own.

time Specifies the time of the transfer. This is usually the timestamp from the event passed to an action routine. You should call *XtLastTimeStampProcessed* to generate the *time* value. If you set *time* to **CurrentTime** or 0, UTM will automatically change the call to *XtLastTimeStampProcessed*.

Return Values

This function returns False if the widget cannot take ownership of the specified selection. Otherwise, it returns True.

Related Information

XmQTtransfer(3), **XmTransferDone**(3), **XmTransferValue**(3), **XmeClipboardSink**(3), **XmeClipboardSource**(3), **XmeConvertMerge**(3), **XmeDragSource**(3), **XmeDropSink**(3), **XmeGetEncodingAtom**(3), **XmePrimarySink**(3), **XmeNamedSink**(3), **XmeSecondarySink**(3), **XmeSecondarySource**(3), **XmeSecondaryTransfer**(3), **XmeStandardConvert**(3), **XmeStandardTargets**(3), and **XmeTransferAddDoneProc**(3).

XmeNamesAreEqual

Purpose

Compares two strings for equality

Synopsis

#include <Xm/XmP.h>

Boolean XmeNamesAreEqual(
    char ****in_str*,**
    char ****test_str*);**

Description

XmeNamesAreEqual compares *in_str* to *test_str* and returns True if they are equivalent. Typically, *in_str* holds some Motif constant, such as "Xm1000TH_INCHES" and *test_str* holds a normalized version of a constant, such as "1000th_inches".

The case of letters in *in_str* is irrelevant. For comparison purposes, **XmeNamesAreEqual** automatically converts all letters in *in_str* to lowercase. By contrast, all letters in *test_str* must be lowercase. If any letter in *test_str* is uppercase, then **XmeNamesAreEqual** automatically returns False.

For example, if *in_str* contains "Goodbye" and *test_str* contains "goodbye", **XmeNamesAreEqual** returns True (because the function converts "Goodbye" to "goodbye"). However, if *in_str* contains "Goodbye" and *test_str* contains "Goodbye", **XmeNamesAreEqual** returns False (because of the uppercase "G" in *test_str*).

XmeNamesAreEqual treats the following prefixes specially: *XM*, **Xm**, **xM**, and **xm**. If one of these prefixes appears in *in_str*, **XmeNamesAreEqual** skips over it and begins the comparison at the third character in *in_str*. For example, if *in_str* contains "XMhello" and *test_str* contains "hello", then **XmeNamesAreEqual** returns True. However, **XmeNamesAreEqual** provides no special treatment if one of these prefixes appears in *test_str*. For example, if *in_str* contains "XMhello" and *test_str* contains "xmhello", then **XmeNamesAreEqual** returns False.

When the strings to be compared are enumerated values, then you should probably consider using the representation type facility of Motif instead of

XmeNamesAreEqual. The representation type facility of Motif obeys the same string comparison rules as **XmeNamesAreEqual**.

XmeNamesAreEqual is used primarily by programmers writing their own resource converters.

in_str	Specifies one of the two strings to compare. A prefix of *XM*, **Xm**, **xM**, or **xm** is ignored, as is the case of letters appearing in the string.
test_str	Specifies the other string to compare. This string should not start with the prefix *XM*, **Xm**, **xM**, or **xm**. Also, all letters in the string must be lowercase letters.

Return Values

Returns True if the strings match. Otherwise, it returns False.

Related Information

XmRepTypeRegister(3).

XmeNavigChangeManaged

Purpose

Helps a change_managed method establish the correct keyboard traversal policy

Synopsis

#include <Xm/XmP.h>

void XmeNavigChangeManaged(
        Widget *widget***);**

Description

If you are writing a manager widget, your **change_managed** method must call **XmeNavigChangeManaged**. The **XmeNavigChangeManaged** function helps a manager widget cope with changes to its child widgets. The **XmeNavigChangeManaged** function establishes the correct keyboard traversal policy for all the child widgets of the manager.

widget	Specifies a manager widget.

Related Information

XmeRedisplayGadgets(3) and **XmGadget**(3).

XmePrimarySink

Purpose A toolkit function that transfers data from the primary selection to a widget

Synopsis **#include <Xm/TransferP.h>**

Boolean XmePrimarySink(
    Widget *widget***,**
    XtEnum *op***,**
    XtPointer *location_data***,**
    Time *time***);**

Description

XmePrimarySink transfers data from the primary selection to a widget.

XmePrimarySink initializes an **XmDestinationCallbackStruct** as follows:

- Sets the *selection* member to *PRIMARY*
- Sets the *operation* member to the value of the *op* argument
- Sets the *location_data* member to the value of the *location_data* argument
- Sets the *time* member to the value of the *time* argument

XmePrimarySink makes the following sequence of calls:

1. Calls the destination widget's **destinationPreHookProc** trait method, if any. **destinationPreHookProc** is one of the trait methods of the *XmQTtransfer* trait. **XmePrimarySink** passes the initialized **XmDestinationCallbackStruct** as the *call_data* argument.
2. Calls any **XmNdestinationCallback** procedures that the application has attached to the destination widget.
3. Calls the destination widget's **destinationProc** trait method, if any, after all transfers initiated by **XmNdestinationCallback** procedures have finished. However, if an **XmNdestinationCallback** procedure has called *XmTransferDone*

with a status of **XmTRANSFER_DONE_DEFAULT**, **XmePrimarySink** does not call the **destinationProc** trait method.

It is the responsibility of the **XmNdestinationCallback** procedures and the **destinationProc** trait method to transfer any data to the destination widget.

If *op* is **XmMOVE** and the data is successfully transferred, **XmePrimarySink** asks the owner of the *PRIMARY* selection to convert that selection to the *DELETE* target. If the selection owner has called **XmePrimarySource** to take ownership of the selection, this conversion request first calls the owner's **XmNconvertCallback** procedures, if any. If no **XmNconvertCallback** procedures exist or if these procedures return a status of **XmCONVERT_DEFAULT** or **XmCONVERT_MERGE**, this request then calls the owner widget's **convertProc** trait method.

widget Specifies the widget that is the destination for the data.

op Specifies the transfer operation. Possible values are **XmCOPY**, **XmMOVE**, and **XmLINK**.

location_data
Specifies information about the location where data is to be transferred. If the value is *NULL*, the data is to be inserted at the widget's cursor position. Otherwise, the type and interpretation of the value are specific to the widget class. If *location_data* cannot fit inside an *XtPointer*, *location_data* must either be a static variable or be allocated. If *location_data* is allocated, a call must be made to **XmeTransferAddDoneProc** to establish a procedure to free the allocated memory. The value of *location_data* is only valid for the duration of a transfer. Once the transfer done procedures start to be called, *location_data* will no longer be stable.

time Specifies the time of the transfer. This is usually the timestamp from the event passed to an action routine. You should call *XtLastTimeStampProcessed* to generate the *time* value. If you set *time* to **CurrentTime** or 0, UTM will automatically change the call to *XtLastTimeStampProcessed*.

Return Values

This function returns False if no transfers take place. Otherwise, it returns True.

Related Information

XmQTtransfer(3), **XmTransferDone**(3), **XmTransferValue**(3), **XmeClipboardSink**(3), **XmeClipboardSource**(3), **XmeConvertMerge**(3), **XmeDragSource**(3), **XmeDropSink**(3), **XmeGetEncodingAtom**(3), **XmePrimarySource**(3), **XmeSecondarySink**(3), **XmeSecondarySource**(3), **XmeSecondaryTransfer**(3), **XmeStandardConvert**(3), **XmeStandardTargets**(3), and **XmeTransferAddDoneProc**(3).

XmePrimarySource

Purpose

A toolkit function that takes ownership of the primary selection

Synopsis

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

Boolean XmePrimarySource(
        Widget widget,
        Time time);
```

Description

XmePrimarySource takes ownership of the primary selection.

This routine establishes a function that is called when the widget is asked to convert the primary selection. That function makes the following sequence of calls:

1. Calls the application's **XmNconvertCallback** procedures, if any. **XmePrimarySource** passes an **XmConvertCallbackStruct** (with the *selection* member set to *PRIMARY*) to each of these **XmNconvertCallback** procedures.
2. Calls the source widget's **convertProc** trait method. (**convertProc** is one of the trait methods of the *XmQTtransfer* trait.) However, if any **XmNconvertCallback** procedure returns **XmCONVERT_DONE** or **XmCONVERT_REFUSE**, then **XmePrimarySource** will not call **convertProc**.

XmePrimarySource also establishes a function that is called when the widget loses the selection. That function makes the following sequence of calls:

1. Calls the application's **XmNconvertCallback** procedures, if any. **XmePrimarySource** passes an **XmConvertCallbackStruct** to each of these **XmNconvertCallback** procedures. **XmePrimarySource** initializes the **XmConvertCallbackStruct** as follows:
    - Sets the *selection* member to *PRIMARY*

- Sets the *target* member to _MOTIF_LOSE_SELECTION

2. Calls the source widget’s **convertProc** trait method. However, if any **XmNconvertCallback** procedure returns **XmCONVERT_DONE** or **XmCONVERT_REFUSE**, then **XmePrimarySource** will not call **convertProc**.

widget Specifies the widget that is to take ownership of the selection.

time Specifies the time of the transfer. This is usually the timestamp from the event passed to an action routine. You should call *XtLastTimeStampProcessed* to generate the *time* value. If you set *time* to **CurrentTime** or 0, UTM will automatically change the call to *XtLastTimeStampProcessed*.

Return Values

This function returns False if the widget cannot take ownership of the primary selection. Otherwise, it returns True.

Related Information

XmQTtransfer(3), **XmTransferDone**(3), **XmTransferValue**(3), **XmeClipboardSink**(3), **XmeClipboardSource**(3), **XmeConvertMerge**(3), **XmeDragSource**(3), **XmeDropSink**(3), **XmeGetEncodingAtom**(3), **XmePrimarySink**(3), **XmeSecondarySink**(3), **XmeSecondarySource**(3), **XmeSecondaryTransfer**(3), **XmeStandardConvert**(3), **XmeStandardTargets**(3), and **XmeTransferAddDoneProc**(3).

XmeQueryBestCursorSize

Purpose

Finds the best cursor size

Synopsis

#include <Xm/ScreenP.h>

void XmeQueryBestCursorSize(
    Widget *widget***,**
    Dimension ****width*,**
    Dimension ****height*);**

Description

XmeQueryBestCursorSize finds the best cursor size for a particular screen. **XmeQueryBestCursorSize** is similar to the Xlib call **XQueryBestSize**. The only difference is that **XmeQueryBestCursorSize** does not make a server request each time it is called. Therefore, **XmeQueryBestCursorSize** is more efficient than **XQueryBestSize**.

widget	Specifies any instantiated widget on the screen for which you desire cursor information.
width	Specifies the suggested cursor width for the display on which *widget* is instantiated.
height	Specifies the suggested cursor height for the display on which *widget* is instantiated.

Related Information

XQueryBestCursor(3) and **XQueryBestSize**(3).

XmeRedisplayGadgets

Purpose

Redisplays all the gadget children of a manager

Synopsis

#include <Xm/XmP.h>

void XmeRedisplayGadgets(
　　Widget *widget***,**
　　XEvent ***event***,**
　　Region *region***);**

Description

XmeRedisplayGadgets redisplays the gadget children of a manager widget. If you set *region* to *NULL*, **XmeRedisplayGadgets** redisplays all the gadget children of the manager widget. If you set *region* to something other than *NULL*, **XmeRedisplayGadgets** redisplays all the gadget children of the manager widget that fall partially or completely within *region*.

widget	Specifies a manager widget.
event	Specifies a pointer to an event (typically, an expose event). This argument will be passed through to the Intrinsics.
region	Specifies the region to be redisplayed. All gadgets falling within this region will be redisplayed. If you specify a value of *NULL* for *region*, then all gadgets belonging to *widget* will be redisplayed.

Related Information

XmeNavigChangeManaged(3), and **XmGadget**(3).

XmeRemoveFocusChangeCallback

Purpose

Removes a focus change callback

Synopsis

#include <Xm/VendorSEP.h>

void XmeRemoveFocusChangeCallback(
  Widget *widget***,**
  XtCallbackProc *callback_procedure***,**
  XtPointer *data***);**

Description

XmeRemoveFocusChangeCallback removes a *callback_procedure* that was installed with **XmeAddFocusChangeCallback**.

widget Specifies the widget whose focus change callback is to be removed. The specified *widget* must be a **VendorShell** or a subclass of **VendorShell**.

callback_procedure
Specifies the callback procedure to be removed from the list of focus change callback procedures.

data Specifies the *data* argument that was passed to **XmeAddFocusChangeCallback**. In order for a focus change callback to be removed, the *data* arguments in **XmeRemoveFocusChangeCallback** and **XmeAddFocusChangeCallback** must match.

Related Information

XmeAddFocusChangeCallback(3).

XmeRenderTableGetDefaultFont

Purpose

Gets information on the default font associated with a specified render table

Synopsis

#include <Xm/XmP.h>

Boolean XmeRenderTableGetDefaultFont(
XmRenderTable *renderTable***,**
XFontStruct *****fontStruct*);**

Description

XmeRenderTableGetDefaultFont returns information about the default font associated with a render table.

XmeRenderTableGetDefaultFont searches **renderTable** for the first rendition tagged with **XmFONTLIST_DEFAULT_TAG**. If such a rendition is found, the default font is the font (or first font in the font set) associated with this tag. If such a rendition is not found, the default font is taken from the first rendition in **renderTable**.

renderTable	Specifies the render table.
fontStruct	Returns information about the default font.

Return Values

Returns True if **renderTable** is not *NULL*. Otherwise, it returns False.

Related Information

XmeGetDefaultRenderTable(3).

XmeReplyToQueryGeometry

Purpose

Handles standard geometry requests

Synopsis

#include <Xm/XmP.h>

XtGeometryResult XmeReplyToQueryGeometry(
 Widget *widget***,**
 XtWidgetGeometry ****intended*,**
 XtWidgetGeometry ****desired*);**

Description

A **query_geometry** method can call **XmeReplyToQueryGeometry** in order to handle certain kinds of geometry requests. **XmeReplyToQueryGeometry** has an important limitation; it returns an *XtGeometryResult* value based solely on width and height values. That is, **XmeReplyToQueryGeometry** ignores other characteristics of widget geometry, such as starting position and border width. Therefore, if your **query_geometry** method needs to analyze any of these other characteristics, then you need to write your own code, rather than relying on **XmeReplyToQueryGeometry**.

Your widget must determine its preferred size prior to calling **XmeReplyToQueryGeometry**. To be more precise, you need only determine the widget's preferred width and height. The *desired* variable holds the widget's preferred dimensions.

widget	Specifies the current widget.
intended	Specifies the geometry proposed by the parent of *widget*.
desired	Specifies the geometry preferred by *widget*.

Return Values

Returns **XtGeometryYes** if the *intended* geometry matches the *desired* geometry, and if the parent of *widget* intends to use these values. Otherwise, returns **XtGeometryNo** if the *desired* width and height match the widget’s current width and height. Otherwise, returns **XtGeometryAlmost**.

Related Information

XtResizeRequest(3).

XmeResolvePartOffsets

Purpose

Allows writing of upward-compatible applications and widgets

Synopsis

#include <Xm/XmP.h>

void XmeResolvePartOffsets(
WidgetClass *widget_class***,**
XmOffsetPtr ****offset*,**
XmOffsetPtr ****constraint_offset*);**

Description

The use of offset records requires two extra global variables per widget class. The variables consist of pointers to arrays of offsets into the widget record and constraint record for each part of the widget structure. The **XmeResolvePartOffsets** function allocates the offset records needed by an application to guarantee upward-compatible access to widget instance and constraint records by applications and widgets. These offset records are used by the widget to access all of the widget's variables. A widget needs to take the steps described in the following paragraphs.

Instead of creating a resource list, the widget creates an offset resource list. To accomplish this, use the **XmPartResource** structure and the **XmPartOffset** macro. The **XmPartResource** data structure looks just like a resource list, but, instead of having one integer for its offset, it has two shorts. This structure is put into the class record as if it were a normal resource list. Instead of using **XtOffset** for the offset, the widget uses **XmPartOffset**.

If the widget is a subclass of the **Constraint** class and it defines additional constraint resources, create an offset resource list for the constraint part as well. Instead of using **XtOffset** for the offset, the widget uses **XmConstraintPartOffset** in the constraint resource list.

Do not create parts that do not preserve alignment (for example, a widget part containing a single Boolean).

```
XmPartResource resources[] = {
        {       BarNxyz, BarCXyz, XmRBoolean, sizeof(Boolean),
                XmPartOffset(Bar,xyz), XmRImmediate, (XtPointer)False } };
XmPartResource constraints[] = {
        {       BarNmaxWidth, BarNMaxWidth,
                XmRDimension, sizeof(Dimension),
                XmConstraintPartOffset(Bar,max_width),
                XmRImmediate, (XtPointer)100 } };
```

Instead of putting the widget size in the class record, the widget puts the widget part size in the same field. If the widget is a subclass of the **Constraint** class, instead of putting the widget constraint record size in the class record, the widget puts the widget constraint part size in the same field.

Instead of putting **XtVersion** in the class record, the widget puts **XtVersionDontCheck** in the class record.

Define a variable of type **XmOffsetPtr** to point to the offset record. If the widget is a subclass of the **Constraint** class, define a variable of type **XmOffsetPtr** to point to the constraint offset record. These can be part of the widget’s class record or separate global variables.

In class initialization, the widget calls **XmeResolvePartOffsets**, passing it pointers to the class record, the address of the offset record, and the address of the constraint offset record. If the widget is not a subclass of the **Constraint** class, it should pass *NULL* as the address of the **Constraint** offset record. This does several things:

- Adds the superclass (which, by definition, has already been initialized) size field to the part size field
- If the widget is a subclass of the **Constraint** class, adds the superclass constraint size field to the constraint size field
- Allocates an array based upon the number of superclasses
- If the widget is a subclass of the **Constraint** class, allocates an array for the constraint offset record
- Fills in the offsets of all the widget parts and constraint parts with the appropriate values, which can be determined by examining the size fields of all superclass records
- Uses the part offset array to modify the offset entries in the resource list to be real offsets, in place

The widget defines a constant that will be the index to its part structure in the offsets array. The value should be 1 greater than the index of the widget's superclass. Constants defined for all Xm widgets can be found in header file **Xm/XmP.h**.

```
#define BarIndex (XmBulletinBIndex + 1)
```

Instead of accessing fields directly, the widget must always go through the offset table. The **XmField** and **XmConstraintField** macros help you access these fields. Because the **XmPartOffset**, **XmConstraintPartOffset**, **XmField**, and **XmConstraintField** macros concatenate things, you must ensure that there is no space after the part argument. For example, the following macros do not work because of the space after the part (Label) argument:

```
XmField(w, offset, Label, text, char *)
XmPartOffset(Label, text).
```

You must not have any spaces after the part (Label) argument, illustrated as follows:

```
XmField(w, offset, Label, text, char *)
XmPartOffset(Label, text).
```

You can define macros for each field to make this easier. For example, the following shows how to define a macro for an enumerated field *xyz* in widget **Bar**:

```
#define BarXyz(w) \
        XmField(w,offsets,Bar,xyz,XtEnum)
```

Define a macro for constraint field *max_width* as follows:

```
#define BarMaxWidth(w) \
        XmConstraintField(w,constraint_offsets,Bar,max_width,Dimension)
```

The arguments for **XmeResolvePartOffsets** are as follows:

widget_class Specifies the widget class pointer for the created widget.

offset Returns the offset record.

constraint_offset
Returns the constraint offset record.

Related Information

XmResolveAllPartOffsets(3).

XmeSecondarySink

Purpose

A toolkit function that establishes a widget as the destination for secondary transfer

Synopsis

#include <Xm/TransferP.h>

Boolean XmeSecondarySink(
    Widget *widget***,**
    Time *time***);**

Description

XmeSecondarySink establishes a widget as the destination for secondary transfer operations. When the **VendorShell XmNkeyboardFocusPolicy** is **XmEXPLICIT**, the destination is the editable component that last had focus. When the **VendorShell XmNkeyboardFocusPolicy** is **XmPOINTER**, the destination is the editable component that last received mouse button or keyboard input.

XmeSecondarySink establishes a widget as the destination by taking ownership of the _MOTIF_DESTINATION selection.

XmeSecondarySink also establishes a function that is called when the widget loses the selection. That function makes the following sequence of calls:

1. Calls the widget's **XmNconvertCallback** procedures, if any. **XmeSecondarySink** passes an **XmConvertCallbackStruct** to each of these **XmNconvertCallback** procedures. **XmeSecondarySink** initializes the **XmConvertCallbackStruct** as follows:

- Sets the *selection* member to _MOTIF_DESTINATION
- Sets the *target* member to _MOTIF_LOSE_SELECTION

2. Calls the source widget's **convertProc** trait method. However, if any **XmNconvertCallback** procedure returns **XmCONVERT_DONE** or **XmCONVERT_REFUSE**, then **XmePrimarySource** will not call **convertProc**.

widget Specifies the widget that is to be the destination.

time Specifies the time of the transfer. This is usually the timestamp from the event passed to an action routine. You should call *XtLastTimeStampProcessed* to generate the *time* value. If you set *time* to **CurrentTime** or 0, UTM will automatically change the call to *XtLastTimeStampProcessed*.

Return Values

This function returns False if the widget cannot take ownership of the _MOTIF_DESTINATION selection. Otherwise, it returns True.

Related Information

XmQTtransfer(3), **XmTransferDone**(3), **XmTransferValue**(3), **XmeClipboardSink**(3), **XmeClipboardSource**(3), **XmeConvertMerge**(3), **XmeDragSource**(3), **XmeDropSink**(3), **XmeGetEncodingAtom**(3), **XmePrimarySink**(3), **XmePrimarySource**(3), **XmeSecondarySource**(3), **XmeSecondaryTransfer**(3), **XmeStandardConvert**(3), **XmeStandardTargets**(3), and **XmeTransferAddDoneProc**(3).

XmeSecondarySource

Purpose

A toolkit function that takes ownership of the secondary selection

Synopsis

#include <Xm/TransferP.h>

Boolean XmeSecondarySource(
    Widget *widget***,**
    Time *time***);**

Description

XmeSecondarySource takes ownership of the secondary selection. A widget usually calls **XmeSecondarySource** from an action routine that starts a secondary selection.

XmeSecondarySource establishes a function that is called when the widget is asked to convert the secondary selection. That function makes the following sequence of calls:

1. Calls the application's **XmNconvertCallback** procedures, if any. **XmeSecondarySource** passes an **XmConvertCallbackStruct** (with the *selection* member set to *SECONDARY*) to each of these **XmNconvertCallback** procedures.
2. Calls the source widget's **convertProc** trait method; **convertProc** is one of the trait methods of the *XmQTtransfer* trait. However, if any **XmNconvertCallback** procedure returns **XmCONVERT_DONE** or **XmCONVERT_REFUSE**, then **XmeSecondarySource** will not call **convertProc**.

XmeSecondarySource also establishes a function that is called when the widget loses the selection. That function makes the following sequence of calls:

1. Calls the application's **XmNconvertCallback** procedures, if any. **XmeSecondarySource** passes an **XmConvertCallbackStruct** to each of these **XmNconvertCallback** procedures. **XmeSecondarySource** initializes the **XmConvertCallbackStruct** as follows:
    - Sets the *selection* member to *SECONDARY*

- Sets the *target* member to _MOTIF_LOSE_SELECTION

2. Calls the source widget's **convertProc** trait method. However, if any **XmNconvertCallback** procedure returns **XmCONVERT_DONE** or **XmCONVERT_REFUSE**, then **XmeSecondarySource** will not call **convertProc**.

XmeSecondarySource is used in conjunction with **XmeSecondaryTransfer**.

widget	Specifies the widget that is to take ownership of the selection.
time	Specifies the time of the transfer. This is usually the timestamp from the event passed to an action routine. You should call *XtLastTimeStampProcessed* to generate the *time* value. If you set *time* to **CurrentTime** or 0, UTM will automatically change the call to *XtLastTimeStampProcessed*.

Return Values

This function returns False if the widget cannot take ownership of the secondary selection. Otherwise, it returns True.

Related Information

XmQTtransfer(3), **XmTransferDone**(3), **XmTransferValue**(3), **XmeClipboardSink**(3), **XmeClipboardSource**(3), **XmeConvertMerge**(3), **XmeDragSource**(3), **XmeDropSink**(3), **XmeGetEncodingAtom**(3), **XmePrimarySink**(3), **XmePrimarySource**(3), **XmeSecondarySink**(3), **XmeSecondaryTransfer**(3), **XmeStandardConvert**(3), **XmeStandardTargets**(3), and **XmeTransferAddDoneProc**(3).

XmeSecondaryTransfer

Purpose

A toolkit function that transfers data from the secondary selection to the destination widget

Synopsis

#include <Xm/TransferP.h>

void XmeSecondaryTransfer(
    Widget *widget***,**
    Atom *target***,**
    XtEnum *op***,**
    Time *time***);**

Description

XmeSecondaryTransfer transfers the data from the secondary selection to the destination widget. A widget usually calls this function from an action routine that ends a secondary selection. The destination widget is the owner of the _MOTIF_DESTINATION selection.

If the destination widget has used **XmeSecondarySink** to take ownership of that selection, **XmeSecondaryTransfer** initializes an **XmDestinationCallbackStruct** as follows:

- Sets the *selection* member to *SECONDARY*
- Sets the *operation* member to **XmLINK** if the *op* argument is **XmLINK**; sets the *operation* member to **XmCOPY** if the *op* argument is **XmCOPY** or **XmMOVE**.
- Sets the *destination_data* member to the value of the *target* argument

XmeSecondaryTransfer makes the following sequence of calls:

1. Calls the destination widget's **destinationPreHookProc** trait method, if any. **destinationPreHookProc** is one of the trait methods of the *XmQTtransfer* trait. **XmeSecondaryTransfer** passes the initialized **XmDestinationCallbackStruct** as the *call_data* argument to the trait method.

2. Calls the application's **XmNdestinationCallback** procedures, if any.
3. Calls the destination widget's **destinationProc** trait method, if any, after all transfers initiated by the **XmNdestinationCallback** procedures have finished. However, if an **XmNdestinationCallback** procedure has called *XmTransferDone* with a status of **XmTRANSFER_DONE_DEFAULT**, then **XmePrimarySink** will not call the **destinationProc** trait method.

It is the responsibility of the destination widget's **XmNdestinationCallback** procedures and the **destinationProc** trait method to transfer any data to the widget.

If *op* is **XmMOVE** and the data is successfully transferred, **XmeSecondaryTransfer** asks the owner of the *SECONDARY* selection to convert that selection to the *DELETE* target. In all cases, when the transfer operation is complete, **XmeSecondaryTransfer** asks the owner of the *SECONDARY* selection to convert that selection to the _MOTIF_LOSE_SELECTION target.

If the selection owner has called **XmeSecondarySource** to take ownership of the selection, these conversion requests first call the owner's **XmNconvertCallback** procedures, if any. If no **XmNconvertCallback** procedures exist or if these procedures return a status of **XmCONVERT_DEFAULT** or **XmCONVERT_MERGE**, these requests then call the owner's **convertProc** trait method; **convertProc** is a trait method of the *XmQTtransfer* trait.

widget	Specifies the widget that is the owner of the secondary selection.
target	Specifies a recommended target for the destination widget to request when converting the secondary selection.
op	Specifies the transfer operation. Possible values are **XmMOVE**, **XmCOPY**, and **XmLINK**.
time	Specifies the time of the transfer. This is usually the timestamp from the event passed to an action routine. You should call *XtLastTimeStampProcessed* to generate the *time* value. If you set *time* to **CurrentTime** or 0, UTM will automatically change the call to *XtLastTimeStampProcessed*.

Related Information

XmQTtransfer(3), **XmTransferDone**(3), **XmTransferValue**(3), **XmeClipboardSink**(3), **XmeClipboardSource**(3), **XmeConvertMerge**(3), **XmeDragSource**(3), **XmeDropSink**(3), **XmeGetEncodingAtom**(3),

XmePrimarySink(3), **XmePrimarySource**(3), **XmeSecondarySink**(3), **XmeSecondarySource**(3), **XmeStandardConvert**(3), **XmeStandardTargets**(3), and **XmeTransferAddDoneProc**(3).

XmeSetWMShellTitle

Purpose

A compound string function that updates the window manager title

Synopsis

#include <Xm/XmP.h>

void XmeSetWMShellTitle(
XmString *string***,**
Widget *shell***);**

Description

XmeSetWMShellTitle updates the title and icon name of a **WMShell** by using the specified compound string.

If *string* consists of a single segment whose tag is "ISO8859-1", the title and icon name use the text of the segment and an encoding of *STRING*. If *string* consists of a single segment whose tag is **XmFONTLIST_DEFAULT_TAG**, the title and icon name use the text of the segment and an encoding of **None**. Otherwise, the compound string is converted to compound text, and the title and icon name use the resulting compound text and an encoding of *COMPOUND_TEXT*.

This routine sets the **XmNtitle**, **XmNtitleEncoding**, **XmNiconName**, and **XmNiconNameEncoding** resources of a **WMShell**.

string	Specifies the compound string to use for the title and icon name.
shell	Specifies a widget that is a subclass of **WMShell**.

Related Information

WMShell(3) and **XmString**(3).

XmeStandardConvert

Purpose

A toolkit function that converts selections to standard targets

Synopsis

#include <Xm/TransferP.h>

void XmeStandardConvert(
  Widget *widget***,**
  XtPointer *ignore***,**
  XmConvertCallbackStruct ****call_data*);**

Description

XmeStandardConvert converts a selection to one of a set of standard targets. This function can be called only from an **XmNconvertCallback** procedure or from a **convertProc** trait method.

The target to which this routine tries to convert the selection is the *target* member of the **XmConvertCallbackStruct** passed in the *call_data* argument. If **XmeStandardConvert** converts the selection to that target, it fills in the *value*, *type*, *format*, and *length* members of the **XmConvertCallbackStruct**. Otherwise, it does not alter the callback struct.

Note that **XmeStandardConvert** does not alter the *status* member of the **XmConvertCallbackStruct**. The routine that calls **XmeStandardConvert** is responsible for looking at the *value* parameter and determining whether or not the conversion succeeded. If the *value* parameter contains *NULL*, then the conversion did not succeed.

Following are the targets to which **XmeStandardConvert** converts selections:

BACKGROUND

The routine sets the *value* member of the **XmConvertCallbackStruct** to the value of widget *widget*'s **XmNbackground** resource. The routine sets the *format* member to 32, the *length* member to 1, and the *type* member to *PIXEL*.

CLASS The routine finds the first shell in the widget *widget*'s hierarchy that has a WM_CLASS property. The properties of this shell determine the *value*, *format*, *length*, and *type* members of the **XmConvertCallbackStruct**.

CLIENT_WINDOW
The routine finds the first shell in the widget *widget*'s hierarchy. The routine sets the *value* member of the **XmConvertCallbackStruct** to the widget ID of this shell. This routine sets the *format* member to 32, the *length* member to 1, and the **Vype** member to *WINDOW*.

COLORMAP
The routine sets the *value* member of the **XmConvertCallbackStruct** to the value of widget *widget*'s **XmNcolormap** resource (or to the value of the **XmNcolormap** resource of widget *widget*'s parent, if *widget* is a gadget). This routine sets the *format* member to 32, the *length* member to 1, and the *type* member to *COLORMAP*.

FOREGROUND
The routine sets the *value* member of the **XmConvertCallbackStruct** to the value of widget *widget*'s **XmNforeground** resource. The routine sets the *format* member to 32, the *length* member to 1, and the *type* member to *PIXEL*.

NAME The routine finds the first shell in widget *widget*'s hierarchy that has a WM_NAME property. The properties of this shell determine the *Vvalue*. This routine sets the *format* member to 8, the *length* member to the number of characters in *value*, and the *type* member to the current locale.

TARGETS The routine sets the *value* member of the **XmConvertCallbackStruct** to the list of targets returned by **XmeStandardTargets**. This routine sets the *format* member to 32, the *length* member to the number of targets in the list, and the *type* member to *ATOM*.

_MOTIF_RENDER_TABLE
The routine sets the *value* member of the **XmConvertCallbackStruct** to the value of the widget *widget*'s **XmNrenderTable** resource if it exists, or else the default text render table. More precisely, the *value* member will hold a string of characters representing the render table. This routine sets the *format* member to 8, the *length* member to the number of characters in the *value* member, and the *type* member to *STRING*.

_MOTIF_ENCODING_REGISTRY
The routine transfers the widget's encoding registry. The routine sets the *value* member of the **XmConvertCallbackStruct** to a list of NULL-separated items in the form of tag encoding pairs. This target symbolizes the transfer target for the Motif Segment Encoding Registry. Widgets and applications can use this Registry to register text encoding formats for specified render table tags. Applications access this Registry by calling **XmRegisterSegmentEncoding** and **XmMapSegmentEncoding**. This routine sets the *format* member to 8, the *length* member to the number of characters in the *value* member, and the *type* member to *STRING*.

Following are the arguments to this function:

widget Specifies the reference widget for the conversion.

ignore This argument is ignored. Its value should always be *NULL*.

call_data Specifies a pointer to the **XmConvertCallbackStruct** to be modified. **XmeStandardConvert** modifies the following members of the structure only. (For details on **XmConvertCallbackStruct**, see **XmPrimitive**(3) in the *Motif 2.1—Programmer's Reference*.)

value An *XtPointer* parameter that contains any data that **XmeStandardConvert** produces as a result of the conversion. **XmeStandardConvert** sets *value* to *NULL* if it cannot convert the target.

type An **Atom** parameter that indicates the type of the data in the *value* member.

format An *int* parameter that specifies whether the data in *value* should be viewed as a list of 8-bit, 16-bit, or 32-bit quantities.

length An **unsigned long** parameter that specifies the number of elements of data in *value*, where each element has the number of bits specified by *format*.

Related Information

XmQTtransfer(3), **XmTransferDone**(3), **XmTransferValue**(3), **XmeClipboardSink**(3), **XmeClipboardSource**(3), **XmeConvertMerge**(3), **XmeDragSource**(3), **XmeDropSink**(3), **XmeGetEncodingAtom**(3),

XmePrimarySink(3), **XmePrimarySource**(3), **XmeSecondarySink**(3), **XmeSecondarySource**(3), **XmeSecondaryTransfer**(3), **XmeStandardTargets**(3), and **XmeTransferAddDoneProc**(3).

XmeStandardTargets

Purpose

A toolkit function that returns a list of standard targets

Synopsis

#include <Xm/TransferP.h>

Atom * XmeStandardTargets(
    Widget *widget***,**
    int *count***,**
    int ***count_return***);**

Description

Use **XmeStandardTargets** to return a list of the targets that your widget can convert. **XmeStandardTargets** is usually called from an **XmNconvertCallback** procedure or from the **convertProc** trait method of an *XmQTtransfer* trait, typically in response to a request to convert a selection to *TARGETS*.

XmeStandardTargets carves out enough dynamic memory to hold a list of all the **Atom**s your widget can convert. This list will consist of the ten standard targets plus any additional nonstandard targets supported by your widget. Use the *count* argument to specify the number of nonstandard targets supported by your widget. Then, **XmeStandardTargets** seeds the first ten entries in this list with the following ten standard targets:

- *BACKGROUND*
- *CLASS*
- *CLIENT_WINDOW*
- *COLORMAP*
- *FOREGROUND*
- *NAME*
- *TARGETS*

- *TIMESTAMP*
- _MOTIF_RENDER_TABLE
- _MOTIF_ENCODING_REGISTRY

Every Motif widget that acts as the source of a data transfer must be able to convert all the standard targets. (See **XmeStandardConvert**(3) for details.)

After **XmeStandardTargets** returns, the conversion routine is responsible for adding the *count* additional items to the end of the returned list, beginning with the *count_return* element.

widget	Specifies the reference widget for the conversion.
count	Specifies the number of additional targets, beyond the standard targets, that the widget supports.
count_return	Returns the number of standard targets at the front of the returned list of targets.

Return Values

This routine returns a list of **Atom**s. The routine puts standard targets at the front of the list, and the list contains an additional *count* atoms at the end that are allocated but not filled in.

Related Information

XmQTtransfer(3), **XmTransferDone**(3), **XmTransferValue**(3), **XmeClipboardSink**(3), **XmeClipboardSource**(3), **XmeConvertMerge**(3), **XmeDragSource**(3), **XmeDropSink**(3), **XmeGetEncodingAtom**(3), **XmePrimarySink**(3), **XmePrimarySource**(3), **XmeSecondarySink**(3), **XmeSecondarySource**(3), **XmeSecondaryTransfer**(3), **XmeStandardConvert**(3), and **XmeTransferAddDoneProc**(3).

XmeToHorizontalPixels

Purpose

Converts from real-world dimensions to pixels

Synopsis

#include <Xm/XmP.h>

XmImportOperator XmeToHorizontalPixels(
    Widget *widget***,**
    int *offset***,**
    XtArgVal ****value*);**

Description

XmeToHorizontalPixels converts a horizontal distance from a real-world dimensional unit (such as thousandths of an inch) to pixels. This conversion is based on the horizontal resolution of the screen. The choice of real-world dimensional unit is stored in the **XmNunitType** resource.

For example, suppose that *widget* has an **XmNunitType** resource value of **Xm1000TH_INCHES**. If the input *value* were 2000 (meaning 2000/1000 inches or 2 inches), then **XmeToHorizontalPixels** determines how many horizontal pixels fall within 2 inches. Because different screens have different resolutions, the returned *value* might be 160 pixels on one screen and 200 pixels on another.

XmeToHorizontalPixels is an **XmImportProc**.

Typically, your widget will call this function from a synthetic resource record.

widget	Specifies the widget containing the resource named by *offset*.
offset	Specifies the offset (in bytes) of a synthetic resource field in the widget record.
value	Specifies a value in the **XmNunitType** real-world dimensional units used by *widget* and returns a value in pixels.

Return Values

Returns one of the following **XmImportOperator** values:

XmSYNTHETIC_NONE
: The caller of the **XmImportProc** is not responsible for copying the converted *value* into the resource specified by *offset*.

XmSYNTHETIC_LOAD
: The caller of the **XmImportProc** is responsible for copying the converted *value* into the resource specified by *offset*.

Motif's synthetic resource mechanism is typically the caller of **XmeToHorizontalPixels**. Therefore, if **XmeToHorizontalPixels** returns **XmSYTHETIC_LOAD**, Motif synthetic resource mechanism will take care of copying (and casting) *value* into the resource symbolized by *offset*.

Related Information

XmeFromHorizontalPixels(3), **XmeFromVerticalPixels**(3), and **XmeToVerticalPixels**(3).

XmeToVerticalPixels

Purpose

Converts from real-world dimensions to pixels

Synopsis

#include <Xm/XmP.h>

XmImportOperator XmeToVerticalPixels(
    Widget *widget***,**
    int *offset***,**
    XtArgVal ****value*);**

Description

XmeToVerticalPixels converts a vertical distance from a real-world dimensional unit (such as thousandths of an inch) to pixels. This conversion is based on the vertical resolution of the screen. The choice of real-world dimensional unit is stored in the **XmNunitType** resource.

For example, suppose that *widget* has an **XmNunitType** resource value of **Xm1000TH_INCHES**. If the input *value* were 2000 (meaning 2000/1000 inches or 2 inches), then **XmeToVerticalPixels** determines how many vertical pixels fall within 2 inches. Because different screens have different resolutions, the returned *value* might be 160 pixels on one screen and 200 pixels on another.

Typically, your widget will call this function in a synthetic resource record.

XmeToVerticalPixels is an **XmImportProc**.

widget	Specifies the widget.
offset	Specifies the offset (in bytes) of a synthetic resource field in the widget record.
value	Specifies a value in the **XmNunitType** real-world dimensional units used by *widget* and returns a value in pixels.

Return Values

Returns one of the following **XmImportOperator** values:

XmSYNTHETIC_NONE
The caller of the **XmImportProc** is not responsible for copying the converted *value* into the resource specified by *offset*.

XmSYNTHETIC_LOAD
The caller of the **XmImportProc** is responsible for copying the converted *value* into the resource specified by *offset*.

Motif's synthetic resource mechanism is typically the caller of **XmeToVerticalPixels**. Therefore, if **XmeToVerticalPixels** returns **XmSYTHETIC_LOAD**, Motif synthetic resource mechanism will take care of copying (and casting) *value* into the resource symbolized by *offset*.

Related Information

XmeFromHorizontalPixels(3), **XmeFromVerticalPixels**(3), and **XmeToHorizontalPixels**(3).

XmeTraitGet

Purpose

Returns the trait record associated with a given object

Synopsis

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

XtPointer XmeTraitGet(
        XtPointer object,
        XrmQuark trait);
```

Description

Use **XmeTraitGet** to determine whether or not a given *object* has installed a given *trait*. (Widgets install traits by calling **XmeTraitSet**.) If *object* has installed *trait*, you can use **XmeTraitGet** to get a pointer to the associated trait record. After obtaining this pointer, you can call the trait methods of *trait*.

object	Specifies the object (typically, a widget class) that you are inquiring about.
trait	Specifies the trait name. The trait name must be an *XrmQuark* value; for example, *XmQTaccessTextual*.

Return Values

If *trait* was installed on *object*, **XmeTraitGet** returns a pointer to the associated trait record. Otherwise, it returns *NULL*.

Related Information

XmeTraitSet(3).

XmeTraitSet

Purpose Installs a trait on a specified object

Synopsis **#include <Xm/TraitP.h>**

Boolean XmeTraitSet(
    XtPointer *object***,**
    XrmQuark *trait***,**
    XtPointer *trait_record***);**

Description

XmeTraitSet installs *trait* on *object*. Typically, **XmeTraitSet** is called from a widget’s **class_part_initialize** method. If it is, Motif propagates the trait to all subclasses of *object*.

object Specifies the object. Typically, the object is a widget class.

trait Specifies the *XrmQuark* value symbolizing the trait. Here is the list of *trait* quarks defined by the Motif toolkit:

XmQTaccessTextual
The *object* can display one primary text parcel.

XmQTactivatable
The *object* can become a command button in a DialogBox.

XmQTcareParentVisual
The *object* can borrow its parent’s visual information.

XmQTcontainer
The *object* can manage children holding the *XmQTcontainer* trait.

XmQTcontainerItem
The *object* can be managed by a widget holding the **XmQTcontainer** trait.

XmQTdialogShellSavvy
The *object* can become a child of **XmDialogShell**.

XmQTjoinSide
The *object* can attach itself to one side of a suitable parent.

XmQTmenuSavvy
The *object* can become a menu child.

XmQTmenuSystem
The *object* can manage a menu system.

XmQTnavigator
The *object* can act as a navigator to a scrollable widget.

XmQTscrollFrame
The *object* can handle one or more widgets holding the *XmQTnavigator* trait.

XmQTspecifyRenderTable
The *object* can supply the names of its default render tables.

XmQTtakesDefault
The *object* can change its appearance to show that it is the default button.

XmQTtransfer
The *object* can serve as either the source or the destination (or possibly both) of a data transfer operation

trait_record Specifies the name of the structure that holds the trait information. Specifying *NULL* turns off the trait for *object* and its subclasses.

Return Values

Returns *True* if the trait was succesfully installed on *object*. Otherwise, it returns *False*. One possible reason for failure is that the *trait_record* was not declared.

Related Information

XmeTraitGet(3).

XmeTransferAddDoneProc

Purpose

A toolkit function that establishes a procedure to be called when data transfer is complete

Synopsis

#include <Xm/TransferP.h>

void XmeTransferAddDoneProc(
    XtPointer *transfer_id***,**
    XmSelectionFinishedProc *done_proc***);**

Description

XmeTransferAddDoneProc establishes a procedure to be called by the toolkit when a data transfer operation is complete. This routine can be called from an **XmNdestinationCallback** procedure or from any function called as a result, including the following:

- From the selection procedures called as a result of calls to *XmTransferValue*,
- From a **destinationProc** trait method; **destinationProc** is one of the trait methods of the *XmQTtransfer* trait.

If more than one such procedure has been established for a given transfer operation, all procedures are called when the transfer operation is complete.

The *done_proc* argument holds a function of type **XmSelectionFinishedProc**, which has the following definition:

void (* XmSelectionFinishedProc)(
    Widget *widget***,**
    XtEnum *operation***,**
    XmTransferDoneCallbackStruct ***call_data***);**

widget	Specifies the widget that requested the conversion.
operation	Specifies the transfer operation. Possible values are **XmMOVE**, **XmCOPY**, **XmLINK**, and **XmOTHER**.

call_data	Specifies a pointer to an **XmTransferDoneCallbackStruct** structure containing information about the status of the transfer.

XmTransferDoneCallbackStruct is defined as follows:

```
typedef struct {
        int reason;
        XEvent *event;
        Atom selection;
        XtPointer transfer_id;
        XmTransferStatus status;
        XtPointer client_data;
} XmTransferDoneCallbackStruct;
```

reason	Indicates why the callback was invoked.
event	Points to the *XEvent* that triggered the callback. It can be *NULL*.
selection	Indicates the selection being converted.
transfer_id	Specifies a unique indentifier for the data transfer operation.
status	Indicates whether or not the transfer completed successfully. Following are the possible values:

XmTRANSFER_DONE_SUCCEED
The transfer completed successfully.

XmTRANSFER_DONE_FAIL
The transfer did not complete successfully.

client_data	Specifies data to be passed to the callback procedure.

The **XmeTransferAddDoneProc** routine takes the following arguments:

transfer_id	Specifies a unique indentifier for the data transfer operation. The value must be the same as the value of the *transfer_id* member of the **XmDestinationCallbackStruct** passed to the **XmNdestinationCallback** procedure or the **destinationProc** *XmQTtransfer* trait method.
done_proc	Specifies a procedure to be called when the data transfer is complete. The procedure is of type **XmSelectionFinishedProc**.

Related Information

XmQTtransfer(3), **XmTransferDone**(3), **XmTransferValue**(3), **XmeClipboardSink**(3), **XmeClipboardSource**(3), **XmeConvertMerge**(3), **XmeDragSource**(3), **XmeDropSink**(3), **XmeGetEncodingAtom**(3), **XmePrimarySink**(3), **XmePrimarySource**(3), **XmeSecondarySink**(3), **XmeSecondarySource**(3), **XmeSecondaryTransfer**(3), **XmeStandardConvert**(3), and **XmeStandardTargets**(3).

XmeVirtualToActualKeysyms

Purpose

Finds the physical keysyms associated with a given virtual keysym.

Synopsis

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

int XmeVirtualToActualKeysyms(
        Display *display,
        KeySym virtualKeysym,
        XmKeyBinding *physicalKeysyms);
```

Description

XmeVirtualToActualKeysyms determines which physical keysym(s) and modifier(s) are associated with a given virtual keysym. A virtual keysym might match multiple physical keysyms. Therefore, **XmeVirtualToActualKeysyms** returns the number of physical keysyms matched. The returned physical keysym and modifiers are stored in an array of **XmKeyBinding** structures.

The **XmeVirtualToActualKeysyms** function allocates dynamic memory to hold the *physicalKeysyms* array. Your widget is responsible for freeing the memory used to hold this array. The amount of memory is equal to the number of physical keysyms that are matched times the size of one **XmKeyBinding** structure.

display Specifies the display.

virtualKeysym
Specifies the virtual keysym that is to be translated.

physicalKeysym
Returns a pointer to an array of **XmKeyBinding**. The **XmKeyBinding** data type is a structure defined as follows:

```
typedef struct {
        KeySym keysym;
        Modifiers modifiers;
} *XmKeyBinding;
```

Return Values

Returns the number of physical keysyms that correspond to *virtualKeysym*. A returned value of 0 indicates that no physical keysyms matched *virtualKeysym*.

Related Information

VirtualBindings(3).

XmeWarning

Purpose

Writes a warning message to the standard error stream

Synopsis

#include <Xm/XmP.h>

void XmeWarning(
    Widget *widget*,
    char ***message*);**

Description

XmeWarning writes a warning message to standard error. This warning message is purely informational; calling **XmeWarning** does not kill the client.

The warning message has the following format:

```
Warning:
Name: name of object detecting this warning
Class: name of widget class detecting this warning
message
```

widget	Specifies the widget claiming responsibility for this warning message.
message	Specifies the warning message. This message can span multiple lines. The entire warning (everything from the word **Warning:** to the last character in *message*) cannot exceed 1024 characters in length.

Related Information

XtAppWarningMsg(3).

Chapter 17

Traits Reference Pages

This chapter contains the reference pages for all Motif traits.

XmQTaccessTextual

Purpose

A widget holding this trait can display textual data

Synopsis

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

typedef struct {
        int version;
        XmAccessTextualGetValuesProc getValue;
        XmAccessTextualSetValuesProc setValue;
        XmAccessTextualPreferredProc preferredFormat;
} XmAccessTextualTraitRec, *XmAccessTextualTrait;

XtPointer (*XmAccessTextualGetValuesProc)(
         Widget,
         int);
void (*XmAccessTextualSetValuesProc)(
         Widget,
         XtPointer,
         int);
int (*XmAccessTextualPreferredProc)(
         Widget);
```

Description

A widget holding the *XmQTaccessTextual* trait is capable of rendering one primary parcel of textual data. The primary parcel may consist of many lines (like in **XmText**) or only one line (like in **XmTextField**). However, if the widget renders more than one parcel of textual data (as does **XmList**), then the widget should not hold this trait.

The following standard Motif primitive widgets hold this trait:

- **XmText**
- **XmTextField**

- **XmLabel** and all its subclasses

The following standard Motif manager widgets examine their child widgets for this trait:

- **XmComboBox**
- **XmNotebook**
- **XmSelectionBox**
- **XmSpinBox**

(See the source code for the **ExmString** demonstration widget for an example use of this trait.)

The getValue Trait Method

XtPointer getValue(
    Widget *widget***,**
    int *format***);**

The **getValue** trait method returns a copy of the textual data associated with widget *widget* in the format specified by *format*.

widget	Specifies the widget holding this trait.
format	Specifies the string format in which the method should return the data. The *format* must be one of the following: **XmFORMAT_XmSTRING**, **XmFORMAT_MBYTE**, or **XmFORMAT_WCS**.

This method returns a copy of the textual data in the format specified by *format*. The caller is responsible for managing the space allocated by the copy. If the *format* is **XmFORMAT_XmSTRING**, then the method returns an **XmString** cast to an **XtPointer**. If the *format* is **XmFORMAT_MBYTE** or **XmFORMAT_WCS**, then the method returns a **char*** cast to an **XtPointer**.

The setValue Trait Method

void setValue(
    Widget *widget***,**
    XtPointer *str***,**
    int *format***);**

A parent widget calls the **setValue** method to set the string data associated with child widget *widget*. The parent specifies both the string data (*str*) and the *format* of that string data.

widget	Specifies the widget holding this trait.
str	Specifies the string data to be associated with the child widget.
format	Specifies the format of the string data. The format must be one of the following: **XmFORMAT_XmSTRING**, **XmFORMAT_MBYTE**, or **XmFORMAT_WCS**.

The preferredFormat Trait Method

int preferredFormat(
    Widget *widget***);**

A parent widget calls the **preferredFormat** method to find the format in which the child widget prefers to receive its string data.

widget	Specifies the widget holding this trait.

This method returns the preferred string format of the child widget. The returned format must be one of the following: **XmFORMAT_XmSTRING**, **XmFORMAT_MBYTE**, or **XmFORMAT_WCS**.

Related Information

ExmString(3).

XmQTactivatable

Purpose

A widget holding this trait will be treated as a command button in a DialogBox or as a major tab in a Notebook

Synopsis

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

typedef struct {
        int version;
        XmActivatableCallBackProc changeCB;
} XmActivatableTraitRec, *XmActivatableTrait;

void (*XmActivatableCallBackProc)(
         Widget,
         XtCallbackProc,
         XtPointer,
         Boolean);
```

Description

A widget holding the *XmQTactivatable* trait wishes to be treated as a command button when its parent widget is a DialogBox. The standard Motif DialogBox widgets (**XmMessageBox**, **XmSelectionBox**, and **XmFileSelectionBox**) place child widgets having this trait at the bottom of the DialogBox widget, underneath the separator.

In addition, if the **XmNotebook** widget is the parent of a widget having this trait, then **XmNotebook** sets the default value of the **XmNnotebookChildType** constraint to **XmMAJOR_TAB**.

The following standard Motif primitive widgets and gadgets hold this trait:

- **XmPushButton**
- **XmDrawnButton**
- **XmArrowButton**
- **XmPushButtonGadget**

- **XmArrowButtonGadget**

In addition, the following standard Motif manager widgets examine their children for this trait:

- **XmMessageBox**
- **XmSelectionBox**
- **XmFileSelectionBox**
- **XmNotebook**

The changeCB Trait Method

void changeCB(
  Widget *widget***,**
  XtCallbackProc *activCB***,**
  XtPointer *clientData***,**
  Boolean *setunset***);**

The **changeCB** trait method is responsible for adding or removing callback *activCB* from the list of callbacks. A child widget holding this trait should provide a callback resource that is called when the widget is activated. (This resource is typically named **XmNactivateCallback**; however, you could give this resource a different name if you prefer.)

widget	Specifies the child widget that is to have its list of callbacks modified.
activCB	Specifies the callback that is to be added or removed from the list of callbacks.
clientData	Specifies additional data to be passed to the callback.
setunset	Specifies a Boolean value. If this value is *True*, the **setValue** trait method adds *activCB* to the list of callbacks. If this value is *False*, the *activCB* is removed from the list of callbacks.

Related Information

XmMessageBox(3), **XmSelectionBox**(3), **XmFileSelectionBox**(3), **XmNotebook**(3), **XmPushButton**(3), **XmDrawnButton**(3), **XmArrowButton**(3), **XmDrawnButtonGadget**(3), **XmArrowButtonGadget**(3), and **ExmCommandButton**(3).

XmQTcareParentVisual

Purpose

A child widget holding this trait wants to be notified whenever its parent's visual state changes

Synopsis

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

typedef struct {
        int version;
        XmCareVisualRedrawProc redraw;
} XmCareVisualTraitRec, *XmCareVisualTrait;

Boolean (*XmCareVisualRedrawProc)(
         Widget,
         Widget,
         Widget,
         Mask);
```

Description

The appearance of many primitive widgets depends on the colors and pixmaps of their parents. For example, Motif button widgets use the background color and background pixmap of their parents in order to unhighlight themselves. Therefore, Motif primitive widgets that depend on their parent's appearance need to be alerted whenever their parent's appearance changes. The alerting mechanism is the *XmQTcareParentVisual* trait. A child widget holding this trait wants to be notified whenever there is a change in its parent's visual appearance that may affect it.

All Manager widgets must notify their children whenever the Manager's visual appearance changes. The **setValues** method of the **XmManager** widget takes care of this notification for all of its subclasses. Therefore, if you are writing a subclass of Manager, you ordinarily do not have to provide any code to notify children of these visual changes. However, Manager does not notify its children of changes to SelectColor. Therefore, if you are writing a subclass of Manager that needs to notify its children about changes to SelectColor, then your subclass of Manager will need to call its children's *redraw* trait method.

The **XmPrimitive** widget and all its subclasses hold this trait. Therefore, if you are writing a subclass of Primitive, the *XmQTcareParentVisual* trait will automatically be installed on it.

The redraw Trait Method

Boolean redraw(
    Widget *childWidget***,**
    Widget *currentParentWidget***,**
    Widget *newParentWidget***,**
    Mask *visualChangeMask***);**

The *redraw* trait method of **XmPrimitive** examines the *visualChangeMask* argument to see what has changed in the visual appearance of its parent. If the parent's background pixmap or background color has changed, then the *redraw* trait method invokes the **primitive.border_unhighlight** method (if unhighlighting is necessary).

If you are writing a primitive widget, you do not have to provide a *redraw* trait method *unless* your primitive widget depends on its parent's appearance in a non-default way. In this case, your *redraw* trait method should probably envelop the *redraw* trait method of **XmPrimitive**.

childWidget Specifies the child widget that is affected by visual changes in its parent.

currentParentWidget
Specifies the current parent widget.

newParentWidget
Specifies the new parent widget.

visualChangeMask
Specifies a bit mask. Each bit in the mask represents a different visual attribute of the parent. If the bit is set, then that visual attribute has changed. The bit mask may consist of any combination of the following constants:

VisualForeground
Specifies a visual change in the parent's foreground color.

VisualHighlightPixmap
Specifies a visual change in the parent's highlight pixmap.

VisualHighlightColor
Specifies a visual change in the parent’s highlight color.

VisualBottomShadowPixmap
Specifies a visual change in the parent’s bottom shadow pixmap.

VisualBottomShadowColor
Specifies a visual change in the parent’s bottom shadow color.

VisualSelectColor
Specifies a visual change in the parent’s select color. (This is the only visual change that **XmManager** does not process.)

VisualTopShadowPixmap
Specifies a visual change in the parent’s top shadow pixmap.

VisualTopShadowColor
Specifies a visual change in the parent’s top shadow color.

VisualBackgroundPixel
Specifies a visual change in the parent’s background color.

VisualBackgroundPixmap
Specifies a visual change in the parent’s background pixmap.

The **NoVisualChange** constant symbolizes that nothing has changed.

This method returns a Boolean value. A returned value of *True* means that the child needs to be redrawn. A returned value of *False* means that the child does not need to be redrawn.

Related Information

XmManager(3) and **XmPrimitive**(3).

XmQTcontainer

Purpose

A widget holding this trait can manage widgets holding the XmQTcontainerItem trait

Synopsis

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

typdef struct {
        Mask valueMask;
        Cardinal *detail_order;
        Cardinal detail_order_count;
        XmTabList detail_tablist;
        Dimension first_column_width;
        unsigned char selection_mode;
        Pixel select_color;
} XmContainerDataRec, *XmContainerData;

typdef struct {
        int version;
        XmContainerGetValuesProc getValues;
} XmContainerTraitRec, *XmContainerTrait;

void (*XmContainerGetValuesProc)(
         Widget,
         XmContainerData);
```

Description

A widget holding the **XmQTcontainer** trait is capable of acting as a container manager to suitable container child widgets. The only standard Motif widget to hold this trait is **XmContainer**. **XmIconGadget** is the only standard Motif widget that asks other widgets if they hold this trait.

If you are writing your own container widget, then it must hold the *XmQTcontainer* trait. Similarly, if you are writing your own container child widget, then it must call the **getValues** trait method of *XmQTcontainer* to determine its own geometry and to render its visuals.

The getValues Trait Method

void getValues(
  Widget *containerWidget***,**
  XmContainerData *containerData***);**

A container child widget calls the **getValues** trait method of its parent container. The information returned by the container helps a container child determine its geometry and render itself.

containerWidget
  Specifies the container widget holding this trait.

containerData
  The caller (that is, the container child widget) must pass an **XmContainerData** structure as an argument. The caller must supply a value for the first field of the structure, *valueMask*, to indicate which fields the caller is interested in. The callee (that is, the container widget) must return a value for all the fields of the structure that are indicated by *valueMask*. In addition, the callee can optionally return values for fields not marked by *valueMask*. Following is a description of each of the fields of the **XmContainerData** structure:

  valueMask The caller supplies a value to indicate which fields it is interested in. Valid values are as follows:

  ContAllValid
    The container widget must return a value for all fields.

  ContDetailOrder
    The container widget must return a value for the *detail_order* and *detail_order_count* fields.

  ContDetailTabList
    The container widget must return a value for the *detail_tablist* field.

  ContFirstColumnWidth
    The container widget must return a value for the *first_column_width* field.

ContSelectionMode
The container widget must return a value for the *selection_mode* field.

ContSelectColor
The container widget must return a value for the *select_color* field.

detail_order The container widget returns an array of **Cardinal**s that specify the detail columns to be rendered and their positions. If the container widget is **XmContainer**, then the returned value corresponds to the **XmNdetailOrder** resource of **XmContainer**. The data returned must not be modified or freed.

detail_order_count
The container widget returns a count of the number of **Cardinal**s in the *detail_order* field. If the *detail_order* field is *NULL* and the container widget returns 0, then no details should be displayed by the caller. However, if the *detail_order* field is *NULL* and the container widget returns a nonzero value, the caller must interpret the value as the size of a virtual *detail_order* array [1,2..n]. If the container widget is **XmContainer**, then the returned value corresponds to the **XmNdetailOrderCount** resource of **XmContainer**.

detail_tablist
The container widget must return an **XmTabList** specifying the start of each column in the **XmDETAIL** view. If the container widget is **XmContainer**, the returned *detail_tablist* will be the value of the **XmNdetailTabList** resource. The data returned must not be modified or freed.

first_column_width
The container widget must return the width of the first column to be used for rendering the label and pixmap of each container child widget. The returned width must include the indentation to be used for each level of the outline. If the container widget is **XmContainer**, the value

of the *first_column_width* corresponds to the value of the *XmNfirstColumnWidth* resource.

selection_mode
The container returns the selection mode currently in use. The possible values are **XmNORMAL_MODE** and **XmADD_MODE**. The caller must draw its border highlight with a solid line in normal mode and with a dashed line in add mode.

select_color The container widget returns the background color to be used by the caller when it redisplays itself in the selected state. If the container widget is **XmContainer**, the value of *select_color* corresponds to the value of the **XmNselectColor** resource.

Related Information

ExmGrid(3), **XmContainer**(3), **XmIconGadget**(3), **XmQTcontainerItem**(3), **XmSeparator**(3), and **XmTabList**(3).

XmQTcontainerItem

Purpose

A widget holding this trait can serve as a child of a widget holding the XmQTcontainer trait

Synopsis

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

typedef struct {
        Mask valueMask;
        unsigned char view_type;
        unsigned char visual_emphasis;
        Dimension icon_width;
        Cardinal detail_count;
} XmContainerItemDataRec, *XmContainerItemData;

typedef struct {
        int version;
        XmContainerItemSetValuesProc setValues;
        XmContainerItemGetValuesProc getValues;
} XmContainerItemTraitRec, *XmContainerItemTrait;

void (*XmContainerItemSetValuesProc)(
         Widget,
         XmContainerItemData);
void (*XmContainerItemGetValuesProc)(
         Widget,
         XmContainerItemData);
```

Description

A widget holding the *XmQTcontainerItem* trait can serve as a child to a widget holding the *XmQTcontainer* trait.

XmIconGadget is the only standard Motif class holding the *XmQTcontainerItem* trait. **XmContainer** is the only standard Motif widget that examines its children for this trait.

The setValues Trait Method

void setValues(
  Widget *containerChildWidget***,**
  XmContainerItemData *containerItemData***);**

A container widget (such as **XmContainer**) calls the **setValues** trait method of its *containerChildWidget* in order to set certain visual properties.

containerChildWidget
  Specifies the container child widget that is holding this trait.

containerItemData
  Specifies an **XmContainerItemData** structure. This structure holds five fields, but only the first three fields are meaningful to the **setValues** trait method. Following is a description of those three fields:

valueMask Specifies a bit mask. Each bit in the bit mask symbolizes a visual property affected by the **setValues** trait method. If a particular bit is set to 1, **setValues** modifies the associated visual property. If this same bit holds a 0, **setValues** does not modify the associated visual property. To help you set bits, the **ContItemT.h** header file provides three relevant constants.

ContItemViewType
  Sets the *view_type* visual property bit to 1.

ContItemVisualEmphasis
  Sets the **setValues** visual property bit to 1.

ContItemAllValid
  Sets both visual property bits to 1.

view_type Specifies the view type of *widget*. For example, when **XmContainer** calls **setValues**, it passes a *view_type* value of either **XmLARGE_ICON** or **XmSMALL_ICON**. (See the **XmIconGadget**(3) reference page for a description of **XmLARGE_ICON** and **XmSMALL_ICON**.)

visual_emphasis

Specifies the visual emphasis of *widget*. For example, when **XmContainer** calls **setValues**, it passes a *visualEmphasis* value of either **XmSELECTED** or **XmNOT_SELECTED**. See the **XmIconGadget**(3) reference page for a description of **XmSELECTED** and **XmNOT_SELECTED**.

The getValues Trait Method

void getValues(
    Widget *containerChildWidget***,**
    XmContainerItemData *containerItemData***);**

A container widget calls the **getValues** trait method of its *containerChildWidget* to determine what visual properties the *containerChildWidget* provides.

containerChildWidget

Specifies the child widget that is holding this trait.

containerItemData

Specifies an **XmContainerItemData** structure. This structure holds the following five fields:

valueMask Specifies a bit mask. Each bit in the bit mask symbolizes a visual property returned by the **getValues** trait method. If the caller sets a particular bit to 1, **getValues** returns meaningful information about the associated visual property. However, if the caller places a 0 in this same bit, then **getValues** does not return meaningful information about the associated visual property. In short, the *valueMask* lets the caller specify the visual properties that it is interested in. To help you set bits, the **ContItemT.h** header file provides the following five constants:

ContItemViewType

Sets the *view_type* visual property bit to 1.

ContItemVisualEmphasis

Sets the *visual_emphasis* visual property bit to 1.

ContItemIconWidth
Sets the **icon_width** visual property bit to 1.

ContItemDetailCount
Sets the *detail_count* visual property bit to 1.

ContItemAllValid
Sets all four visual property bits to 1.

view_type Returns the view type of *childContainerWidget*. If **XmContainer** is managing *containerChildWidget*, **XmContainer** expects that the returned value will be either **XmLARGE_ICON** or **XmSMALL_ICON**. See the **XmIconGadget**(3) reference page for a description of these two values.

visual_emphasis
Returns the visual emphasis of *childContainerWidget*. If **XmContainer** is managing *widget*, **XmContainer** expects that the returned value will be either **XmSELECTED** or **XmNOT_SELECTED**. (See the **XmIconGadget**(3) reference page for a description of these two values.)

icon_width Returns the width of the icon. (This width will not include the width of any associated detail strings.)

detail_count Returns the number of details strings in this container child.

Related Information

ExmGrid(3), **XmContainer**(3), **XmIconGadget**(3), and **XmQTcontainer**(3).

XmQTdialogShellSavvy

Purpose

A dialog widget holding this trait can become a child of the XmDialogShell widget

Synopsis

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

typedef struct {
        int version;
        XmDialogSavvyMapUnmapProc callMapUnmapCB;
} XmDialogSavvyTraitRec,*XmDialogSavvyTrait;

void (*XmDialogSavvyMapUnmapProc)(
         Widget,
         Boolean);
```

Description

A widget holding the *XmQTdialogSavvy* trait can become an immediate child of the **XmDialogShell** widget. In other words, the *XmQTdialogSavvy* trait announces to the **XmDialogShell** widget that your widget is an acceptable child.

Every dialog widget that holds the *XmQTdialogShellSavvy* trait must provide the following:

- A **callMapUnmapCB** trait method
- A default position resource
- Code that detects an **XmDIALOG_SAVVY_FORCE_ORIGIN** situation

Any widget that installs the *XmQTdialogShellSavvy* trait must provide a **Boolean** default position resource. For example, **ExmGrid** and **XmBulletinBoard** both provide a **Boolean** default position resource named **XmNdefaultPosition**. This resource controls the positioning of the DialogShell managing your *XmQTdialogShellSavvy* widget. This resource has no influence if your *XmQTdialogShellSavvy* widget is not managed by a DialogShell. For example, if the parent of the DialogShell is an ApplicationShell, then the center of the DialogShell will be at the same coordinates as the center of the ApplicationShell. If the DialogShell becomes unmapped (but stays managed) and then remapped, this resource has no influence on the DialogShell's position. If the default position resource is False, the DialogShell does not automatically center itself. Instead, the DialogShell (and therefore its *XmQTdialogShellSavvy* child) will be positioned according to the values of **XmNx** and **XmNy**. Motif will treat the values of **XmNx** and **XmNy** as offsets from the upper-left corner of the screen (rather than as offsets from the upper-left corner of the parent shell).

The **Xm/DialogSavvyT.h** header file provides a special macro constant named **XmDIALOG_SAVVY_FORCE_ORIGIN**. Any widget holding the *XmQTdialogShellSavvy* trait must use this constant. Here is the problem that **XmDIALOG_SAVVY_FORCE_ORIGIN** solves. The current position of a dialog child widget within an **XmDialogShell** widget is always 0,0. Suppose a user or application calls **XtSetValues** to set the dialog child widget's x-coordinate or y-coordinate to 0. In this case, the Intrinsics will not detect a geometry change and will therefore not trigger a geometry request. To tell the **XmDialogShell** widget that you really do want the child to move to a coordinate of 0, your dialog child widget must catch this request and respond to it by setting the x-coordinate or y-coordinate to **XmDIALOG_SAVVY_FORCE_ORIGIN** instead of 0. For example:

```
if (my_dialog_widget->core.x == 0)
  my_dialog_widget->core.x = XmDIALOG_SAVVY_FORCE_ORIGIN;
...
if (my_dialog_widget->core.y == 0)
  my_dialog_widget->core.y = XmDIALOG_SAVVY_FORCE_ORIGIN;
```

In the standard Motif widget set, the **XmBulletinBoard** widget and all its subclasses hold the *XmQTdialogShellSavvy* trait. In the Exm demonstration widget set, the *ExmGrid* widget installs the *XmQTdialogShellSavvy* trait.

The **XmDialogShell** widget is the only standard Motif widget that examines its children for this trait.

The callMapUnmapCB Trait Method

void callMapUnmapCB(
    Widget *dialogWidget***,**
    Boolean *map_unmap***);**

All dialog widgets holding the *XmQTdialogShellSavvy* trait must provide the **callMapUnmapCB** trait method. The **callMapUnmapCB** trait method is responsible for calling the map or unmap callback of the widget. For example, following is one possible way of implementing this trait method:

```
static void
CallMapUnmap(
         Widget dialogWidget,
         Boolean map_unmap)
{
 ExmMyDialogWidget dw = (ExmMyDialogWidget) dialogWidget;
 XmAnyCallbackStruct call_data;
   call_data.reason = map_unmap ? XmCR_MAP: XmCR_UNMAP;
   call_data.event  = NULL;
   if (map_unmap)
     XtCallCallbackList (dialogWidget, dw->my_dialog.map_callback,
                         &call_data);
   else
     XtCallCallbackList (dialogWidget, dw->my_dialog.unmap_callback,
                         &call_data);
}
```

dialogWidget
Specifies the dialog widget.

map_unmap Specifies a **Boolean** value. If this value is *True*, then the trait method should invoke the map callback. If this value is *False*, then the trait method should invoke the unmap callback.

Related Information

XmBulletinBoard(3) and **XmDialogShell**(3).

XmQTjoinSide

Purpose

A child widget holding this trait can be attached to one side of a suitable parent widget

Synopsis

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

typedef struct {
        int version;
        XmJoinSideSetValueProc setValue;
        XmJoinSideGetValueProc getValue;
} XmJoinSideTraitRec, *XmJoinSideTrait;

void (*XmJoinSideSetValueProc)(
         Widget,
         unsigned char,
         Dimension);
unsigned char (*XmJoinSideGetValueProc)(
         Widget,
         Dimension);
```

Description

A child widget holding the *XmQTjoinSide* trait knows how to affix itself to one side of its parent (assuming that the parent supports this trait). For example, the demonstration widget *ExmTabButton* can affix itself to one side of the **XmNotebook** widget. The affixed widget resembles a handle coming out of one side of the parent.

The join side is the side of the child widget that is joined to the parent. For example, if the join side of the child is the left side, then the left side of the child is affixed to the right side of the parent.

The join thickness is the shadow thickness of the parent. For example, if the parent has a shadow thickness of 4 pixels, then the join thickness must be 4 pixels. The child is responsible for providing visuals that make it appear that the child is attached or "growing out of" the parent. To do this, the child and parent must share a common join thickness on the join side. In other words, the child is responsible for drawing a visual "handle" that is as thick as the parent's shadow. The parent is responsible for telling the child what the join thickness is. The parent does this by calling the child's **setValue** trait method.

Although not required, the child may also want to clear part of the join side. The **ExmTabButton** demonstration widget does this to enhance the illusion that the child is affixed to its parent.

Child widgets holding this trait must provide the **setValue** trait method and should also provide the **getValue** trait method.

Currently, no standard Motif widgets hold this trait, and only the **XmNotebook** widget examines its children for this trait.

The setValue Trait Method

void setValue(
    Widget *childWidget***,**
    unsigned char *join_side***,**
    Dimension *join_thickness***);**

The **setValue** trait method is responsible for assigning the visual characteristics for the side of the child widget that gets joined to its parent.

childWidget Specifies the child widget that is to be joined to its parent.

join_side Specifies the side of the child widget that is to be joined to the parent. The *join_side* must be one of the following: **XmNONE**, **XmLEFT**, **XmRIGHT**, **XmTOP**, or **XmBOTTOM**. Specifying **XmNONE** means that the manager will not provide an open side. In other words, **XmNONE** should make the *childWidget* behave like a widget that does not hold the *XmQTjoinSide* trait.

join_thickness
Specifies the thickness (in pixels) of the shadow on the join side. A thickness of 0 means that the manager and child will not be joined.

The getValue Trait Method

unsigned char getValue(
    Widget *childWidget***,**
    Dimension ****join_thickness*);**

The **getValue** trait method is responsible for returning the visual characteristics for the side of the child widget that gets joined to its parent.

childWidget Specifies the child widget that is to be joined to its parent.

join_thickness
Returns the thickness (in pixels) of the shadow on the join side.

This method must return a constant symbolizing the join side. The returned value must be one of the following: **XmLEFT**, **XmRIGHT**, **XmTOP**, **XmBOTTOM**, or **XmNONE**.

Related Information

ExmTabButton(3) and **XmNotebook**(3).

XmQTmenuSavvy

Purpose

A widget holding this trait can become a menu child

Synopsis

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

typedef struct {
        int version;
        XmMenuSavvyDisableProc disableCallback;
        XmMenuSavvyGetAcceleratorProc getAccelerator;
        XmMenuSavvyGetMnemonicProc getMnemonic;
        XmMenuSavvyGetActivateCBNameProc getActivateCBName;
} XmMenuSavvyTraitRec, *XmMenuSavvyTrait;

void (*XmMenuSavvyDisableProc)(
         Widget,
         XtActivateState);
char *(*XmMenuSavvyGetAcceleratorProc)(
         Widget);
Keysym (*XmMenuSavvyGetMnemonicProc)(
         Widget);
char *(*XmMenuSavvyGetActivateCBNameProc)(
         Void);
```

Description

Menu parent widgets (like **XmRowColumn**) examine their children for the *XmQTmenuSavvy* trait. Only those child widgets holding the *XmQTmenuSavvy* trait can become menu children.

The following standard Motif primitive widgets hold this trait:

- **XmLabel**
- **XmLabelGadget**
- **XmPushButton**

- **XmPushButtonGadget**
- **XmSeparator**
- **XmSeparatorGadget**
- **XmToggleButton**
- **XmToggleButtonGadget**
- **XmCascadeButton**
- **XmCascadeButtonGadget**
- **XmDrawnButton**

Among the standard Motif manager widgets, only **XmRowColumn** examines its children for this trait.

The disableCallback Trait Method

void disableCallback(
  Widget *childWidget***,**
  XtActivateState *enable_disable***);**

The **disableCallback** trait method allows the menu parent (typically, the **XmRowColumn** widget) to enable or disable the activate callback associated with the child. **XmRowColumn** calls this trait method when its **XmNentryCallback** resource is set to a value other than *NULL*.

childWidget Specifies the child widget that holds this trait.

enable_disable
Specifies whether to enable or disable the activate callback method associated with the child widget. There are two possible values for this argument: **XmENABLE_CALLBACK** or **XmDISABLE_CALLBACK**. **XmENABLE_CALLBACK** enables the activate callback associated with *childWidget*; **XmDISABLE_CALLBACK** disables it.

The getAccelerator Trait Method

char *getAccelerator(
  Widget *childWidget***);**

The **getAccelerator** trait method returns the accelerator associated with *childWidget*.

childWidget Specifies a menu child widget.

This trait method returns the accelerator associated with *childWidget*. If there is no accelerator associated with *childWidget*, **getAccelerator** returns *NULL*.

The getMnemonic Trait Method

Keysym getMnemonic(
    Widget *childWidget***);**

The **getMnemonic** trait method returns the mnemonic associated with *childWidget*.

childWidget   Specifies a menu child widget.

This trait method returns the mnemonic associated with *childWidget*. If there is no mnemonic associated with *childWidget*, **getMnemonic** returns *NULL*.

The getActivateCBName Trait Method

char *getActivateCBName(
 void

The **getActivateCBName** trait method returns a pointer to a static string. This string names the resource that contains the activation callback list for this widget. For example, this trait method will return **activateCallback** for **XmPushButton** and **valueChangedCallback** for **XmToggleButton**.

Related Information

XmRowColumn(3), **XmLabel**(3), **XmPushButton**(3), **XmDrawnButton**(3), **XmCascadeButton**(3), **XmToggleButton**(3), **XmPushButtonGadget**(3), **XmToggleButtonGadget**(3), **XmCascadeButtonGadget**(3), and **ExmMenuButton**(3).

XmQTmenuSystem

Purpose

A widget holding this trait can serve as a menu system

Synopsis

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

typedef struct {
        int version;
        XmMenuSystemTypeProc type;
        XmMenuSystemStatusProc status;
        XmMenuSystemCascadeProc cascade;
        XmMenuSystemVerifyProc verifyButton;
        XmMenuSystemControlTraversalProc controlTraversal;
        XmMenuSystemMenuBarCleanupProc menuBarCleanup;
        XmMenuSystemPopdownProc popdown;
        XmMenuSystemPopdownProc buttonPopdown;
        XmMenuSystemReparentProc reparentToTearOffShell;
        XmMenuSystemReparentProc reparentToMenuShell;
        XmMenuSystemArmProc arm;
        XmMenuSystemDisarmProc disarm;
        XmMenuSystemTearOffArmProc tearOffArm;
        XmMenuSystemEntryCallbackProc entryCallback;
        XmMenuSystemUpdateHistoryProc updateHistory;
        XmMenuSystemGetPostedFromWidgetProc getLastSelectToplevel;
        XmMenuSystemPositionProc position;
        XmMenuSystemUpdateBindingsProc updateBindings;
        XmMenuSystemRecordPostFromWidgetProc recordPostFromWidget;
        XmMenuSystemPopdownAllProc popdownEveryone;
        XmMenuSystemChildFocusProc childFocus;
        XmMenuSystemPopupPostedProc getPopupPosted;
} XmMenuSystemTraitRec, *XmMenuSystemTrait;
#define XmMenuSystemTypeProc XmMenuSystemWidgetProc
```

```
#define XmMenuSystemStatusProc XmMenuSystemWidgetProc
#define XmMenuSystemMenuBarCleanupProc XmMenuSystemDisarmProc
#define XmMenuSystemReparentProc XmMenuSystemPositionProc
#define XmMenuSystemArmProc XmMenuSystemDisarmProc
#define XmMenuSystemTearOffArmProc XmMenuSystemDisarmProc
#define XmMenuSystemGetPostedFromWidgetProc XmMenuSystemDisarmProc
#define XmMenuSystemPopdownAllProc XmMenuSystemPositionProc
#define XmMenuSystemChildFocusProc XmMenuSystemDisarmProc

void (*XmMenuSystemCascadeProc)(
          Widget,
          Widget,
          XEvent*);
Boolean (*XmMenuSystemVerifyProc)(
          Widget,
          XEvent*);
void (*XmMenuSystemControlTraversalProc)(
          Widget,
          Boolean);
Boolean (*XmMenuSystemPopdownProc)(
          Widget,
          XEvent*);
void (*XmMenuSystemDisarmProc)(
          Widget);
void (*XmMenuSystemEntryCallbackProc)(
          Widget,
          Widget,
          XtPointer);
Boolean (*XmMenuSystemUpdateHistoryProc)(
          Widget,
          Widget,
          Boolean);
void (*XmMenuSystemPositionProc)(
          Widget,
          XEvent*);
void (*XmMenuSystemUpdateBindingsProc)(
          Widget,
          int);
```

```
void (*XmMenuSystemRecordPostFromWidgetProc)(
        Widget,
        Widget,
        Boolean);
Widget (*XmMenuSystemPopupPostedProc)(
        Widget);
int (*XmMenuSystemWidgetProc)(
        Widget);
```

Description

A widget holding the *XmQTmenuSystem* trait can be configured as a menu system. In the standard Motif widget set, only the **XmRowColumn** widget holds this trait.

The *XmQTmenuSystem* trait provides many trait methods. If you are writing a menu child widget, then you will need to call some of these trait methods. For example, the *ExmMenuButton* demonstration widget uses the following trait methods of *XmQTmenuSystem*:

- **status**
- **childFocus**
- **reparentToTearOffShell**
- **buttonPopdown**
- **getLastSelectTopLevel**
- **entryCallback**
- **verifyButton**
- **controlTraversal**
- **getPopupPosted**
- **tearOffArm**
- **popdownEveryone**
- **popdown**

We do not recommend writing your own menu manager widget.

All the primitive button widgets and button gadgets in the standard Motif widget set call trait methods of *XmQTmenuSystem*.

The type Trait Method

int type(
    Widget *rowColumnWidget***);**

The *type* trait method returns the kind of menu system (for example, a Pulldown menu) that widget *rowColumnWidget* is managing.

rowColumnWidget
: Specifies the parent widget that is managing the menu system. Typically, *rowColumnWidget* is the parent widget of the current widget.

This trait method returns the type of menu. The returned type must be one of the following: **XmWORK_AREA**, **XmMENU_BAR**, **XmMENU_PULLDOWN**, **XmMENU_POPUP**, or **XmMENU_OPTION**.

The status Trait Method

int status(
    Widget *rowColumnWidget***);**

The *status* trait method returns the current status of certain menu operations.

rowColumnWidget
: Specifies the **XmRowColumn** widget that is managing the menu system. Typically, *rowColumnWidget* is the parent widget of the current widget.

This trait method returns a bit mask symbolizing the current status of certain menu operations. The **Xm/XmP.h** header file provides widget writers with the following macros for interpreting the returned mask:

XmIsTorn(*mask*)
: This macro returns a nonzero value if the menu has already been torn off. Otherwise, this macro returns 0.

XmIsTearOffShellDescendant(*mask*)
: This macro returns a nonzero value if the menu is a descendant of the tear-off shell. Otherwise, this macro returns 0.

XmPopupPosted(*mask*)
: This macro returns a nonzero value if the menu is a Popup menu that has already been posted. Otherwise, this macro returns 0.

XmIsInDragMode(*mask*)
: This macro returns a nonzero value if the menu is in drag mode. Otherwise, this macro returns 0.

The cascade Trait Method

void cascade(
  Widget *rowColumnWidget***,**
  Widget *cascadeButtonWidget***,**
  XEvent ****event*);**

We do not recommend writing your own cascade button widget; however, if you do, then your cascade button widget must call the *cascade* trait method. More specifically, your cascade button widget must call *cascade* immediately prior to performing the menu cascade itself. The *cascade* trait method records the data causing the cascade. Furthermore, the trait method positions the submenu. Note that this trait method does not actually perform the menu cascade itself.

rowColumnWidget
Specifies the submenu to post.

cascadeButtonWidget
Specifies the cascade button widget that is about to be cascaded.

event Specifies a pointer to the event causing the menu cascade.

The verifyButton Trait Method

Boolean verifyButton(
  Widget *rowColumnWidget***,**
  XEvent ****event*);**

A child button widget calls the **verifyButton** trait method to determine whether *event* is supposed to cause menu posting. The event that is supposed to cause menu posting is defined by the **XmNmenuPost** resource of the parent **XmRowColumn** widget.

rowColumnWidget
Specifies the parent **XmRowColumn** widget that holds the menu posting event.

event Specifies a pointer to the event received by the child button widget that is to be verified.

This trait method returns *True* if the *event* matches the menu posting specification defined by the **XmNmenuPost** resource of *rowColumnWidget*. Otherwise, it returns *False*.

The controlTraversal Trait Method

void controlTraversal(
  Widget *rowColumnWidget***,**
  Boolean *traverse***);**

We do not recommend writing your own cascade button widget; however, if you do, then your cascade button widget will call **controlTraversal** to control traversal within the MenuBar. This trait method turns menu traversal on or off.

rowColumnWidget
  Specifies the RowColumn widget that will handle traversal.

traverse This is a **Boolean** value. Ordinarily, this should be set to False. Specifying *False* establishes default Motif traversal behavior within a MenuBar, which is to say that traversal is disabled. Specifying *True* enables menu traversal within the MenuBar.

The menuBarCleanup Trait Method

void menuBarCleanup(
  Widget *menuBarWidget***);**

A user may invoke the osfMenuBar action of **XmRowColumn** (typically, by pressing the F10 function key) to activate traversal within the MenuBar. When the user invokes osfMenuBar a second time to deactivate menu traversal, the **XmRowColumn** needs to "clean up" the MenuBar. This cleanup involves lowering any Pulldown or Popup menus. In the standard Motif widget set, **XmCascadeButton** and **XmCascadeButtonGadget** both call the **menuBarCleanup** trait method.

menuBarWidget
  Specifies the menubar widget (typically, an **XmRowColumn** widget) that needs to be cleaned up.

The popdown Trait Method

Boolean popdown(
  Widget *rowColumnWidget***,**
  XEvent ***event***);**

The **popdown** trait method pops down all the Popup menus associated with the *rowColumnWidget*. This trait method is similar to the **buttonPopdown** trait method. Your widget should call **popdown** if *rowColumnWidget* is in a tear-off shell.

rowColumnWidget
: Specifies the parent widget (typically, an **XmRowColumn** widget) that is managing the Popup menus.

event
: Specifies a pointer to the event causing the popdown.

This trait method returns *True* if *rowColumnWidget* was managing any Popup menus. Otherwise, it returns *False*.

The buttonPopdown Trait Method

Boolean buttonPopdown(
  Widget *rowColumnWidget***,**
  XEvent ****event*);**

The **buttonPopdown** trait method pops down all the Popup menus associated with the *rowColumnWidget*. This trait method is similar to the **buttonPopdown** trait method. Your widget should call **popdown** if *rowColumnWidget* is not in a tear-off shell. The **buttonPopdown** trait method provides a slight delay between the time the user presses the button and the popdown occurs. This delay gives the widget time to draw visuals that simulate the button being pressed.

rowColumnWidget
: Specifies the Popup menu widget that needs to be popped down.

event
: Specifies the event that caused the popdown.

This trait method returns *True* if the *rowColumnWidget* was managing any Popup menus. Otherwise, returns *False*.

The reparentToTearOffShell Trait Method

void reparentToTearOffShell(
  Widget *rowColumnWidget***,**
  XEvent ****event*);**

When a user tears off a tear-off menu, the *rowColumnWidget* holding the menu needs to be reparented. In other words, the *rowColumnWidget* that was a child of the MenuBar must now become a child of the tear-off shell. In order to do this, the child button widget calls the **reparentToTearOffShell** trait method.

rowColumnWidget
: Specifies the menu to reparent.

event
: Specifies a pointer to the event that caused the tear off.

The reparentToMenuShell Trait Method

void reparentToMenuShell(
  Widget *rowColumnWidget***,**
  XEvent ****event*);**

When a user collapses a tear-off shell, the *rowColumnWidget* holding the menu needs to be reparented. That is, the menu inside the tear-off shell must become a child of the MenuBar. Therefore, when the user collapses a tear-off shell, the child button widget (typically, a cascade button widget) must call **reparentToMenuShell**.

rowColumnWidget
: Specifies the menu to reparent.

event
: Specifies a pointer to the event causing the tear-off shell to collapse.

The arm Trait Method

void arm(
  Widget *rowColumnWidget***);**

A cascade button widget must call the *arm* trait method as part of its **Arm** or **ArmAndActivate** method. More precisely, a cascade button widget must call *arm* whenever the user posts a Pulldown method.

If *rowColumnWidget* is not already armed, then the *arm* trait method makes current the calling widget and saves the focus widget. If *rowColumnWidget* is already armed, then the *arm* trait method does nothing.

rowColumnWidget
: Specifies the **XmRowColumn** widget to arm. The **XmRowColumn** widget must have the **XmNrowColumnType** resource set to **XmMENU_BAR**.

The disarm Trait Method

void disarm(
  Widget *rowColumnWidget***);**

The *disarm* trait method undoes the changes caused by the *arm* trait method. A cascade button widget must call *disarm* whenever the user unposts a Pulldown menu.

The *disarm* trait method undoes the actions performed by the *arm* trait method. That is, if *rowColumnWidget* is armed, then the *disarm* trait method makes current the active item that had the focus before the menu was armed.

rowColumnWidget
Specifies the menu widget to disarm. The **XmRowColumn** widget must have the **XmNrowColumnType** resource set to **XmMENU_BAR**.

The tearOffArm Trait Method

void tearOffArm(
Widget *rowColumnWidget***);**

Your child button widget should call the **tearOffArm** trait method when both of the following are true:

- If your child's parent was a TearOff Menu container and it has already been torn off
- The user has selected your child widget as the initial selection

The **tearOffArm** trait method places the menu system into an active state by setting up grabs. After doing this, the **tearOffArm** trait method itself calls the **menuArm** trait method. Upon completion of the **menuArm** trait method the **tearOffArm** trait method sets up modal grabs.

rowColumnWidget
Specifies the menu widget to arm.

The entryCallback Trait Method

void entryCallback(
Widget *rowColumnWidget***,**
Widget *childButtonWidget***,**
XtPointer *call_value***);**

A button widget that supports an activate callback must call the **entryCallback** trait method. More specifically, a button widget must call this trait method when the user activates the button. This trait method calls the entry callback of *rowColumnWidget*. The entry callback of the *rowColumnWidget* is defined by the value of the **XmNentryCallback** resource.

rowColumnWidget
Specifies the menu widget.

childButtonWidget
Specifies the child menu widget that the user has activated.

call_value
Specifies the client data to pass to the entry callback.

The updateHistory Trait Method

Boolean updateHistory(
  Widget *rowColumnWidget***,**
  Widget *childButtonWidget***,**
  Boolean *updateOnMemWidgetMatch***);**

We do not recommend calling this trait method.

The *childButtonWidget* should call this trait method from the widget's *initialize* and **set_values** methods. This trait method updates the value of the **XmNmenuHistory** resource of the *rowColumnWidget*.

rowColumnWidget
: Specifies the RowColumn widget that is the parent of your button widget.

childButtonWidget
: Specifies the button widget calling this trait method.

updateOnMemWidgetMatch
: Specifies False.

This trait method returns *True* if *childButtonWidget* is in a Pulldown or Option Menu. Otherwise, it returns *False*.

The getLastSelectToplevel Trait Method

void getLastSelectTopLevel(
  Widget *rowColumnWidget***);**

We do not recommend calling this trait method.

Menu button widgets need to call this trait method immediately prior to arming themselves. That is, a menu button widget should call the **getLastSelectTopLevel** trait method from its **Arm** or **ArmAndActivate** methods.

rowColumnWidget
: Specifies the RowColumn widget that is the parent of your button widget.

The positionMenu Trait Method

void positionMenu(
  Widget *popupMenuPaneWidget***,**
  XButtonPressedEvent ****buttonevent*);**

You may need to call the **positionMenu** trait method from your cascade button widget. The **positionMenu** trait method positions a Popup MenuPane by using the information in the specified event. The *popupMenuPaneWidget* uses the *x_root* and *y_root* values in the specified *buttonevent* to determine the menu position.

Note that the **positionMenu** trait method works almost identically to the **XmMenuPosition**. The only difference between the two is that **XmMenuPosition** requires its first argument to be an **XmRowColumn** widget, but **positionMenu** can accept any menu manager.

popupMenuPaneWidget
Specifies the Popup menu to be positioned. Typically, this will be an **XmRowColumn** widget.

buttonevent Specifies a pointer to the button event that caused the menu to pop up.

The updateBindings Trait Method

void updateBindings(
Widget *widget***,**
int *mode***);**

Your button *widget* must call the **updateBindings** trait when its accelerator or mnemonic changes. Calling this trait method informs the menu parent widget of any changes to accelerators or mnemonics.

widget Specifies a widget. Ordinarily, this will be a menu child widget.

mode Specifies whether an accelerator or mnemonic has been added, removed, or replaced. You should specify one of the following constants: **XmADD**, **XmDELETE**, or **XmREPLACE**.

The recordPostFromWidget Trait Method

void recordPostFromWidget(
Widget *rowColumnWidget***,**
Widget *cascadeButtonWidget***,**
Boolean *attach***);**

We do not recommend writing your own cascade button widget, but if you do, you will need to call the **recordPostFromWidget**. This trait method signals that a MenuPane has either been attached to or detached from a cascade button.

rowColumnWidget
Specifies the RowColumn widget containing the Popup Menu.

cascadeButtonWidget
Specifies a cascade button widget.

attach This is a **Boolean** value. If *True*, the *cascadeButtonWidget* is signaling that a MenuPane has been attached to a cascade button widget. If *False*, the *cascadeButtonWidget* is signaling that a MenuPane has been detached from a cascade button widget.

The popdownEveryone Trait Method

void popdownEveryone(
Widget *menuShellWidget***,**
XEvent ****event*);**

The **popdownEveryone** trait method pops down all the Popup Menus associated with *menuShellWidget*, from bottom to top.

menuShellWidget
Specifies the menu shell widget.

event Specifies a pointer to the event causing the popdown.

The childFocus Trait Method

void childFocus(
Widget *menuChildWidget***);**

The **childFocus** trait method gives *menuChildWidget* the keyboard focus. This trait method performs other special handling to ensure that traversal ignores the focus change. This is needed so that keyboard actions (for example, osfSelect or osfHelp) may be taken during the drag.

menuChildWidget
Specifies the menu child widget.

The getPopupPosted Trait Method

Widget getPopupPosted(
Widget *rowColumnWidget***);**

Each *rowColumnWidget* maintains one shell to encompass all of its Popup Menus. Use the **getPopupPosted** trait method to get the widget identifier of that shell.

menuChildWidget
Specifies the **XmRowColumn** widget managing your button widget.

This trait method returns the widget identifier of the shell managing the posted Popup Menu.

Related Information

XmRowColumn(3), **XmCascadeButton**(3), **XmPushButton**(3), **XmDrawnButton**(3), **XmArrowButton**(3), **XmPushButtonGadget**(3), **XmArrowButtonGadget**(3) and **XmMenuPosition**(3).

XmQTnavigator

Purpose

A widget holding this trait can act as a navigator

Synopsis

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

typedef struct {
        int version;
        XmNavigatorMoveCBProc changeMoveCB;
        XmNavigatorSetValueProc setValue;
        XmNavigatorGetValueProc getValue;
} XmNavigatorTraitRec, *XmNavigatorTrait;

void (*XmNavigatorMoveCBProc)(
         Widget,
         XtCallbackProc,
         XtPointer,
         Boolean);
void (*XmNavigatorSetValueProc)(
         Widget,
         XmNavigatorData,
         Boolean);
void (*XmNavigatorGetValueProc)(
         Widget,
         XmNavigatorData);
```

Description

A child widget holding the *XmQTnavigator* trait can act as a navigator widget. A user can manipulate a navigator widget in order to make different parts of a scrollable widget visible.

In the standard Motif widget set, the **XmScrollBar** and **XmSpinBox** widgets hold the *XmQTnavigator* trait. In the Exm demonstration widget set, the **ExmPanner** widget holds this trait.

Widgets holding the *XmQTnavigator* trait can cooperate with widgets holding the *XmQTscrollFrame* trait.

Two of the trait methods, **setValue** and **getValue**, require an **XmNavigatorData** structure, which has the following definition:

```
typedef struct {
        Mask valueMask;
        Mask dimMask;
        XmTwoDIntRec *value;
        XmTwoDIntRec *minimum;
        XmTwoDIntRec *maximum;
        XmTwoDIntRec *slider_size;
        XmTwoDIntRec *increment;
        XmTwoDIntRec *page_increment;
} XmNavigatorDataRec, *XmNavigatorData;
```

where *XmTwoDIntRec* is a 2-field structure defined as follows:

```
typedef struct {
        int x;
        int y;
} XmTwoDIntRec, *XmTwoDInt;
```

Following is a detailed description of the fields of the **XmNavigatorData** structure:

valueMask Holds a bit vector mask. Each bit in the bit vector mask represents a different field in the **XmNavigatorData** structure. If a bit is set, then the data for that field is valid. If that bit is not set, then the data within that field is ignored. Motif provides the following constants, each representing one bit of the bit mask. The caller can add any combination of the following constants in order to form the appropriate *valueMask* value:

NavDimMask
Validates the *dimMask* field.

NavValue Validates the *value* field.

NavMinimum
Validates the *minimum* field.

NavMaximum
Validates the *maximum* field.

NavSliderSize
Validates the *slider_size* field.

NavIncrement
Validates the *increment* field.

NavPageIncrement
Validates the *page_increment* field.

NavAllValid
Validates all fields.

dimMask Holds a bit vector mask. Each bit in the bit vector mask represents a different dimension. Currently, the only two supported dimensions are *x* and *y*. Therefore, 1 bit of this mask represents the *x* dimension and another bit represents the *y* dimension. Motif provides the following constants, each representing 1 bit of the bit mask. The caller may add any combination of the following constants in order to form the appropriate *dimMask* value:

NavigDimensionX
Validates the *x* dimension.

NavigDimensionY
Validates the *y* dimension.

The value of the *dimMask* field affects the interpretation of the *value*, *minimum*, *maximum*, *slider_size*, *increment*, and *page_increment* fields. The value of *dimMask* affects these other fields even if **NavDimMask** is not set. In fact, the only time that **NavDimMask** has influence is when the caller is trying to change or read the value of *dimMask*.

value Holds a **TwoDIntRec** structure containing two values: one is the value of the navigator widget in the *x* dimension, and the other is the value in the *y* dimension. For example, suppose the *value* in the *x* dimension is 50, the *minimum* is 10, and the *maximum* is 90. In this case, the navigator widget will have a horizontal position halfway between the left and right sides of the widget.

minimum Holds a **TwoDIntRec** structure containing two values: one is the minimum value of the navigator widget in the *x* dimension and the other is the minimum value in the *y* dimension.

maximum Holds a **TwoDIntRec** structure containing two values: one is the maximum value of the navigator widget in the *x* dimension and the other is the maximum value in the *y* dimension.

slider_size Holds a **TwoDIntRec** structure containing two values: one is the slider size of the navigator widget in the *x* dimension and the other is the slider size in the *y* dimension. The *slider_size* value is not an absolute value; for example, it is not a size in pixels. The actual size of the slider is based on the ratio of the *slider_size* to the difference between the *minimum* and *maximum*. For example, suppose the *slider_size* in the *x* dimension is 20, the *minimum* is 10, and the *maximum* is 90. In this case, the difference between the *minimum* and the *maximum* is 80. Therefore, the slider will occupy 20/80 (or 25%) of the allocated widget space in the *x* dimension.

increment Holds a **TwoDIntRec** structure containing two values: one is the increment size of the navigator widget in the *x* dimension and the other is the increment size in the *y* dimension. The increment size is the amount by which the *value* increases or decreases when the user takes an action that moves the slider by one increment.

page_increment
Holds a **TwoDIntRec** structure containing two values: one is the page increment size of the navigator widget in the *x* dimension and the other is the page increment size in the *y* dimension. The page increment size is the amount by which the *value* increases or decreases when the user takes an action that moves the slider by one page increment.

The changeMoveCB Trait Method

void changeMoveCB(
    Widget *navigatorWidget***,**
    XtCallbackProc *moveCB***,**
    XtPointer *closure***,**
    Boolean *setUnset***);**

The **changeMoveCB** trait method is responsible for adding or removing the *moveCB* callback from the list of callbacks. A child widget holding the *XmQTnavigator* trait should provide a resource that holds the name of the move callback procedure. This callback will be activated whenever a user moves an indicator in the *navigatorWidget*.

Following is a sample implementation of this trait method. In this implementation, the *ExmNmoveCallback* resource holds the name of the move callback procedure.

```
NavigChangeMoveCB(
           Widget navigatorWidget,
           XtCallbackProc moveCB,
           XtPointer closure,
           Boolean setunset)
{
  if (setunset)
    XtAddCallback (navigatorWidget, ExmNmoveCallback, moveCB, closure);
  else
    XtRemoveCallback (navigatorWidget, ExmNmoveCallback, moveCB, closure);
}
```

navigatorWidget
Specifies the child widget that is to have its list of callbacks modified.

moveCB Specifies the callback procedure that is to be added or removed from the list of callbacks.

closure Specifies additional data to be passed to the callback.

setUnset Specifies a **Boolean** value. If this value is *True*, the **setValue** trait method adds *moveCB* to the list of callbacks. If this value is *False*, the *moveCB* is removed from the list of callbacks.

The setValue Trait Method

void setValue(
Widget *navigatorWidget*,
XmNavigatorData *navigatorData*,
Boolean *notify*);

The **setValue** trait method allows the caller to pass new navigator data to the *navigatorWidget*.

navigatorWidget
Specifies the navigator widget that provides this method.

navigatorData
Specifies an **XmNavigatorData** structure whose fields describe the characteristics of the navigator widget.

notify Specifies a **Boolean** value. If *True*, the caller wants **setValue** to activate the move callback procedure whenever there is a change in the *value* field of the *navigatorData* structure.

The getValue Trait Method

void getValue(
  Widget *navigatorWidget***,**
  XmNavigatorData *navigatorData***);**

The **getValue** trait method returns the current *navigatorData* held by the *navigatorWidget*. Not all of the fields in the returned *navigatorData* will hold valid data. The caller of **getValue** must analyze the returned *valueMask* and *dimMask* fields to determine which of the other fields hold valid data.

navigatorWidget
Specifies the navigator widget that provides this method.

navigatorData
Specifies an **XmNavigatorData** structure whose fields describe the characteristics of the navigator widget.

Related Information

XmQTscrollFrame(3), **XmScrollBar**(3), **XmSpinBox**(3), and **ExmPanner**(3).

XmQTscrollFrame

Purpose A widget holding this trait can handle one or more navigator widgets

Synopsis

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

typdef struct {
        int version;
        XmScrollFrameInitProc init;
        XmScrollFrameGetInfoProc getInfo;
        XmScrollFrameAddNavigatorProc addNavigator;
        XmScrollFrameRemoveNavigatorProc removeNavigator;
} XmSrollFrameTraitRec, *XmScrollFrameTrait;

void (*XmScrollFrameInitProc)(
         Widget,
         XtCallbackProc,
         Widget);
Boolean (*XmScrollFrameGetInfoProc)(
         Widget,
         Cardinal*,
         Widget**,
         Cardinal*);
void (*XmScrollFrameAddNavigatorProc)(
         Widget,
         Widget,
         Mask);
void (*XmScrollFrameRemoveNavigatorProc)(
         Widget,
         Widget);
```

Description

A widget holding the *XmQTscrollFrame* trait can handle one or more navigator widgets and use them to pan a scrollable object. The trait methods of *XmQTscrollFrame* are useful if you are writing you own navigator or scrollable widget.

The following standard Motif manager widgets hold this trait:

- **XmScrolledWindow**
- **XmNotebook**

In addition, the following standard Motif widgets access the trait methods of *XmQTscrollFrame*:

- **XmList**
- **XmText**

The init Trait Method

void init(
    Widget *scrollFrameWidget***,**
    XtCallbackProc *moveCB***,**
    Widget *scrollableWidget***);**

The **init** trait method initializes several internal data fields. One of these fields must hold the name the default move callback (*moveCB*) procedure that is to be associated with this widget.

scrollFrameWidget
Specifies the widget that holds the *XmQTscrollFrame* trait.

moveCB Specifies the default move callback procedure that is to be associated with this widget. Typically, this callback is going to be added to a list of navigator widgets maintained by the *scrollFrameWidget*.

scrollableWidget
Specifies client data to be passed to the *moveCB*. Typically, the client data will be the name of a scrollable widget.

The getInfo Trait Method

Boolean getInfo(
Widget *scrollFrameWidget***,**
Cardinal ****dimension***,**
Widget *****nav_list***,**
Cardinal ****num_nav_list***);**

The **getInfo** trait method returns information about the *scrollFrameWidget*. The caller may set the *dimension*, *nav_list*, or *num_nav_list* fields to *NULL* if the returned information would be of no interest.

scrollFrameWidget
Specifies the widget that is providing this method.

dimension Returns the dimension(s) that *scrollFrameWidget* is capable of operating upon.

nav_list Returns the current list of navigators associated with this *scrollFrameWidget*. This data is returned to internal storage, and should not be modified or freed by the caller.

num_nav_list
Returns the number of navigators returned in *nav_list*.

This method returns *True* if the **init** trait method was already called. This method returns *False* if **init** has not yet been called. If this method returns *False*, then the returned information in the other fields should be ignored.

The addNavigator Trait Method

void addNavigator(
Widget *scrollFrameWidget***,**
Widget *navigatorWidget***,**
Mask *dimMask***);**

The **addNavigator** trait method associates a *navigatorWidget* with a *scrollFrameWidget*. Since one *scrollFrameWidget* can support multiple navigator widgets, it is possible that this method will be called multiple times.

scrollFrameWidget
Specifies the scroll frame widget that is providing this method.

navigatorWidget
Specifies the navigator widget to be associated with the *scrollFrameWidget*.

dimMask Specifies the dimension(s) upon which the *navigatorWidget* is going to operate.

The removeNavigator Trait Method

void removeNavigator(
  Widget *scrollFrameWidget***,**
  Widget *navigatorWidget***);**

The **removeNavigator** trait method disassociates a *navigatorWidget* from a *scrollFrameWidget*. Since one *scrollFrameWidget* can support multiple *navigatorWidget*s, this method can be called multiple times.

It is the responsibility of whomever associated the *navigatorWidget* with the *scrollFrameWidget* to call **removeNavigator** prior to destroying it.

scrollFrameWidget
Specifies the scroll frame widget that is providing this method.

navigatorWidget
Specifies the navigator widget to be disassociated from the *scrollFrameWidget*.

Related Information

XmScrolledWindow(3), **XmNotebook**(3), and **XmQTnavigator**(3).

XmQTspecifyRenderTable

Purpose

A widget holding this trait can supply the names of its default render tables to any requestor

Synopsis

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

typedef struct {
        int version;
        XmSpecRenderGetTableProc getRenderTable;
} XmSpecRenderTraitRec, *XmSpecRenderTrait;

XmRenderTable (*XmSpecRenderGetTableProc)(
         Widget,
         XtEnum);
```

Description

A widget holding the *XmQTspecifyRenderTable* trait is responsible for providing a requestor with a suitable render table. Any manager widget holding this trait should provide the following three resources:

- A resource to hold the label render table
- A resource to hold the button render table
- A resource to hold the text render table

The **getRenderTable** trait method of *XmQTspecifyRenderTable* will use the values of the three resources as the basis for its returned information.

The *XmQTspecifyRenderTable* trait is somewhat unusual in that other widgets do not access its trait method through the usual trait mechanisms. Instead, other widgets access the **getRenderTable** trait method by calling **XmeGetDefaultRenderTable**.

The following standard Motif widgets hold this trait:

- **XmBulletinBoard** and all its subclasses

- *XmVendorShell* and all its subclasses
- **XmMenuShell**

The *ExmGrid* demonstration widget also installs this trait.

The getRenderTable Trait Method

XmRenderTable getRenderTable(
  Widget *widget***,**
  XtEnum *renderTableType***);**

The **getRenderTable** trait method returns a render table corresponding to the *renderTableType* of widget *widget*. This value (if non-NULL) is the internal value of the manager's render table, and should not be modified or freed.

For example, given a manager widget named **MyManagerWidget** that supports the correct render table resources, the following is a sample implementation of the trait method:

```
static XmRenderTable
GetRenderTable( Widget widget,
                XtEnum renderTableType)
{
XmMyManagerWidget mm = (XmMyManagerWidget) widget;
 switch(renderTableType) {
   case XmLABEL_RENDER_TABLE: return mm->my_manager.label_render_table;
   case XmBUTTON_RENDER_TABLE: return mm->my_manager.button_render_table;
   case XmTEXT_RENDER__RENDER_TABLE: return mm->my_manager.text_render_table;
 }
 return NULL;
}
```

widget Specifies the widget containing render table information.

renderTableType
Specifies the desired type of render table; the specified value must be one of the following: **XmLABEL_RENDER_TABLE**, **XmBUTTON_RENDER_TABLE**, and **XmTEXT_RENDER_TABLE**.

Related Information

XmBulletinBoard(3), **VendorShell**(3), **XmMenuShell**(3), and **XmeGetDefaultRenderTable**(3).

XmQTtakesDefault

Purpose

A button widget holding this trait can change its appearance to show that it is the default button

Synopsis

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

typedef struct {
        int version;
        XmTakesDefaultNotifyProc showAsDefault;
} XmTakesDefaultTraitRec, *XmTakesDefaultTrait;

void (*XmTakesDefaultNotifyProc)(
         Widget,
         XtEnum);
```

Description

You will use the *XmQTtakesDefault* trait if you are writing a PushButton-style primitive widget or a DialogBox-style manager widget.

A DialogBox widget displays several PushButton-style children. For example, a typical DialogBox widget might display three PushButton widgets: an OK button, a Cancel button, and a Help button. One of these PushButton-style children should be the default button. The default button is the PushButton-style child that is activated when the user presses Return anywhere in the widget. (The **parent_process** method of the DialogBox is responsible for detecting the activation event.)

A PushButton-style widget must be capable of altering its appearance to show that it is the default choice. Most PushButton-style widgets do this by highlighting their borders in a special way.

If you are writing a PushButton-style widget, then your widget should hold the *XmQTtakesDefault* trait. This trait announces to DialogBox widgets that the child is capable of changing its appearance to show that it is the default choice. Conversely, if you are writing a DialogBox widget, then your DialogBox widget should examine its button children for this trait.

The following standard Motif primitives hold this trait:

- **XmPushButton**
- **XmPushButtonGadget**

In addition, the **ExmCommandButton** demonstrates how to install this trait and how to define a **showAsDefault** trait method.

The following standard Motif managers examine their children widgets for this trait and call the **showAsDefault** trait method:

- **XmBulletinBoard** and all its subclasses

The showAsDefault Trait Method

void showAsDefault(
    Widget *childWidget***,**
    XtEnum *state***);**

Every PushButton-style widget holding the *XmQTtakesDefault* trait must provide a **showAsDefault** trait method. The DialogBox that manages the PushButton-style widgets will call the *XmQTtakesDefault* trait method, each time passing a different value of *state*. The typical sequence of calls from the DialogBox manager is as follows:

- The DialogBox calls the **showAsDefault** trait method of each child widget, specifying a *state* of **XmDEFAULT_READY**. This message tells the child to prepare to become the default button. The *CDE 2.1/Motif 2.1—Style Guide and Glossary* does not mandate a particular way of handling this message, so your widget may visually do as you please. One possible response would be to increase the widget’s internal margins. (Calling **XmDEFAULT_READY** helps the manager avoid future unnecessary geometry requests.)
- The DialogBox calls the **showAsDefault** trait method of the chosen default child, specifying a *state* of **XmDEFAULT_ON**. This message tells the child to become the default button. The child must change its visual appearance in some way to show the user that it is now the default button. For example, the *ExmCommandButton* demonstration widget increases its shadow thickness to show that it is the default. Another widget might display some sort of icon (perhaps an arrow) to symbolize that it is the default button.
- If the default child widget changes, then the DialogBox calls the **showAsDefault** trait method twice. The first call specifies a *state* of **XmDEFAULT_OFF** in

order to turn off one default button. The second call specifies a *state* of **XmDEFAULT_ON** in order to turn on the new default button. Upon receiving the **XmDEFAULT_OFF** message, the child widget must change its appearance to show that it is no longer the default.

XmDEFAULT_FORGET resets the visual appearance of a child widget so that it looks as it did prior to the **XmDEFAULT_READY** call.

childWidget Specifies the child widget holding this trait.

state Specifies one of the following states: **XmDEFAULT_READY**, **XmDEFAULT_ON**, **XmDEFAULT_OFF**, and **XmDEFAULT_FORGET**.

Related Information

XmBulletinBoard(3), **XmPushButton**(3), and **ExmCommandButton**(3).

XmQTtransfer

Purpose

A trait that implements data transfer to and from a widget

Synopsis

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

typedef struct {
        int version;
        XmConvertCallbackProc convertProc;
        XmDestinationCallbackProc destinationProc;
        XmDestinationCallbackProc destinationPreHookProc;
} XmTransferTraitRec, *XmTransferTrait;

void (*XmConvertCallbackProc)(
         Widget,
         XtPointer,
         XtPointer);
void (*XmDestinationCallbackProc)(
         Widget,
         XtPointer,
         XtPointer);
```

Description

The *XmQTtransfer* trait is installed on any widget that can be a source or destination for data transfer. The usual means of transferring data is the conversion of a selection. The selections supported by Motif are *PRIMARY*, *SECONDARY*, *CLIPBOARD*, and _MOTIF_DROP (for drag and drop operations). A widget can be the source or destination for zero or more of these selections.

If your widget is to be a source of data in a transfer, then your widget must supply a **convertProc** trait method. This trait method is responsible for converting data to a requested target. If your widget is to be a data transfer destination, then your widget must supply a **destinationProc** trait method and can optionally supply a **destinatinoPreHookProc** trait method. The **destinationProc** trait method is responsible for requesting target(s) from the source and for providing the name of a transfer procedure to handle the returned data. The **destinationPreHookProc** trait method is used to prepare data for the **destinationProc** method.

The standard Motif widgets **XmContainer**, **XmLabel**, **XmLabelGadget**, **XmList**, **XmScale**, **XmText**, and **XmTextField** all hold this trait. **XmLabel**, **XmLabelGadget**, **XmList**, and **XmScale** provide only **convertProc** methods. **XmContainer**, **XmText**, and **XmTextField** provide both **convertProc** and **destinationProc** methods.

(See the *ExmStringTransfer* demonstration widget in the **demos/widgets/Exm/lib** directory for an example usage of this trait.)

The convertProc Trait Method

void convertProc(
    Widget *widget***,**
    XtPointer *client_data***,**
    XtPointer *call_data***);**

The **convertProc** method is invoked when the widget is the source for a data transfer operation. This method converts a selection to a requested target. The *call_data* argument is a pointer to an **XmConvertCallbackStruct** structure. The method typically examines the *selection* and *target* members of this structure. It returns the converted data in the *value* member and returns an indicator of the conversion status in the *status* member.

When the *target* member of the structure is *TARGETS*, the **convertProc** method usually provides data by calling **XmeStandardTargets** and then adding any targets to which the **convertProc** method converts data. The method can call **XmeStandardConvert** to convert data to any of the standard targets.

The **convertProc** method is invoked after any **XmNconvertCallback** procedures are called. The callback procedures can perform all or part of the conversion themselves, refuse the conversion, or let the **convertProc** method handle the conversion. If the callback procedures perform the complete conversion or refuse the conversion, the **convertProc** method is not called. More specifically, the **convertProc** method is not called if, after all callback procedures have returned, the value of the *status* member of the **XmConvertCallbackStruct** is other than **XmCONVERT_DEFAULT** or **XmCONVERT_MERGE**.

widget	Specifies the widget that is asked to convert the data.
client_data	This argument is currently unused. The value is always *NULL*.
call_data	Specifies a pointer to an **XmConvertCallbackStruct** structure.

Following is a description of the **XmConvertCallbackStruct** structure:

typedef struct {
  int *reason***;**
  XEvent ***event*****;**
  Atom *selection***;**
  Atom *target***;**
  XtPointer *source_data***;**
  XtPointer *location_data***;**
  int *flag***;**
  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 conversion target.

source_data Contains information about the selection source. When the selection is _MOTIF_DROP, *source_data* is the DragContext. Otherwise, it 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.

flag This member 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 8-bit, 16-bit, or 32-bit quantities. Possible values are 0 (when *parm* is *NULL*), 8, 16, and 32.

parm_length Specifies the number of elements of data in *parm*, where each element has the number of bits 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 set by the toolkit is **XmCONVERT_DEFAULT**, but an **XmNconvertCallback** procedure may have set a different value. Following are the possible values:

XmCONVERT_DEFAULT
This value means that the **convertProc** trait method, if it wishes, should perform the entire conversion. If the **convertProc** produces any data, it overwrites the data provided by the callback procedures in the *value* member.

XmCONVERT_MERGE
This value means that the **convertProc** trait method should merge its converted data, if any, with the data provided by the callback procedures. If the **convertProc** produces any data, it uses **XmeConvertMerge** to append 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
The *status* member never has this value on entry to **convertProc**. When the **convertProc** trait method finishes a successful conversion, it sets *status* to **XmCONVERT_DONE**.

XmCONVERT_REFUSE
The *status* member never has this value on entry to the **convertProc**. When the **convertProc** trait method terminates an unsuccessful conversion, it sets *status* to **XmCONVERT_REFUSE**.

Before returning, the **convertProc** trait method should always set *status* to either **XmCONVERT_DONE**, indicating a successful conversion, or **XmCONVERT_REFUSE**, indicating an unsuccessful conversion.

value An **IN/OUT** parameter that contains any data that the callback procedures or the **convertProc** trait method produces as a result of the conversion. The initial value is *NULL*. If the **convertProc** trait method sets this member, it must ensure that the *type*, *format*, and *length* members correspond to the data in *value*. The **convertProc** trait method 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 8-bit, 16-bit, or 32-bit quantities. The initial value is **8**. The **convertProc** trait method can set this member to **8**, **16**, or **32**.

length An **IN/OUT** member that specifies the number of elements of data in *value*, where each element has the number of bits specified by *format*. The initial value is 0.

The destinationProc Trait Method

void destinationProc(
Widget *widget*,
XtPointer *client_data*,
XtPointer *call_data*);

The **destinationProc** trait method is invoked when the widget is the destination for a data transfer operation. The *call_data* argument is a pointer to an **XmDestinationCallbackStruct** structure. This method requests conversion of a selection to an appropriate target, usually by calling *XmTransferValue*. The method usually supplies a callback to *XmTransferValue* that receives and possibly inserts the converted data. The *XmTransferValue* callback returns an indicator of the transfer status. Either the **destinationProc** trait method or the *XmTransferValue* callback can also terminate the transfer operation by calling *XmTransferDone*.

The **destinationProc** trait method is invoked after any **XmNdestinationCallback** procedures are called and after all data conversions initiated by those callback procedures are complete. The callback procedures can perform the data transfer themselves, refuse the transfer, or let the **destinationProc** trait method handle the transfer. If the callback procedures perform the transfer themselves or refuse the transfer, the **destinationProc** trait method is not called. More specifically, the **destinationProc** trait method is not invoked if a callback procedure has called *XmTransferDone* with a transfer status other than **XmTRANSFER_DEFAULT**.

widget Specifies the widget that is the destination for the transfer.

client_data This argument is currently unused. The value is always *NULL*.

call_data Specifies a pointer to an **XmDestinationCallbackStruct** structure.

Following is a description of the **XmDestinationCallbackStruct** structure:

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 *CLIPBOARD*, *PRIMARY*, *SECONDARY*, and _MOTIF_DROP.

operation Indicates the type of transfer operation requested.

- When the selection is *PRIMARY*, possible values are **XmMOVE**, **XmCOPY**, and **XmLINK**.
- When the selection is *SECONDARY* or *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 **destinationProc** trait method is 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 recommmended 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 value is always *NULL* when the selection is *SECONDARY* or *CLIPBOARD*. If the value is *NULL*, the data is to be inserted at the widget's cursor position. Otherwise, the type and interpretation of the value are specific to the widget class. The value of *location_data* is only valid for the duration of a transfer. Once transfer done procedures start to be called, *location_data* will no longer be stable.

time Indicates the time when the transfer operation began.

The destinationPreHookProc Trait Method

void destinationPreHookProc(
    Widget *widget***,**
    XtPointer *client_data***,**
    XtPointer *call_data***);**

The **destinationPreHookProc** trait method is invoked when the widget is the destination for a data transfer operation. This method is called before any **XmNdestinationCallback** procedures are invoked. The *call_data* argument is a pointer to an **XmDestinationCallbackStruct** structure. The method can provide any information, such as a value for the *location_data* member of the structure, that the **XmNdestinationCallback** procedures or the **destinationProc** trait method may need.

widget	Specifies the widget that is the destination for the transfer.
client_data	This argument is currently unused. The value is always *NULL*.
call_data	Specifies a pointer to an **XmDestinationCallbackStruct** structure.

Related Information

ExmStringTransfer(3), **XmTransferDone**(3), **XmTransferValue**(3), **XmeClipboardSink**(3), **XmeClipboardSource**(3), **XmeConvertMerge**(3), **XmeDragSource**(3), **XmeDropSink**(3), **XmeGetEncodingAtom**(3), **XmePrimarySink**(3), **XmePrimarySource**(3), **XmeSecondarySink**(3), **XmeSecondarySource**(3), **XmeSecondaryTransfer**(3), **XmeStandardConvert**(3), **XmeStandardTargets**(3), and **XmeTransferAddDoneProc**(3).

Chapter 18

Exm Demonstration Widgets Reference Pages

The Exm widget set contains sample C and C++ code for several widgets. In this chapter, we define the behavior of each of these widgets. Feel free to experiment with the code from the Exm widget set, but DO NOT USE THESE WIDGETS IN A REAL APPLICATION.

ExmCommandButton

Purpose

The ExmCommandButton widget class

Synopsis

#include <Exm/CommandB.h>

Description

ExmCommandButton is a demonstration widget. OSF provides this widget solely to teach programmers how to write their own Motif widgets. OSF does not support this widget in any way.

The source code for the **ExmCommandButton** widget illustrates how to do the following:

- Create a Motif button widget. Such a button widget could serve, for example, as an OK button in a Dialog box.
- Establish Motif-style actions for buttons.
- Install the *XmQTactivatable* trait.
- Install the *XmQTtakesDefault* trait.
- Establish a simple Motif activate callback.

ExmCommandButton is an instantiable widget.

ExmCommandButton is a subclass of the **ExmString** widget and is also used as a superclass for other widgets, such as *ExmTabButton*. Therefore, **ExmCommandButton** inherits all of **ExmString**’s resources for rendering compound strings. **ExmCommandButton** provides many action methods, but **ExmString** does not.

When a user activates an **ExmCommandButton** widget, the widget calls the callback procedure associated with the **XmNactivateCallback** resource.

Classes

CommandButton inherits behavior and resources from **Core**, **XmPrimitive**, *ExmSimple*, and **ExmString**.

The class pointer is **exmCommandButtonWidgetClass**.

The class name is **ExmCommandButton**.

New Resources

The following table defines a set of widget resources used by the programmer to specify data. The programmer can also set the resource values for the inherited classes to set attributes for this widget. To reference a resource by name or by class in a **.Xdefaults** file, remove the **XmN**, *ExmN*, **XmC** or *ExmC* prefix and use the remaining letters. To specify one of the defined values for a resource in a **.Xdefaults** file, remove the **Xm** or **Exm** prefix and use the remaining letters (in either lowercase or uppercase, but include any underscores between words). The codes in the "Access" column indicate if the given resource can be set at creation time (C), set by using **XtSetValues** (S), retrieved by using **XtGetValues** (G), or is not applicable (N/A).

ExmCommandButton Resource Set				
Name	**Class**	**Type**	**Default**	**Access**
XmNactivateCallback	**XmCCallback**	**XtCallbackList**	*NULL*	C

XmNactivateCallback
Specifies the list of callbacks that is called when **ExmCommandButton** is activated. The type of the structure whose address is passed to this callback is **XmAnyCallbackStruct**. The reason is **XmCR_ACTIVATE**.

Inherited Resources

CommandButton inherits behavior and resources from the following superclasses. For a complete description of each resource, refer to the reference page for that superclass.

ExmString Resource Set				
Name	**Class**	**Type**	**Default**	**Access**
ExmNcompoundString	**ExmCCompoundString**	**XmString**	*NULL*	CSG
XmNalignment	**XmCAlignment**	**unsigned char**	*XmALIGNMENT_-CENTER*	CSG

XmNrecomputeSize	**XmCRecomputeSize**	**Boolean**	*true*	CSG
XmNrenderTable	**XmCRenderTable**	**XmRenderTable**	*dynamic*	CSG

ExmSimple Resource Set				
Name	**Class**	**Type**	**Default**	**Access**
ExmNsimpleShape	**ExmCSimpleShape**	**unsigned char**	*ExmSHAPE_OVAL*	CSG
XmNmarginHeight	**XmCMarginHeight**	**Dimension**	4	CSG
XmNmarginWidth	**XmCMarginWidth**	**Dimension**	4	CSG

XmPrimitive Resource Set				
Name	**Class**	**Type**	**Default**	**Access**
XmNbottom-ShadowColor	**XmCBottom-ShadowColor**	**Pixel**	*dynamic*	CSG
XmNbottom-ShadowPixmap	**XmCBottom-ShadowPixmap**	**Pixmap**	XmUNSPECIFIED_-PIXMAP	CSG
XmNconvert- Callback	**XmCCallback**	**XtCallback- List**	*NULL*	C
XmNforeground	**XmCForeground**	**Pixel**	*dynamic*	CSG
XmNhelpCallback	**XmCCallback**	**XtCallback- List**	*NULL*	C
XmNhighlightColor	**XmCHighlight-Color**	**Pixel**	*dynamic*	CSG
XmNhighlightOn- Enter	**XmCHighlight-OnEnter**	**Boolean**	*false*	CSG
XmNhighlightPixmap	**XmCHighlight-Pixmap**	**Pixmap**	*dynamic*	CSG
XmNhighlight- Thickness	**XmCHighlight-Thickness**	**Dimension**	2	CSG
XmNlayoutDirection	**XmCLayout-Direction**	**XmDirection**	*dynamic*	CG
XmNnavigationType	**XmCNavigation-Type**	**XmNavigation-Type**	XmNONE	CSG
XmNpopup-HandlerCallback	**XmCCallback**	**XtCallback- List**	*NULL*	C
XmNshadowThickness	**XmCShadow-Thickness**	**Dimension**	3	CSG

XmNtopShadowColor	**XmCTop-ShadowColor**	**Pixel**	*dynamic*	CSG
XmNtop- ShadowPixmap	**XmCTop-ShadowPixmap**	**Pixmap**	*dynamic*	CSG
XmNtraversalOn	**XmCTraversalOn**	**Boolean**	*true*	CSG
XmNunitType	**XmCUnitType**	**unsigned char**	*dynamic*	CSG
XmNuserData	**XmCUserData**	**XtPointer**	*NULL*	CSG

Core Resource Set				
Name	**Class**	**Type**	**Default**	**Access**
XmNaccelerators	**XmCAccelerators**	**XtAccelerators**	*dynamic*	CSG
XmNancestor-Sensitive	**XmCSensitive**	**Boolean**	*dynamic*	G
XmNbackground	**XmCBackground**	**Pixel**	*dynamic*	CSG
XmNbackground-Pixmap	**XmCPixmap**	**Pixmap**	XmUNSPECIFIED_-PIXMAP	CSG
XmNborderColor	**XmCBorderColor**	**Pixel**	XtDefault- Foreground	CSG
XmNborderPixmap	**XmCPixmap**	**Pixmap**	XmUNSPECIFIED_-PIXMAP	CSG
XmNborderWidth	**XmCBorderWidth**	**Dimension**	0	CSG
XmNcolormap	**XmCColormap**	**Colormap**	*dynamic*	CG
XmNdepth	**XmCDepth**	**int**	*dynamic*	CG
XmNdestroy- Callback	**XmCCallback**	**XtCallbackList**	*NULL*	C
XmNheight	**XmCHeight**	**Dimension**	*dynamic*	CSG
XmNinitialResources-Persistent	**XmCInitial-ResourcesPersistent**	**Boolean**	*true*	C
XmNmapped-WhenManaged	**XmCMapped-WhenManaged**	**Boolean**	*true*	CSG
XmNscreen	**XmCScreen**	**Screen ***	*dynamic*	CG
XmNsensitive	**XmCSensitive**	**Boolean**	*true*	CSG
XmNtranslations	**XmCTranslations**	**XtTranslations**	*dynamic*	CSG
XmNwidth	**XmCWidth**	**Dimension**	*dynamic*	CSG

XmNx	**XmCPosition**	**Position**	0	CSG
XmNy	**XmCPosition**	**Position**	0	CSG

Callback Information

A pointer to the following structure is passed to each callback:

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

reason	Indicates why the callback was invoked.
event	Points to the *XEvent* that triggered the callback.

Translations

ExmCommandButton provides the following translations:

<EnterWindow>:
    ExmCommandButtonEnter()

<LeaveWindow>:
    ExmCommandButtonLeave()

~c <Btn1Down>:
    ExmCommandButtonArm()

<Btn1Up>:
    ExmCommandButtonActivate() ExmCommandButtonDisarm()

~s ~m ~a <Key>space:
    ExmCommandButtonArmAndActivate()

~s ~m ~a <Key>Return:
    PrimitiveParentActivate()

In addition to the preceding translations, **ExmCommandButton** also inherits all the traversal translations of **XmPrimitive**. For details on these translations, see the **XmPrimitive**(3) reference page.

Action Routines

The **ExmCommandButton** action routines are as follows:

ExmCommandButtonActivate()
: If the pointer is within the widget, calls the callback list of **XmNactivateCallback**.

ExmCommandButtonArm()
: Arms the widget. Renders the border in the selected state and the shadow in the armed state.

ExmCommandButtonArmAndActivate()
: Draws the shadow in the armed state. Calls the callbacks named by **XmNactivateCallback**. Upon completion, draws the shadow in the disarmed state.

ExmCommandButtonDisarm()
: If the widget is armed, this action disarms it. Draws the shadow in the disarmed state.

ExmCommandButtonEnter()
: If the widget is already armed, draws the shadow in the armed state. Otherwise, this action does nothing.

ExmCommandButtonLeave()
: If the widget is already armed, draws the shadow in the disarmed state. Otherwise, this action does nothing.

Virtual Bindings

For information about bindings for virtual buttons and keys, see **VirtualBindings**(3).

Related Information

Core(3), **ExmSimple**(3), **ExmString**(3), and **XmPrimitive**(3).

ExmGrid

Purpose

The Grid widget class

Synopsis

#include <Exm/Grid.h>

Description

ExmGrid is a demonstration widget. OSF provides this widget solely to teach programmers how to write their own Motif widgets. OSF does not support this widget in any way.

The source code for the **ExmGrid** widget illustrates how to do the following:

- Write a Motif-style manager widget supporting Motif-style resources and constraints
- Install the *XmQTdialogShellSavvy* trait and its methods
- Install the *XmQTspecifyRenderTable* trait and its methods

ExmGrid is a general-purpose manager capable of containing any widget type as a child. In general, it requires no special knowledge about how its children function.

ExmGrid is like a light-weight version of the standard Motif toolkit **XmRowColumn** widget. (**XmRowColumn** has an order of magnitude more code than **ExmGrid**.) Like **XmRowColumn**, **ExmGrid** lays out its children widgets within the cells of a two-dimensional matrix. Unlike **XmRowColumn**, **ExmGrid** cannot manage menus.

ExmGrid ignores the value of the **XmNshadowThickness** resource of **XmManager**.

Classes

ExmGrid inherits behavior and resources from **Core**, **Composite**, **Constraint** and **XmManager**.

The class pointer is **exmGridWidgetClass**.

The class name is **ExmGrid**.

New Resources

The following table defines a set of widget resources used by the programmer to specify data. The programmer can also set the resource values for the inherited classes to set attributes for this widget. To reference a resource by name or by class in a **.Xdefaults** file, remove the **XmN**, *ExmN*, **XmC** or *ExmC* prefix and use the remaining letters. To specify one of the defined values for a resource in a **.Xdefaults** file, remove the **Xm** or **Exm** prefix and use the remaining letters (in either lowercase or uppercase, but include any underscores between words). The codes in the "Access" column indicate if the given resource can be set at creation time (C), set by using **XtSetValues** (S), retrieved by using **XtGetValues** (G), or is not applicable (N/A).

ExmGrid Resource Set

Name	Class	Type	Default	Access
XmNbutton- RenderTable	**XmCButton- RenderTable**	**XmRender- Table**	*NULL*	CSG
XmNcolumns	**XmCColumns**	**short**	4	CSG
XmNdefaultPosition	**XmCDefault- Position**	**Boolean**	*true*	CSG
XmNdialogTitle	**XmCDialogTitle**	**XmString**	*NULL*	CSG
XmNlabel- RenderTable	**XmCLabel- RenderTable**	**XmRender- Table**	*NULL*	CSG
XmNmapCallback	**XmCCallback**	**XtCallbackList**	*NULL*	C
XmNmarginHeight	**XmCMargin- Height**	**Dimension**	10	CSG
XmNmarginWidth	**XmCMargin- Width**	**Dimension**	10	CSG
XmNrows	**XmCRows**	**short**	4	CSG
XmNtext- RenderTable	**XmCText- RenderTable**	**XmRenderTable**	*NULL*	CSG
XmNunmapCallback	**XmCCallback**	**XtCallbackList**	*NULL*	C

XmNbuttonRenderTable

Specifies the default button render table used by Grid's button descendants (for example, **ExmCommandButton** or **XmPushButton**). If a button descendant has not defined a render table, Motif searches the button descendants parent hierarchy for the first ancestor that has the *XmQTspecifyRenderTable* trait installed. (For example, **VendorShell**, **MenuShell**, **ExmGrid**, **XmBulletinBoard** and its subclasses all install the *XmQTspecifyRenderTable* trait.) If the first ancestor encountered is an **ExmGrid** widget, then the value of **XmNbuttonRenderTable** becomes the button descendant's render table. If **XmNbuttonRenderTable** of **ExmGrid** is *NULL*, then Motif continues searching ancestors until it finds one that defines a button render table. Refer to **XmRenderTable**(3) for more information on the creation and structure of a render table.

XmNcolumns

Specifies the number of columns in the Grid. The default is 4 columns.

XmNdefaultPosition

Controls the positioning of a DialogShell managing a Grid. This resource has no influence if the Grid is not managed by a DialogShell. If **XmNdefaultPosition** is *true*, the DialogShell will center itself at the center of its own parent. For example, if the parent of the DialogShell is an ApplicationShell, then the center of the DialogShell will be at the same coordinates as the center of the ApplicationShell. If the DialogShell becomes unmapped (but stays managed) and then remapped, this resource has no influence on the DialogShell's position. If this resource is *False*, the DialogShell does not automatically center itself. Instead, the DialogShell (and therefore the Grid) will be positioned according to the values of **XmNx** and **XmNy**. Motif will treat the values of **XmNx** and **XmNy** as offsets from the upper-left corner of the screen (rather than as offsets from the upper-left corner of the parent shell).

XmNdialogTitle

Specifies the dialog title. If this resource is not *NULL*, and the parent of the Grid is a subclass of WMShell, Grid sets the **XmNtitle** and **XmNtitleEncoding** of its parent. If the only character set in **XmNdialogTitle** is ISO8859-1, **XmNtitle** is set to the string of the title, and **XmNtitleEncoding** is set to *STRING*. If **XmNdialogTitle** contains character sets other than ISO8859-1, **XmNtitle** is set to the string of the title converted to a compound text string, and **XmNtitleEncoding** is set to *COMPOUND_TEXT*. The direction of the title is based on the **XmNlayoutDirection** resource of the widget.

XmNlabelRenderTable

Specifies the default button render table used by Grid's label descendants (for example, **ExmString** or **XmLabel**). If a label descendant has not defined a render table, Motif searches the label descendants parent hierarchy for the first ancestor that has the *XmQTspecifyRenderTable* trait installed. (For example, **VendorShell**,

MenuShell, **ExmGrid**, **XmBulletinBoard** and its subclasses all install the *XmQTspecifyRenderTable* trait.) If the first ancestor encountered is an **ExmGrid** widget, then the value of **XmNlabelRenderTable** becomes the label descendant's render table. If **XmNlabelRenderTable** of **ExmGrid** is *NULL*, then Motif continues searching ancestors until it finds one that defines a label render table. Refer to **XmRenderTable**(3) for more information on the creation and structure of a render table.

XmNmapCallback

Specifies the list of callbacks that is called only when the parent of the Grid is a DialogShell. If the parent of the Grid is not a DialogShell, **XmNmapCallback** has no influence. Assuming that the parent of the Grid is a DialogShell, the callback list is invoked when the DialogShell (and therefore the Grid) is mapped. The callback reason is **XmCR_MAP**. DialogShells are usually mapped when the Grid is managed.

XmNmarginHeight

Specifies the amount of blank space between the top edge of the Grid widget and the first item in each column, and between the bottom edge of the Grid widget and the last item in each column. The default value is 10 pixels.

XmNmarginWidth

Specifies the amount of blank space between the left edge of the Grid widget and the first item in each row, and between the right edge of the Grid widget and the last item in each row. The default value is 10 pixels.

XmNrows Specifies the number of rows in the Grid. The default is 4 rows.

XmNtextRenderTable

Specifies the default button render table used by Grid's text descendants (for example, **XmText**, **XmTextField**, and **XmList**). If a text descendant has not defined a render table, Motif searches the text descendants parent hierarchy for the first ancestor that has the *XmQTspecifyRenderTable* trait installed. (For example, **VendorShell**, **MenuShell**, **ExmGrid**, **XmBulletinBoard** and its subclasses all install the *XmQTspecifyRenderTable* trait.) If the first ancestor encountered is an **ExmGrid** widget, then the value of **XmNtextRenderTable** becomes the text descendant's render table. If **XmNtextRenderTable** of **ExmGrid** is *NULL*, then Motif continues searching ancestors until it finds one that defines a text render table. Refer to **XmRenderTable**(3) for more information on the creation and structure of a render table.

XmNunmapCallback
Specifies the list of callbacks that is called only when the parent of the Grid is a DialogShell. In this case, this callback list is invoked when the Grid is unmapped. The callback reason is **XmCR_UNMAP**. DialogShells are usually unmapped when the Grid is unmanaged.

ExmGrid Constraint Resource Set

Name	Class	Type	Default	Access
ExmNgridMargin-HeightWithinCell	**ExmCGridMargin-HeightWithinCell**	**Dimension**	0	CSG
ExmNgridMargin-WidthWithinCell	**ExmCGridMargin-WidthWithinCell**	**Dimension**	0	CSG

ExmNgridMarginHeightWithinCell
Specifies the amount of blank space between the top edge of a child widget and the top of its cell. Also specifies the amount of blank space between the bottom edge of a child widget and the bottom of its cell.

ExmNgridMarginWidthWithinCell
Specifies the amount of blank space between the left edge of a child widget and the left edge of its cell. Also specifies the amount of blank space between the right edge of a child widget and the right edge of its cell.

Inherited Resources

ExmGrid inherits behavior and resources from the following superclasses. For a complete description of each resource, refer to the reference page for that superclass.

XmManager Resource Set

Name	Class	Type	Default	Access
XmNbottom-ShadowColor	**XmCBottom-ShadowColor**	**Pixel**	*dynamic*	CSG
XmNbottom-ShadowPixmap	**XmCBottom-ShadowPixmap**	**Pixmap**	XmUNSPECIFIED_-PIXMAP	CSG
XmNforeground	**XmCForeground**	**Pixel**	*dynamic*	CSG

XmNhelpCallback	**XmCCallback**	**XtCallbackList**	*NULL*	C
XmNhighlightColor	**XmCHighlightColor**	**Pixel**	*dynamic*	CSG
XmNhighlightPixmap	**XmCHighlightPixmap**	**Pixmap**	*dynamic*	CSG
XmNinitialFocus	**XmCInitialFocus**	**Widget**	*NULL*	CSG
XmNlayoutDirection	**XmCLayoutDirection**	**XmDirection**	*dynamic*	CG
XmNnavigationType	**XmCNavigationType**	**XmNavigation-Type**	*XmTAB_GROUP*	CSG
XmNpopup-HandlerCallback	**XmCCallback**	**XtCallbackList**	*NULL*	C
XmNshadow-Thickness	**XmCShadow-Thickness**	**Dimension**	0	CSG
XmNstring- Direction	**XmCString- Direction**	**XmString-Direction**	*dynamic*	CG
XmNtopShadow-Color	**XmCTop-ShadowColor**	**Pixel**	*dynamic*	CSG
XmNtop-ShadowPixmap	**XmCTop-ShadowPixmap**	**Pixmap**	*dynamic*	CSG
XmNtraversalOn	**XmCTraversalOn**	**Boolean**	*true*	CSG
XmNunitType	**XmCUnitType**	**unsigned char**	*dynamic*	CSG
XmNuserData	**XmCUserData**	**XtPointer**	*NULL*	CSG

Composite Resource Set				
Name	**Class**	**Type**	**Default**	**Access**
XmNchildren	**XmCReadOnly**	**WidgetList**	*NULL*	G
XmNinsertPosition	**XmCInsertPosition**	**XtOrderProc**	*NULL*	CSG
XmNnumChildren	**XmCReadOnly**	**Cardinal**	0	G

Core Resource Set				
Name	**Class**	**Type**	**Default**	**Access**
XmNaccelerators	**XmCAccelerators**	**XtAccelerators**	*dynamic*	CSG
XmNancestor-Sensitive	**XmCSensitive**	**Boolean**	*dynamic*	G
XmNbackground	**XmCBackground**	**Pixel**	*dynamic*	CSG

XmNbackgroundPixmap	**XmCPixmap**	**Pixmap**	*XmUNSPECIFIED_PIXMAP*	CSG
XmNborderColor	**XmCBorderColor**	**Pixel**	*XtDefaultForeground*	CSG
XmNborderPixmap	**XmCPixmap**	**Pixmap**	*XmUNSPECIFIED_PIXMAP*	CSG
XmNborderWidth	**XmCBorderWidth**	**Dimension**	0	CSG
XmNcolormap	**XmCColormap**	**Colormap**	*dynamic*	CG
XmNdepth	**XmCDepth**	**int**	*dynamic*	CG
XmNdestroyCallback	**XmCCallback**	**XtCallbackList**	*NULL*	C
XmNheight	**XmCHeight**	**Dimension**	*dynamic*	CSG
XmNinitialResourcesPersistent	**XmCInitialResourcesPersistent**	**Boolean**	*true*	C
XmNmappedWhenManaged	**XmCMappedWhenManaged**	**Boolean**	*true*	CSG
XmNscreen	**XmCScreen**	**Screen ***	*dynamic*	CG
XmNsensitive	**XmCSensitive**	**Boolean**	*true*	CSG
XmNtranslations	**XmCTranslations**	**XtTranslations**	*dynamic*	CSG
XmNwidth	**XmCWidth**	**Dimension**	*dynamic*	CSG
XmNx	**XmCPosition**	**Position**	0	CSG
XmNy	**XmCPosition**	**Position**	0	CSG

Translations

ExmGrid inherits all the translations of **XmManager**. **ExmGrid** does not provide any additional translations beyond those defined by **XmManager**.

Action Routines

ExmGrid provides no action routines of its own.

Related Information

Core(3), **Composite**(3), **Constraint**(3), and **XmManager**(3).

ExmMenuButton

Purpose

The MenuButton widget class demonstration widget

Synopsis

#include <Exm/MenuB.h>

Description

ExmMenuButton is a demonstration widget. OSF provides this widget solely to teach programmers how to write their own Motif widgets. OSF does not support this widget in any way.

The source code for the **ExmMenuButton** widget illustrates how to do the following:

- Create a menu child widget
- Install the *XmQTmenuSavvy* trait
- Call some of the trait methods of *XmQTmenuSystem*

ExmMenuButton is similar to the standard Motif toolkit widget **XmPushButton**. The primary difference is that **XmPushButton** supports a richer set of actions and callbacks than **ExmMenuButton**. **ExmMenuButton** illustrates only the essentials of creating a menu button.

ExmMenuButton is an instantiable widget. **ExmMenuButton** expects **XmRowColumn** to be its parent.

ExmMenuButton is a subclass of the **ExmString** widget. Therefore, **ExmMenuButton** inherits all of **ExmString**'s resources for rendering compound strings.

When a user activates an **ExmMenuButton** widget, the widget calls the callback procedure associated with the **XmNactivateCallback** resource.

Classes

ExmMenuButton inherits behavior and resources from **Core**, **XmPrimitive**, **ExmSimple**, and **ExmString**.

The class pointer is **exmMenuButtonWidgetClass**.

The class name is **ExmMenuButton**.

New Resources

The following table defines a set of widget resources used by the programmer to specify data. The programmer can also set the resource values for the inherited classes to set attributes for this widget. To reference a resource by name or by class in a **.Xdefaults** file, remove the **XmN**, **ExmN**, **XmC** or **ExmC** prefix and use the remaining letters. To specify one of the defined values for a resource in a **.Xdefaults** file, remove the **Xm** or **Exm** prefix and use the remaining letters (in either lowercase or uppercase, but include any underscores between words). The codes in the "Access" column indicate if the given resource can be set at creation time (C), set by using **XtSetValues** (S), retrieved by using **XtGetValues** (G), or is not applicable (N/A).

ExmMenuButton Resource Set				
Name	**Class**	**Type**	**Default**	**Access**
XmNaccelerator	**XmCAccelerator**	**String**	*NULL*	CSG
XmNacclleratorText	**XmCAccelerator- Text**	**XmString**	*NULL*	CSG
XmNactivateCallback	**XmCCallback**	**XtCallbackList**		CSG
XmNmnemonic	**XmCMnemonic**	**KeySym**	**XK_VoidSymbol**	CSG
XmNmnemonic-CharSet	**XmCMnemonic-CharSet**	**XmStringCharSet**	**XmFONTLIST_-DEFAULT_TAG**	CSG

XmNaccelerator

Sets the accelerator on a MenuButton widget in a menu, which activates a visible or invisible, but managed, button from the keyboard. This resource is a string that describes a set of modifiers and the key that may be used to select the button. The format of this string is identical to that used by the translations manager, with the exception that only a single event may be specified and only **KeyPress** events are allowed.

Accelerators for MenuButtons are supported only in Pulldown and Popup MenuPanes.

XmNacceleratorText

Specifies the text displayed for the accelerator. The text is displayed adjacent to the label string or pixmap. The direction of its layout depends on the **XmNlayoutDirection** resource of the widget. Accelerator text for buttons is displayed only for MenuButtons in Pulldown and Popup Menus.

XmNactivateCallback

Specifies the list of callbacks that is called when MenuButton is activated. MenuButton is activated when the user presses and releases the active mouse button while the pointer is inside that widget. Activating the MenuButton also disarms it. For this callback, the reason is **XmCR_ACTIVATE**.

XmNmnemonic

Provides the user with an alternate means of activating a button. A MenuButton in a MenuBar, a Popup MenuPane, or a Pulldown MenuPane can have a mnemonic.

This resource contains a literal as listed in the X11 literal table. The first character in the label string that exactly matches the mnemonic in the character set specified in **XmNmnemonicCharSet** is underlined when the button is displayed.

When a mnemonic has been specified, the user activates the button by pressing the mnemonic key while the button is visible. The user can activate the button by pressing either the shifted or the unshifted mnemonic key.

XmNmnemonicCharSet

Specifies the character set of the mnemonic for the label. The default is **XmFONTLIST_DEFAULT_TAG**.

Inherited Resources

MenuButton inherits behavior and resources from the following superclasses. For a complete description of each resource, refer to the reference page for that superclass.

ExmString Resource Set				
Name	**Class**	**Type**	**Default**	**Access**
ExmNcompound-String	**ExmCCompound-String**	**XmString**	*NULL*	CSG
XmNalignment	**XmCAlignment**	**unsigned char**	**XmALIGNMENT_-BEGINNING**	CSG
XmNrecomputeSize	**XmCRecompute- Size**	**Boolean**	*True*	CSG
XmNrenderTable	**XmCRenderTable**	**XmRenderTable**	*dynamic*	CSG

ExmSimple Resource Set				
Name	**Class**	**Type**	**Default**	**Access**
ExmNsimpleShape	**ExmCSimpleShape**	**unsigned char**	**ExmSHAPE_OVAL**	CSG
XmNmarginHeight	**XmCMarginHeight**	**Dimension**	4	CSG
XmNmarginWidth	**XmCMarginWidth**	**Dimension**	4	CSG

XmPrimitive Resource Set				
Name	**Class**	**Type**	**Default**	**Access**
XmNbottom-ShadowColor	**XmCBottom-ShadowColor**	**Pixel**	*dynamic*	CSG
XmNbottom-ShadowPixmap	**XmCBottom-ShadowPixmap**	**Pixmap**	**XmUNSPECIFIED_-PIXMAP**	CSG
XmNconvertCallback	**XmCCallback**	**XtCallbackList**	*NULL*	C
XmNforeground	**XmCForeground**	**Pixel**	*dynamic*	CSG
XmNhelpCallback	**XmCCallback**	**XtCallbackList**	*NULL*	C
XmNhighlightColor	**XmCHighlightColor**	**Pixel**	*dynamic*	CSG
XmNhighlightOn-Enter	**XmCHighlight-OnEnter**	**Boolean**	*False*	CSG
XmNhighlightPixmap	**XmCHighlight-Pixmap**	**Pixmap**	*dynamic*	CSG
XmNhighlight-Thickness	**XmCHighlight-Thickness**	**Dimension**	0	CSG
XmNlayoutDirection	**XmCLayout- Direction**	**XmDirection**	*dynamic*	CG
XmNnavigationType	**XmCNavigation- Type**	**XmNavigationType**	**XmNONE**	CSG
XmNpopupHandler-Callback	**XmCCallback**	**XtCallbackList**	*NULL*	C
XmNshadow-Thickness	**XmCShadow-Thickness**	**Dimension**	2	CSG
XmNtopShadowColor	**XmCTopShadow-Color**	**Pixel**	*dynamic*	CSG
XmNtopShadow-Pixmap	**XmCTopShadow-Pixmap**	**Pixmap**	*dynamic*	CSG
XmNtraversalOn	**XmCTraversalOn**	**Boolean**	*True*	CSG

XmNunitType	**XmCUnitType**	**unsigned char**	*dynamic*	CSG
XmNuserData	**XmCUserData**	**XtPointer**	*NULL*	CSG

Core Resource Set				
Name	**Class**	**Type**	**Default**	**Access**
XmNaccelerators	**XmCAccelerators**	**XtAccelerators**	*dynamic*	CSG
XmNancestor-Sensitive	**XmCSensitive**	**Boolean**	*dynamic*	G
XmNbackground	**XmCBackground**	**Pixel**	*dynamic*	CSG
XmNbackground-Pixmap	**XmCPixmap**	**Pixmap**	**XmUNSPECIFIED_-PIXMAP**	CSG
XmNborderColor	**XmCBorderColor**	**Pixel**	**XtDefaultForeground**	CSG
XmNborderPixmap	**XmCPixmap**	**Pixmap**	**XmUNSPECIFIED_-PIXMAP**	CSG
XmNborderWidth	**XmCBorderWidth**	**Dimension**	0	CSG
XmNcolormap	**XmCColormap**	**Colormap**	*dynamic*	CG
XmNdepth	**XmCDepth**	**int**	*dynamic*	CG
XmNdestroyCallback	**XmCCallback**	**XtCallbackList**	*NULL*	C
XmNheight	**XmCHeight**	**Dimension**	*dynamic*	CSG
XmNinitial-ResourcesPersistent	**XmCInitial-ResourcesPersistent**	**Boolean**	*True*	C
XmNmapped-WhenManaged	**XmCMapped-WhenManaged**	**Boolean**	*True*	CSG
XmNscreen	**XmCScreen**	**Screen ***	*dynamic*	CG
XmNsensitive	**XmCSensitive**	**Boolean**	*True*	CSG
XmNtranslations	**XmCTranslations**	**XtTranslations**	*dynamic*	CSG
XmNwidth	**XmCWidth**	**Dimension**	*dynamic*	CSG
XmNx	**XmCPosition**	**Position**	0	CSG
XmNy	**XmCPosition**	**Position**	0	CSG

ExmMenuButton provides the following translations:

EnterWindow:
　　　　MenuButtonEnter()

LeaveWindow:	**MenuButtonLeave()**
BtnDown:	**BtnDown()**
BtnUp:	**BtnUp()**
:osfActivate:	**ArmAndActivate()**
:osfCancel:	**MenuEscape()**
:osfHelp:	**MenuButtonHelp()**
~s ~m ~a <Key>Return:	**ArmAndActivate()**
~s ~m ~a <Key>space:	**ArmAndActivate()**

In addition, **ExmMenuButton** provides the following traversal translations:

Unmap:	**Unmap()**
FocusOut:	**FocusOut()**
FocusIn:	**FocusIn()**
osfCancel:	**MenuEscape()**
osfLeft:	**MenuTraverseLeft()**
osfRight:	**MenuTraverseRight()**
osfUp:	**MenuTraverseUp()**
osfDown:	**MenuTraverseDown()**

Action Routines

The **ExmMenuButton** action routines are

ArmAndActivate():
Unposts all menus in the menu hierarchy and, unless the button is already armed, calls the **XmNactivateCallback** callbacks.

BtnDown(): This action unposts any menus posted by the MenuButton's parent menu, disables keyboard traversal for the menu, and enables mouse traversal for the menu. It draws the shadow in the armed state.

BtnUp(): This action unposts all menus in the menu hierarchy and activates the MenuButton. It calls the **XmNactivateCallback** callbacks.

MenuButtonEnter:
If keyboard traversal is enabled, this action does nothing. Otherwise, it draws the shadow in the armed state and calls the **XmNarmCallback** callbacks.

MenuButtonHelp():
In a Pulldown or Popup MenuPane, unposts all menus in the menu hierarchy and, when the shell's keyboard focus policy is **XmEXPLICIT**, restores keyboard focus to the widget that had the focus before the menu system was entered. This action calls the callbacks for **XmNhelpCallback** if any exist. If there are no help callbacks for this widget, this action calls the help callbacks for the nearest ancestor that has them.

MenuButtonLeave:
If keyboard traversal is enabled, this action does nothing. Otherwise, it draws the shadow in the unarmed state and calls the **XmNdisarmCallback** callbacks.

The following actions are handled by the **XmRowColumn** widget:

- MenuEscape
- MenuTraverseDown
- MenuTraverseLeft
- MenuTraverseRight
- MenuTraverseUp

Virtual Bindings

The bindings for virtual keys are vendor specific. For information about bindings for virtual buttons and keys, see **VirtualBindings**(3).

Related Information

Core(3), **ExmSimple**(3), **ExmString**(3), and **XmPrimitive**(3).

ExmPanner

Purpose

The Panner widget class

Synopsis

#include <Exm/Panner.h>

Description

ExmPanner is a demonstration widget. OSF provides this widget solely to teach programmers how to write their own Motif widgets. OSF does not support this widget in any way.

ExmPanner is a Motif version of the Athena Panner widget. The source code for the **ExmPanner** widget illustrates how to do the following:

- Write a two-dimensional navigator widget
- Install the *XmQTnavigator* trait

In order to properly exercise the **ExmPanner** widget, a test application should associate **ExmPanner** with a scrollable widget. A scrollable widget, such as **XmScrolledWindow**, is one that holds the *XmQTscrollFrame* trait. In order to make this association, the test application must call the **addNavigator** trait method of the *XmQTscrollFrame* trait.

Once attached to a scrollable widget, the **ExmPanner** widget displays a rectangle floating inside its window. This rectangle is called a slider. As the user moves this slider, different parts of the scrollable widget become visible. Thus, **ExmPanner** is similar to an **XmScrollBar** widget. The primary difference is that **XmScrollBar** only permits navigation in one dimension, but **ExmPanner** permits simultaneous navigation in two dimensions.

As the user moves the **ExmPanner** widget, the associated scrollable widget will automatically scroll.

If **ExmPanner** is associated with a scrollable widget, then the scrollable widget will control the values of the **ExmPannerCanvasWidth**, **ExmPannerCanvasHeight**, **ExmPannerSliderWidth**, and **ExmPannerSliderHeight** resources. That is, a user or application cannot control the values of these resources when the **ExmPanner** is associated with a scrollable widget.

ExmPanner ignores the value of the **ExmNsimpleShape** resource of **ExmSimple**.

Classes

ExmPanner inherits behavior and resources from **Core**, **XmPrimitive**, and **ExmSimple**.

The class pointer is **exmPannerWidgetClass**.

The class name is **ExmPanner**.

New Resources

The following table defines a set of widget resources used by the programmer to specify data. The programmer can also set the resource values for the inherited classes to set attributes for this widget. To reference a resource by name or by class in a **.Xdefaults** file, remove the **XmN**, **ExmN**, **XmC** or **ExmC** prefix and use the remaining letters. To specify one of the defined values for a resource in a **.Xdefaults** file, remove the **Xm** or **Exm** prefix and use the remaining letters (in either lowercase or uppercase, but include any underscores between words). The codes in the "Access" column indicate if the given resource can be set at creation time (C), set by using **XtSetValues** (S), retrieved by using **XtGetValues** (G), or is not applicable (N/A).

ExmPanner Resource Set

Name	**Class**	**Type**	**Default**	**Access**
ExmNcanvasHeight	**ExmCCanvasHeight**	**Dimension**	0	CSG
ExmNcanvasWidth	**ExmCCanvasWidth**	**Dimension**	0	CSG
ExmNreportCallback	**ExmCReportCallback**	**XtCallbackList**	*NULL*	C
ExmNrubberBand	**ExmCRubberBand**	**Boolean**	*false*	CSG
ExmNsliderX	**ExmCSliderX**	**Position**	0	CSG
ExmNsliderY	**ExmCSliderY**	**Position**	0	CSG
ExmNsliderHeight	**ExmCSliderHeight**	**Dimension**	0	CSG
ExmNsliderWidth	**ExmCSliderWidth**	**Dimension**	0	CSG

ExmNcanvasHeight
Specifies the vertical size of the canvas.

ExmNcanvasWidth
Specifies the horizontal size of the canvas.

ExmNreportCallback
Specifies a list of callbacks that is called as a result of a Btn1Up or Btn2Up action.

ExmNrubberBand
If *false*, the widget holding the *XmQTscrollFrame* trait will pan on a Btn1Motion or Btn2Motion event. If *true*, the widget holding the *XmQTscrollFrame* trait will only pan on a Btn1Up or Btn2Up event.

ExmNsliderX
Specifies the horizontal position of the leftmost part of the slider. The horizontal position will be based on a ratio of the value of **ExmNsliderX** to the value of **ExmNcanvasWidth**.

ExmNsliderY
Specifies the starting vertical position of the leftmost part of the slider. The starting vertical position will be based on a ratio of the value of **ExmNsliderX** to the value of **ExmNcanvasHeight**.

ExmNsliderHeight
Specifies the logical height of the slider. The height will be based on a ratio of the value of **ExmNsliderHeight** to the value of **ExmNcanvasHeight**.

ExmNsliderWidth
Specifies the logical width of the slider. The width will be based on a ratio of the value of **ExmNsliderWidth** to the value of **ExmNcanvasWidth**.

ExmSimple Resource Set				
Name	**Class**	**Type**	**Default**	**Access**
ExmNsimpleShape	**ExmCSimpleShape**	**unsigned char**	ExmSHAPE_OVAL	CSG
XmNmarginHeight	**XmCMarginHeight**	**Dimension**	4	CSG
XmNmarginWidth	**XmCMarginWidth**	**Dimension**	4	CSG

XmPrimitive Resource Set				
Name	**Class**	**Type**	**Default**	**Access**
XmNbottom-ShadowColor	**XmCBottom-ShadowColor**	**Pixel**	*dynamic*	CSG

XmNbottom-ShadowPixmap	**XmCBottom-ShadowPixmap**	**Pixmap**	**XmUNSPECIFIED_-PIXMAP**	CSG
XmNconvertCallback	**XmCCallback**	**XtCallbackList**	*NULL*	C
XmNforeground	**XmCForeground**	**Pixel**	*dynamic*	CSG
XmNhelpCallback	**XmCCallback**	**XtCallbackList**	*NULL*	C
XmNhighlightColor	**XmCHighlightColor**	**Pixel**	*dynamic*	CSG
XmNhighlightOn-Enter	**XmCHighlightOn- Enter**	**Boolean**	*false*	CSG
XmNhighlightPixmap	**XmCHighlightPixmap**	**Pixmap**	*dynamic*	CSG
XmNhighlight-Thickness	**XmCHighlight-Thickness**	**Dimension**	2	CSG
XmNlayoutDirection	**XmCLayoutDirection**	**XmDirection**	*dynamic*	CG
XmNnavigationType	**XmCNavigationType**	**XmNavigationType**	**XmSTICKY_TAB_-GROUP**	CSG
XmNpopupHandler-Callback	**XmCCallback**	**XtCallbackList**	*NULL*	C
XmNshadow-Thickness	**XmCShadow- Thickness**	**Dimension**	2	CSG
XmNtopShadowColor	**XmCTopShadowColor**	**Pixel**	*dynamic*	CSG
XmNtopShadow-Pixmap	**XmCTopShadow-Pixmap**	**Pixmap**	*dynamic*	CSG
XmNtraversalOn	**XmCTraversalOn**	**Boolean**	*true*	CSG
XmNunitType	**XmCUnitType**	**unsigned char**	*dynamic*	CSG
XmNuserData	**XmCUserData**	**XtPointer**	*NULL*	CSG

Core Resource Set				
Name	**Class**	**Type**	**Default**	**Access**
XmNaccelerators	**XmCAccelerators**	**XtAccelerators**	*dynamic*	CSG
XmNancestor-Sensitive	**XmCSensitive**	**Boolean**	*dynamic*	G
XmNbackground	**XmCBackground**	**Pixel**	*dynamic*	CSG
XmNbackground-Pixmap	**XmCPixmap**	**Pixmap**	**XmUNSPECIFIED_-PIXMAP**	CSG

XmNborderColor	**XmCBorderColor**	**Pixel**	**XtDefaultForeground**	CSG
XmNborderPixmap	**XmCPixmap**	**Pixmap**	**XmUNSPECIFIED_-PIXMAP**	CSG
XmNborderWidth	**XmCBorderWidth**	**Dimension**	0	CSG
XmNcolormap	**XmCColormap**	**Colormap**	*dynamic*	CG
XmNdepth	**XmCDepth**	**int**	*dynamic*	CG
XmNdestroyCallback	**XmCCallback**	**XtCallbackList**	*NULL*	C
XmNheight	**XmCHeight**	**Dimension**	*dynamic*	CSG
XmNinitialResources-Persistent	**XmCInitialResources-Persistent**	**Boolean**	*true*	C
XmNmapped-WhenManaged	**XmCMapped-WhenManaged**	**Boolean**	*true*	CSG
XmNscreen	**XmCScreen**	**Screen ***	*dynamic*	CG
XmNsensitive	**XmCSensitive**	**Boolean**	*true*	CSG
XmNtranslations	**XmCTranslations**	**XtTranslations**	*dynamic*	CSG
XmNwidth	**XmCWidth**	**Dimension**	*dynamic*	CSG
XmNx	**XmCPosition**	**Position**	0	CSG
XmNy	**XmCPosition**	**Position**	0	CSG

Translations

ExmPanner provides the following translations:

<Unmap>: **PrimitiveUnmap()**

<Enter>: **PrimitiveEnter()**

<Leave>: **PrimitiveLeave()**

<FocusIn>: **PrimitiveFocusIn()**

<FocusOut>:
PrimitiveFocusOut()

:<Key>osfActivate:
PrimitiveParentActivate()

:<Key>osfHelp:
PrimitiveHelp()

:<Key>osfCancel:
ExmPannerAbort()

~s ~c ~m ~a <Btn1Down>:
ExmPannerStart(1)

~s ~c ~m ~a <Btn1Motion>:
ExmPannerMove()

~s ~c ~m ~a <Btn1Up>:
ExmPannerNotify() ExmPannerStop()

~s ~c ~m ~a <Btn2Down>:
ExmPannerStart(2)

~s ~c ~m ~a <Btn2Motion>:
ExmPannerMove()

~s ~c ~m ~a <Btn2Up>:
ExmPannerNotify() ExmPannerStop()

:<Key>space:
ExmPannerSet(rubberband,toggle)

:<Key>osfPageUp:
ExmPannerPage(+0, -1p)

:c <Key>**osfPageUp**:
ExmPannerPage(+0, -1c)

:<Key>osfPageDown:
ExmPannerPage(+0, +1p)

:c <Key>osfPageDown:
ExmPannerPage(+0, +1c)

:<Key>osfPageLeft:
ExmPannerPage(-1p, +0)

:c <Key>osfPageLeft:
ExmPannerPage(-1c, +0)

:<Key>osfPageRight:
ExmPannerPage(+1p, +0)

:c <Key>osfPageRight:
ExmPannerPage(+1c, +0)

:<Key>osfLeft:
    ExmPannerPage(-1, +0)

:c <Key>osfLeft:
    ExmPannerPage(-1p, +0)

:<Key>osfRight:
    ExmPannerPage(+1, +0)

:c <Key>osfRight:
    ExmPannerPage(+1p, +0)

:<Key>osfUp:
    ExmPannerPage(+0, -1)

:c <Key>osfUp:
    ExmPannerPage(+0, -1p)

:<Key>osfDown:
    ExmPannerPage(+0, +1)

:c <Key>osfDown:
    ExmPannerPage(+0, +1p)

:<Key>osfEndLine:
    ExmPannerPage(+1c, +1c)

:<Key>osfBeginLine:
    ExmPannerPage(0,0)

~s ~m ~a <Key>Return:
    PrimitiveParentActivate()

s ~m ~a <Key>Tab:
    PrimitivePrevTabGroup()

~m ~a <Key>Tab:
    PrimitiveNextTabGroup()

Action Routines

All the actions that begin with the *Primitive* prefix are defined by the **XmPrimitive** widget. (See the **XmPrimitive**(3) reference page of the *Motif 2.1—Programmer's Reference* for details.) The actions defined by **ExmPanner** are as follows:

ExmPannerAbort():
    Discontinues whatever pan operation is currently in progress. Returns the slider to the position it held when **ExmPannerStart** was invoked.

ExmPannerMove():

If **ExmNrubberBand** is *false*, this action moves the outline of the slider. If **ExmNrubberBand** is *true*, this action moves the slider itself.

ExmPannerNotify():

Calls the callbacks for **XmNreportCallback**.

ExmPannerPage(*x*,*y*):

Moves the slider in two dimensions. The first argument represents movement in the *x* dimension and the second argument represents the movement in the *y* dimension. Each argument must consist of an integer, optionally followed by the suffix **p** or **c**. A positive value classnameizes a move down or right. A negative value classnameizes a move up or left. The **p** suffix classnameizes "page." A page is defined as the slider size in that dimension. So, for example, if the slider height is 10 pixels, then a *y* argument of **--2p** moves the slider up 2 pages, which is 20 pixels. The **c** suffix classnameizes "canvas". The coordinates of the upper left of the canvas are (0c,0c), while the lower right of the canvas is at (1c,1c). If the argument does not contain a suffix, then the argument is interpreted as a relative pixel move. For example, a *y* argument of **--40** moves the slider 40 pixels up from its current position.

ExmPannerSet(**resource,value**):

Sets the value of the **XmNrubberBand** resource to **value**. That is, the first argument must be **XmNrubberBand** and the second argument must be **On**, **Off**, or **Toggle**.

ExmPannerStart():

Signals the start of a pan operation. This routine merely initializes several fields of the callback structure; this routine does not move the slider or perform any panning operations.

ExmPannerStop():

Signals the end of the current pan operation.

Virtual Bindings

The bindings for virtual keys are vendor specific. For information about bindings for virtual buttons and keys, see **VirtualBindings**(3).

Related Information

Core(3), **ExmCommandButton**(3), **ExmMenuButton**(3), **ExmSimple**(3), and **XmPrimitive**(3).

ExmSimple

Purpose

The Simple widget class demonstration widget

Synopsis

#include <Exm/Simple.h>

Description

ExmSimple is a demonstration widget. OSF provides this widget solely to teach programmers how to write their own Motif widgets. OSF does not support this widget in any way.

The source code for the **ExmSimple** widget illustrates how to do the following:

- Use Xlib drawing functions to render a simple visual inside a Motif widget
- Create a primitive widget that can serve as a superclass for other Motif widgets
- Create a Motif representation type
- Install the *XmQTcontainerItem* trait and define its trait methods
- Install the *XmQTcareParentVisual* trait and define its trait method

ExmSimple is an instantiable widget and is also used as a superclass for other widgets, such as **ExmString** and **ExmPanner**.

ExmSimple displays a simple visual. The value of the **ExmNsimpleShape** resource determines whether this visual is a rectangle or an oval. Note that none of the Exm subclasses of **ExmSimple** display this visual. That is, none of the Exm subclasses of **ExmSimple** inherit the **draw_visual** method of **ExmSimple**.

When **ExmSimple** is insensitive, the visual has a stippled appearance.

ExmSimple displays its visual in the center of the widget. A margin region lies between the the window decorations and the visual. The dimensions of the margin region are determined by the values of the **XmNmarginWidth** and **XmNmarginHeight** resources. An application cannot directly control the size of the visual; **ExmSimple** will automatically set the visual size so that it fits snugly inside the margin region.

The help callback is the only callback supported.

Classes

ExmSimple inherits behavior and resources from **Core** and **XmPrimitive**.

The class pointer is **exmSimpleWidgetClass**.

The class name is **ExmSimple**.

New Resources

The following table defines a set of widget resources used by the programmer to specify data. The programmer can also set the resource values for the inherited classes to set attributes for this widget. To reference a resource by name or by class in a **.Xdefaults** file, remove the **XmN**, *ExmN*, **XmC** or *ExmC* prefix and use the remaining letters. To specify one of the defined values for a resource in a **.Xdefaults** file, remove the **Xm** or **Exm** prefix and use the remaining letters (in either lowercase or uppercase, but include any underscores between words). The codes in the "Access" column indicate if the given resource can be set at creation time (C), set by using **XtSetValues** (S), retrieved by using **XtGetValues** (G), or is not applicable (N/A).

ExmSimple Resource Set

Name	Class	Type	Default	Access
ExmNsimpleShape	**ExmCSimpleShape**	**unsigned char**	**ExmSHAPE_OVAL**	CSG
XmNmarginHeight	**XmCMarginHeight**	**Dimension**	4	CSG
XmNmarginWidth	**XmCMarginWidth**	**Dimension**	4	CSG

ExmNsimpleShape

Specifies the shape to be displayed in the widget. The two possible values for this resource are *ExmSHAPE_OVAL* and *ExmSHAPE_RECTANGLE*. The *ExmSHAPE_OVAL* (the default) causes **ExmSimple** to display an oval and the *ExmSHAPE_RECTANGLE* causes **ExmSimple** to display a rectangle.

XmNmarginHeight

Specifies the distance between the inside top edge of the window decorations and the outside top edge of the graphics (the oval or rectangle). This resource also specifies the distance between the inside bottom edge of the window decorations and the outside bottom edge of the graphics. If the widget is not big enough to display both the graphics and the margin, the widget will reduce the margin height.

XmNmarginWidth

Specifies the distance between the inside left edge of the window decorations and the outside left edge of the graphics (the oval or rectangle). This resource also specifies the distance between the inside right edge of the window decorations and the outside right edge of the graphics. If the widget is not big enough to display both the graphics and the margin, the widget will reduce the margin width.

Inherited Resources

ExmSimple inherits behavior and resources from the following superclasses. For a complete description of each resource, refer to the reference page for that superclass.

XmPrimitive Resource Set				
Name	**Class**	**Type**	**Default**	**Access**
XmNbottom-ShadowColor	**XmCBottom-ShadowColor**	**Pixel**	*dynamic*	CSG
XmNbottom-ShadowPixmap	**XmCBottom-ShadowPixmap**	**Pixmap**	**XmUNSPECIFIED_-PIXMAP**	CSG
XmNconvertCallback	**XmCCallback**	**XtCallbackList**	*NULL*	C
XmNforeground	**XmCForeground**	**Pixel**	*dynamic*	CSG
XmNhelpCallback	**XmCCallback**	**XtCallbackList**	*NULL*	C
XmNhighlightColor	**XmCHighlightColor**	**Pixel**	*dynamic*	CSG
XmNhighlightOnEnter	**XmCHighlight-OnEnter**	**Boolean**	False	CSG
XmNhighlightPixmap	**XmCHighlightPixmap**	**Pixmap**	*dynamic*	CSG
XmNhighlight-Thickness	**XmCHighlight-Thickness**	**Dimension**	2	CSG
XmNlayoutDirection	**XmCLayout- Direction**	**XmDirection**	*dynamic*	CG
XmNnavigationType	**XmCNavigationType**	**XmNavigationType**	**XmNONE**	CSG
XmNpopup-HandlerCallback	**XmCCallback**	**XtCallbackList**	*NULL*	C
XmNshadowThickness	**XmCShadow-Thickness**	**Dimension**	2	CSG
XmNtopShadow-Color	**XmCTop-ShadowColor**	**Pixel**	*dynamic*	CSG

XmNtopShadow-Pixmap	**XmCTop-ShadowPixmap**	**Pixmap**	*dynamic*	CSG
XmNtraversalOn	**XmCTraversalOn**	**Boolean**	*True*	CSG
XmNunitType	**XmCUnitType**	**unsigned char**	*dynamic*	CSG
XmNuserData	**XmCUserData**	**XtPointer**	*NULL*	CSG

Core Resource Set				
Name	**Class**	**Type**	**Default**	**Access**
XmNaccelerators	**XmCAccelerators**	**XtAccelerators**	*dynamic*	CSG
XmNancestor-Sensitive	**XmCSensitive**	**Boolean**	*dynamic*	G
XmNbackground	**XmCBackground**	**Pixel**	*dynamic*	CSG
XmNbackground-Pixmap	**XmCPixmap**	**Pixmap**	**XmUNSPECIFIED_-PIXMAP**	CSG-
XmNborderColor	**XmCBorderColor**	**Pixel**	**XtDefaultForeground**	CSG
XmNborderPixmap	**XmCPixmap**	**Pixmap**	**XmUNSPECIFIED_-PIXMAP**	CSG
XmNborderWidth	**XmCBorderWidth**	**Dimension**	0	CSG
XmNcolormap	**XmCColormap**	**Colormap**	*dynamic*	CG
XmNdepth	**XmCDepth**	**int**	*dynamic*	CG
XmNdestroy- Callback	**XmCCallback**	**XtCallbackList**	*NULL*	C
XmNheight	**XmCHeight**	**Dimension**	*dynamic*	CSG
XmNinitial-ResourcesPersistent	**XmCInitial-ResourcesPersistent**	**Boolean**	*True*	C
XmNmapped-WhenManaged	**XmCMapped-WhenManaged**	**Boolean**	*True*	CSG
XmNscreen	**XmCScreen**	**Screen ***	*dynamic*	CG
XmNsensitive	**XmCSensitive**	**Boolean**	*True*	CSG
XmNtranslations	**XmCTranslations**	**XtTranslations**	*dynamic*	CSG
XmNwidth	**XmCWidth**	**Dimension**	*dynamic*	CSG
XmNx	**XmCPosition**	**Position**	0	CSG
XmNy	**XmCPosition**	**Position**	0	CSG

Translations

ExmSimple inherits all the translations of **XmPrimitive**. **ExmSimple** does not provide any additional translations beyond those defined by **XmPrimitive**.

Action Routines

ExmSimple provides no action routines of its own.

Related Information

Core(3) and **XmPrimitive**(3).

ExmString

Purpose

The String widget class

Synopsis

#include <Exm/String.h>

Description

ExmString is a demonstration widget. OSF provides this widget solely to teach programmers how to write their own Motif widgets. OSF does not support this widget in any way.

The source code for the **ExmString** widget illustrates how to do the following:

- Use the recommended resources for handling compound strings
- Install the *XmQTaccessTextual* trait and define its trait methods
- Calculate the baselines of each line of text in the compound string

ExmString is similar to the standard Motif toolkit widget **XmLabel**. The major difference between **ExmString** and **XmLabel** is that **XmLabel** can display either text or a pixmap, but **ExmString** can display only text.

ExmString is an instantiable widget and is also used as a superclass for other widgets, such as **ExmCommandButton**, **ExmStringTransfer**, and **ExmMenuButton**.

The **ExmString** widget receives enter and leave events, but it does not accept any button or key input.

The help callback is the only callback defined.

The text displayed by **ExmString** is a noneditable compound string. The *ExmNcompoundString* resource holds the value of this compound string. (See the *Motif 2.1—Programmer’s Guide* for more information on compound strings.) The text can be multilingual, multiline, and/or multifont. When an **ExmString** is insensitive, its text is stippled.

Unlike **XmLabel**, **ExmString** does not support accelerators or mnemonics.

ExmString forces the **XmNtraversalOn** resource to *False*.

Classes

ExmString inherits behavior and resources from **Core**, **XmPrimitive**, and **ExmSimple**.

The class pointer is **exmStringWidgetClass**.

The class name is **ExmString**.

New Resources

The following table defines a set of widget resources used by the programmer to specify data. The programmer can also set the resource values for the inherited classes to set attributes for this widget. To reference a resource by name or by class in a **.Xdefaults** file, remove the **XmN**, *ExmN*, **XmC** or *ExmC* prefix and use the remaining letters. To specify one of the defined values for a resource in a **.Xdefaults** file, remove the **Xm** or **Exm** prefix and use the remaining letters (in either lowercase or uppercase, but include any underscores between words). The codes in the "Access" column indicate if the given resource can be set at creation time (C), set by using **XtSetValues** (S), retrieved by using **XtGetValues** (G), or is not applicable (N/A).

ExmString Resource Set

Name	Class	Type	Default	Access
ExmNcompound-String	**ExmCCompound-String**	**XmString**	*NULL*	CSG
XmNalignment	**XmCAlignment**	**unsigned char**	**XmALIGNMENT_-CENTER**	CSG
XmNrecomputeSize	**XmCRecomputeSize**	**Boolean**	*True*	CSG
XmNrenderTable	**XmCRenderTable**	**XmRenderTable**	*dynamic*	CSG

ExmNcompoundString
Specifies the text to be displayed.

XmNalignment
Specifies the alignment of the text:

XmALIGNMENT_BEGINNING (left alignment)
Causes the left sides of the lines of text to be aligned with the left edge of the widget window.

XmALIGNMENT_CENTER (center alignment)
Causes each line of text to be centered half way between the left and right edges of the widget window.

XmALIGNMENT_END (right alignment)
Causes the right sides of the lines of text to be aligned with the right edge of the widget window.

The preceding descriptions for text are correct when **XmNstringDirection** is **XmSTRING_DIRECTION_L_TO_R**. When that resource is **XmSTRING_DIRECTION_R_TO_L**, the descriptions for **XmALIGNMENT_BEGINNING** and **XmALIGNMENT_END** are switched.

XmNrecomputeSize
Specifies a Boolean value that indicates whether the widget attempts to resize to accommodate its contents as a result of an **XtSetValues** resource value that would change the size of the widget. If *True*, the widget attempts to shrink or expand to exactly fit the compound string. If *False*, the widget never attempts to change size on its own.

XmNrenderTable
Specifies the render table of the text used in the widget. If this value is *NULL* at initialization, the parent hierarchy of the widget is searched for an ancestor that holds the *XmQTspecifyRenderTable* trait. If such an ancestor is found, the font list is initialized to **XmNbuttonRenderTable** (for button subclasses) or **XmNlabelRenderTable** of the ancestor widget. If no such ancestor is found, the default is implementation dependent. Refer to **XmRenderTable**(3) for more information on the creation and structure of a font list.

Inherited Resources

ExmString inherits behavior and resources from the following superclasses. For a complete description of each resource, refer to the reference page for that superclass.

ExmSimple Resource Set				
Name	**Class**	**Type**	**Default**	**Access**
ExmNsimpleShape	**ExmCSimpleShape**	**unsigned char**	**ExmSHAPE_- OVAL**	CSG
XmNmarginHeight	**XmCMarginHeight**	**Dimension**	4	CSG
XmNmarginWidth	**XmCMarginWidth**	**Dimension**	4	CSG

XmPrimitive Resource Set				
Name	**Class**	**Type**	**Default**	**Access**
XmNbottom-ShadowColor	**XmCBottom-ShadowColor**	**Pixel**	*dynamic*	CSG
XmNbottom-ShadowPixmap	**XmCBottom-ShadowPixmap**	**Pixmap**	**XmUNSPECIFIED_-PIXMAP**	CSG
XmNconvert- Callback	**XmCCallback**	**XtCallbackList**	*NULL*	C
XmNforeground	**XmCForeground**	**Pixel**	*dynamic*	CSG
XmNhelpCallback	**XmCCallback**	**XtCallbackList**	*NULL*	C
XmNhighlightColor	**XmCHighlightColor**	**Pixel**	*dynamic*	CSG
XmNhighlightOnEnter	**XmCHighlightOn-Enter**	**Boolean**	*False*	CSG
XmNhighlightPixmap	**XmCHighlight-Pixmap**	**Pixmap**	*dynamic*	CSG
XmNhighlight-Thickness	**XmCHighlight-Thickness**	**Dimension**	2	CSG
XmNlayout- Direction	**XmCLayout- Direction**	**XmDirection**	*dynamic*	CG
XmNnavigation- Type	**XmCNavigationType**	**XmNavigationType**	**XmNONE**	CSG
XmNpopup-HandlerCallback	**XmCCallback**	**XtCallbackList**	*NULL*	C
XmNshadow-Thickness	**XmCShadow-Thickness**	**Dimension**	2	CSG
XmNtop-ShadowColor	**XmCTop-ShadowColor**	**Pixel**	*dynamic*	CSG
XmNtop-ShadowPixmap	**XmCTop-ShadowPixmap**	**Pixmap**	*dynamic*	CSG
XmNtraversalOn	**XmCTraversalOn**	**Boolean**	*False*	CSG
XmNunitType	**XmCUnitType**	**unsigned char**	*dynamic*	CSG
XmNuserData	**XmCUserData**	**XtPointer**	*NULL*	CSG

Core Resource Set				
Name	**Class**	**Type**	**Default**	**Access**
XmNaccelerators	**XmCAccelerators**	**XtAccelerators**	*dynamic*	CSG

XmNancestor-Sensitive	**XmCSensitive**	**Boolean**	*dynamic*	G
XmNbackground	**XmCBackground**	**Pixel**	*dynamic*	CSG
XmNbackground-Pixmap	**XmCPixmap**	**Pixmap**	**XmUNSPECIFIED_-PIXMAP**	CSG
XmNborderColor	**XmCBorderColor**	**Pixel**	**XtDefault- Foreground**	CSG
XmNborderPixmap	**XmCPixmap**	**Pixmap**	**XmUNSPECIFIED_-PIXMAP**	CSG
XmNborderWidth	**XmCBorderWidth**	**Dimension**	0	CSG
XmNcolormap	**XmCColormap**	**Colormap**	*dynamic*	CG
XmNdepth	**XmCDepth**	**int**	*dynamic*	CG
XmNdestroyCallback	**XmCCallback**	**XtCallbackList**	*NULL*	C
XmNheight	**XmCHeight**	**Dimension**	*dynamic*	CSG
XmNinitial-ResourcesPersistent	**XmCInitial-ResourcesPersistent**	**Boolean**	*True*	C
XmNmapped-WhenManaged	**XmCMapped-WhenManaged**	**Boolean**	*True*	CSG
XmNscreen	**XmCScreen**	**Screen ***	*dynamic*	CG
XmNsensitive	**XmCSensitive**	**Boolean**	*True*	CSG
XmNtranslations	**XmCTranslations**	**XtTranslations**	*dynamic*	CSG
XmNwidth	**XmCWidth**	**Dimension**	*dynamic*	CSG
XmNx	**XmCPosition**	**Position**	0	CSG
XmNy	**XmCPosition**	**Position**	0	CSG

Translations

ExmString inherits all the translations of **XmPrimitive**. **ExmString** does not provide any additional translations beyond those defined by **XmPrimitive**.

Action Routines

ExmString provides no action routines of its own.

Related Information

Core(3), **ExmCommandButton**(3), **ExmMenuButton**(3), **ExmSimple**(3), and **XmPrimitive**(3).

ExmStringTransfer

Purpose

The StringTransfer widget class

Synopsis

#include <Exm/StringTrans.h>

Description

ExmStringTransfer is a demonstration widget. OSF provides this widget solely to teach programmers how to write their own Motif widgets. OSF does not support this widget in any way.

ExmStringTransfer is a subclass of **ExmString**. **ExmStringTransfer** inherits all the visuals of **ExmString**; therefore, **ExmStringTransfer** displays one compound string. Unlike **ExmString**, **ExmStringTransfer** supports data transfer. That is, a user can transfer data into or out of an **ExmStringTransfer** widget. In fact, the purpose of this widget is to demonstrate how to implement UTM data transfer in a widget.

ExmStringTransfer is an instantiable widget and can also used as a superclass for other widgets.

The **ExmStringTransfer** class installs the *XmQTtransfer* trait and implements this trait’s **convertProc** and **destinationProc** trait methods.

ExmStringTransfer supports two callbacks:

- **XmNdestinationCallback**, which allows applications to intercede a data transfer on behalf of the destination widget.
- **XmNconvertCallback**, which allows applications to intercede a data transfer on behalf of the source widget. This resource is inherited from **XmPrimitive**.

Data Transfer Behavior

ExmStringTransfer can be the source or destination of a primary, clipboard, or drag and drop transfer. **ExmStringTransfer** does not support secondary transfers.

Unlike more sophisticated widgets such as **XmText**, a user cannot copy selected portions of an **ExmStringTransfer**. Instead, a user can only copy the entire compound string held by *ExmNcompoundString*, not a subset of it. Similarly, when an **ExmStringTransfer** is the destination, any text copied into it will overwrite all of the previous contents.

As a source of data, **ExmStringTransfer** supports the following targets and associated conversions of data to these targets:

locale
: If the *locale* target matches the widget's locale, the widget transfers *ExmNcompoundString* in the encoding of the locale.

COMPOUND_TEXT
: The widget transfers *ExmNcompoundString* as type **COMPOUND_TEXT**.

STRING
: The widget transfers *ExmNcompoundString* as type **STRING**.

TEXT
: If *ExmNcompoundString* is fully convertible to the encoding of the locale, the widget transfers the value as text in the encoding of the locale. Otherwise, the widget transfers the value as type **COMPOUND_TEXT**.

_MOTIF_CLIPBOARD_TARGETS
: The widget transfers, as type **ATOM**, a list of the targets it supports for immediate transfer for the *CLIPBOARD* selection. These include _MOTIF_COMPOUND_STRING. If *ExmNcompoundString* is fully convertible to **STRING**, these also include **STRING**; otherwise, they also include **COMPOUND_TEXT**.

_MOTIF_COMPOUND_STRING
: The widget transfers *ExmNcompoundString* as a compound string in byte stream format.

_MOTIF_EXPORT_TARGETS
: The widget transfers, as type *ATOM*, a list of the targets to be used as the value of the DragContext's **XmNexportTargets** in a drag-and-drop transfer. These include _MOTIF_COMPOUND_STRING, **COMPOUND_TEXT**, the encoding of the locale, **STRING**, **TEXT**, **BACKGROUND**, and **FOREGROUND**.

As a source of data, **ExmStringTransfer** also supports the following standard Motif targets:

BACKGROUND
: The widget transfers **XmNbackground** as type **PIXEL**.

CLASS The widget finds the first shell in the widget hierarchy that has a WM_CLASS property and transfers the contents as text in the current locale.

CLIENT_WINDOW
The widget finds the first shell in the widget hierarchy and transfers its window as type **WINDOW**.

COLORMAP
The widget transfers **XmNcolormap** as type **COLORMAP**.

FOREGROUND
The widget transfers **XmNforeground** as type **PIXEL**.

NAME The widget finds the first shell in the widget hierarchy that has a WM_NAME property and transfers the contents as text in the current locale.

TARGETS
The widget transfers, as type **ATOM**, a list of the targets it supports. These include the standard targets in this list. These also include _MOTIF_COMPOUND_STRING, **COMPOUND_TEXT**, the encoding of the locale, **STRING**, and **TEXT**.

TIMESTAMP
The widget transfers the timestamp used to acquire the selection as type **INTEGER**.

_MOTIF_RENDER_TABLE
The widget transfers **XmNrenderTable** if it exists, or else the default text render table, as type **STRING**.

As a destination for data, when **ExmStringTransfer** receives data that can be converted to a compound string, it sets *ExmNcompoundString* to the transferred compound string.

As a destination, **ExmStringTransfer** chooses a target and requests conversion of the selection to that target. If the encoding of the locale is present in the list of available targets, **ExmStringTransfer** chooses a requested target from the available targets in the following order of preference:

1. _MOTIF_COMPOUND_STRING
2. **TEXT**
3. **COMPOUND_TEXT**

4. The encoding of the locale
5. **STRING**

If the encoding of the locale is not present in the list of available targets, **ExmStringTransfer** chooses a requested target from the available targets in the following order of preference:

1. _MOTIF_COMPOUND_STRING
2. **COMPOUND_TEXT**
3. **STRING**

Classes

ExmStringTransfer inherits behavior and resources from **Core**, **XmPrimitive**, **ExmSimple**, and **ExmString**.

The class pointer is **exmStringTransferWidgetClass**.

The class name is **ExmStringTransfer**.

New Resources

The following table defines a set of widget resources used by the programmer to specify data. The programmer can also set the resource values for the inherited classes to set attributes for this widget. To reference a resource by name or by class in a **.Xdefaults** file, remove the **XmN**, *ExmN*, **XmC** or *ExmC* prefix and use the remaining letters. To specify one of the defined values for a resource in a **.Xdefaults** file, remove the **Xm** or **Exm** prefix and use the remaining letters (in either lowercase or uppercase, but include any underscores between words). The codes in the "Access" column indicate if the given resource can be set at creation time (C), set by using **XtSetValues** (S), retrieved by using **XtGetValues** (G), or is not applicable (N/A).

ExmStringTransfer Resource Set				
Name	**Class**	**Type**	**Default**	**Access**
XmNdestinationCallback	**XmCCallback**	**XtCallbackList**	*NULL*	C

XmNdestinationCallback

Specifies a list of callbacks called when the widget is the destination of a transfer operation. The type of the structure whose address is passed to these callbacks is **XmDestinationCallbackStruct**. The reason is **XmCR_OK**.

Inherited Resources

ExmStringTransfer inherits behavior and resources from the following superclasses. For a complete description of each resource, refer to the reference page for that superclass.

ExmString Resource Set				
Name	**Class**	**Type**	**Default**	**Access**
ExmNcompound-String	**ExmCCompound-String**	**XmString**	*NULL*	CSG
XmNalignment	**XmCAlignment**	unsigned char	**XmALIGNMENT_-CENTER**	CSG
XmNrecomputeSize	**XmCRecomputeSize**	**Boolean**	*True*	CSG
XmNrenderTable	**XmCRenderTable**	**XmRenderTable**	*dynamic*	CSG

ExmSimple Resource Set				
Name	**Class**	**Type**	**Default**	**Access**
ExmNsimpleShape	**ExmCSimpleShape**	**unsigned char**	**ExmSHAPE_- OVAL**	CSG
XmNmarginHeight	**XmCMarginHeight**	**Dimension**	4	CSG
XmNmarginWidth	**XmCMarginWidth**	**Dimension**	4	CSG

XmPrimitive Resource Set				
Name	**Class**	**Type**	**Default**	**Access**
XmNbottom-ShadowColor	**XmCBottom-ShadowColor**	**Pixel**	*dynamic*	CSG
XmNbottom-ShadowPixmap	**XmCBottom-ShadowPixmap**	**Pixmap**	**XmUNSPECIFIED_-PIXMAP**	CSG
XmNconvert- Callback	**XmCCallback**	**XtCallbackList**	*NULL*	C
XmNforeground	**XmCForeground**	**Pixel**	*dynamic*	CSG
XmNhelpCallback	**XmCCallback**	**XtCallbackList**	*NULL*	C
XmNhighlightColor	**XmCHighlightColor**	**Pixel**	*dynamic*	CSG
XmNhighlightOnEnter	**XmCHighlight-OnEnter**	**Boolean**	*False*	CSG

XmNhighlightPixmap	**XmCHighlight-Pixmap**	**Pixmap**	*dynamic*	CSG
XmNhighlight-Thickness	**XmCHighlight-Thickness**	**Dimension**	2	CSG
XmNlayoutDirection	**XmCLayout- Direction**	**XmDirection**	*dynamic*	CG
XmNnavigationType	**XmCNavigationType**	**XmNavigationType**	**XmNONE**	CSG
XmNpopup-HandlerCallback	**XmCCallback**	**XtCallbackList**	*NULL*	C
XmNshadow-Thickness	**XmCShadow-Thickness**	**Dimension**	2	CSG
XmNtopShadow-Color	**XmCTop-ShadowColor**	**Pixel**	*dynamic*	CSG
XmNtopShadow-Pixmap	**XmCTop-ShadowPixmap**	**Pixmap**	*dynamic*	CSG
XmNtraversalOn	**XmCTraversalOn**	**Boolean**	*True*	CSG
XmNunitType	**XmCUnitType**	**unsigned char**	*dynamic*	CSG
XmNuserData	**XmCUserData**	**XtPointer**	*NULL*	CSG

Core Resource Set				
Name	**Class**	**Type**	**Default**	**Access**
XmNaccelerators	**XmCAccelerators**	**XtAccelerators**	*dynamic*	CSG
XmNancestor-Sensitive	**XmCSensitive**	**Boolean**	*dynamic*	G
XmNbackground	**XmCBackground**	**Pixel**	*dynamic*	CSG
XmNbackground-Pixmap	**XmCPixmap**	**Pixmap**	**XmUNSPECIFIED_-PIXMAP**	CSG
XmNborderColor	**XmCBorderColor**	**Pixel**	**XtDefaultForeground**	CSG
XmNborderPixmap	**XmCPixmap**	**Pixmap**	**XmUNSPECIFIED_-PIXMAP**	CSG
XmNborderWidth	**XmCBorderWidth**	**Dimension**	0	CSG
XmNcolormap	**XmCColormap**	**Colormap**	*dynamic*	CG
XmNdepth	**XmCDepth**	**int**	*dynamic*	CG
XmNdestroyCallback	**XmCCallback**	**XtCallbackList**	*NULL*	C

XmNheight	**XmCHeight**	**Dimension**	*dynamic*	CSG
XmNinitial-ResourcesPersistent	**XmCInitial-ResourcesPersistent**	**Boolean**	*True*	C
XmNmapped-WhenManaged	**XmCMapped-WhenManaged**	**Boolean**	*True*	CSG
XmNscreen	**XmCScreen**	**Screen ***	*dynamic*	CG
XmNsensitive	**XmCSensitive**	**Boolean**	*True*	CSG
XmNtranslations	**XmCTranslations**	**XtTranslations**	*dynamic*	CSG
XmNwidth	**XmCWidth**	**Dimension**	*dynamic*	CSG
XmNx	**XmCPosition**	**Position**	0	CSG
XmNy	**XmCPosition**	**Position**	0	CSG

Callback Information

A pointer to the following callback structure is passed to the **XmNdestinationCallback** procedures:

```
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 *CLIPBOARD*, *PRIMARY*, and _MOTIF_DROP.
operation	Indicates the type of transfer operation requested.

- When the selection is **PRIMARY**, 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. Otherwise, *destination_data* is *NULL*.

location_data
Contains information about the location where data is to be transferred. The value is always *NULL* when the selection is **SECONDARY** or **CLIPBOARD**. If the value is *NULL*, the data is to be inserted at the widget's cursor position. The value of *location_data* is only valid for the duration of a transfer. Once transfer done procedures start to be called, *location_data* will no longer be stable.

time Indicates the time when the transfer operation began.

Translations

ExmStringTransfer provides the following translations:

<EnterWindow>:
  PrimitiveEnter()

<LeaveWindow>:
  PrimitiveLeave()

c <Btn1Down>:
  ExmStringTransferMoveFocus()

<Btn1Down>:
  ExmStringTransferMoveFocus() ExmStringTransferCopyPrimary()

<Btn2Down>:
  ExmStringTransferProcessDrag()

:<Key>osfPaste:
  ExmStringTransferPasteClipboard()

:<Key>osfCut:
  ExmStringTransferCopyClipboard()

:s <Key>osfInsert:
  ExmStringTransferPasteClipboard()

:s <Key>osfDelete:
  ExmStringTransferCopyClipboard()

:<Key>osfActivate:
  PrimitiveParentActivate()

:<Key>osfCancel:
  PrimitiveParentCancel()

:<Key>osfHelp:
  PrimitiveHelp()

~s ~m ~a <Key>Return:XS
  PrimitiveParentActivate()

Action Routines

All the actions that begin with the *Primitive* prefix are defined by the **XmPrimitive** widget. (See the **XmPrimitive**(3) reference page of the *Motif 2.1—Programmer’s Reference* for details.) The actions defined by **ExmStringTransfer** are as follows:

ExmStringTransferCopyClipboard():
  If this widget owns the primary selection, this action copies the contents of *ExmNcompoundString* to the clipboard. This action calls

the **XmNconvertCallback** procedures, possibly multiple times, for the **CLIPBOARD** selection.

ExmStringTransferCopyPrimary():
: Calls the **XmNdestinationCallback** procedures for the **PRIMARY** selection and the **XmCOPY** operation. It calls the selection owner's **XmNconvertCallback** procedures, possibly multiple times, for the **PRIMARY** selection.

ExmStringTransferMoveFocus():
: Causes the widget to take keyboard focus, without activating the widget. This action does not change the selection; it merely changes the keyboard focus.

ExmStringTransferPasteClipboard():
: Pastes the contents of the clipboard into the widget, overwriting any text that was previously displayed. This action calls any **XmNdestinationCallback** procedures registered, passing **CLIPBOARD** in the *selection* member of the **XmDestinationCallbackStruct**. The pasted text becomes the new value of the *ExmNcompoundString* resource.

ExmStringTransferPastePrimary():
: Pastes the primary selection into the specified widget. This action overwrites any text that was previously displayed. The pasted text becomes the new value of the *ExmNcompoundString* resource.

ExmStringTransferProcessDrag():
: Drags the contents of the *ExmNcompoundString* resource. This action sets the **XmNconvertProc** of the DragContext to a function that calls the **XmNconvertCallback** procedures, possibly multiple times, for the _MOTIF_DROP selection.

Virtual Bindings

The bindings for virtual keys are vendor specific. For information about bindings for virtual buttons and keys, see **VirtualBindings**(3).

Related Information

Core(3), **ExmSimple**(3), **ExmString**(3), **XmPrimitive**(3), and **XmQTtransfer**(3).

ExmTabButton

Purpose

The TabButton widget class demonstration widget

Synopsis

#include <Exm/TabB.h>

Description

ExmTabButton is a demonstration widget. OSF provides this widget solely to teach programmers how to write their own Motif widgets. OSF does not support this widget in any way.

The source code for the **ExmTabButton** widget illustrates how to do the following:

- Use the *XmQTjoinSide* trait
- Use several Xme drawing functions

ExmTabButton is an instantiable widget. **ExmTabButton** is a subclass of the **ExmCommandButton** widget, which is itself a subclass of the **ExmString** widget. Therefore, **ExmTabButton** inherits all of **ExmString**’s resources for rendering compound strings.

When a user activates an **ExmTabButton** widget, the widget calls the callback procedure associated with the **XmNactivateCallback** resource.

Classes

ExmTabButton inherits behavior and resources from **Core**, **XmPrimitive**, **ExmSimple**, **ExmString**, and **ExmCommandButton**.

The class pointer is **exmTabButtonWidgetClass**.

The class name is **ExmTabButton**.

New Resources

The following table defines a set of widget resources used by the programmer to specify data. The programmer can also set the resource values for the inherited classes to set attributes for this widget. To reference a resource by name or by class in a **.Xdefaults** file, remove the **XmN**, *ExmN*, **XmC** or *ExmC* prefix and use the remaining letters. To specify one of the defined values for a resource in a **.Xdefaults** file, remove the **Xm** or **Exm** prefix and use the remaining letters (in either lowercase or uppercase, but include any underscores between words). The codes in the "Access" column indicate if the given resource can be set at creation time (C), set by using **XtSetValues** (S), retrieved by using **XtGetValues** (G), or is not applicable (N/A).

ExmTabButton Resource Set				
Name	**Class**	**Type**	**Default**	**Access**
ExmNopenSide	**ExmCOpenSide**	**ExmROpenSide**	**XmLEFT**	CSG

ExmNopenSide

Specifies the side of the child widget that is to be joined to the manager. *ExmNopenSide* must be one of the following: **XmNONE**, **XmLEFT**, **XmRIGHT**, **XmTOP**, or **XmBOTTOM**. For example, if *ExmNopenSide* is set to **XmLEFT**, then the left side of the **ExmTabButton** will be affixed to the right side of the manager widget. **XmNONE** specifies that the **ExmTabButton** will not be affixed to its parent. The parent of the **ExmTabButton** typically overrides the value of the *ExmNopenSide* resource.

Inherited Resources

TabButton inherits behavior and resources from the following superclasses. For a complete description of each resource, refer to the reference page for that superclass.

ExmCommandButton Resource Set				
Name	**Class**	**Type**	**Default**	**Access**
XmNactivateCallback	**XmCCallback**	**XtCallbackList**	*NULL*	C

ExmString Resource Set				
Name	**Class**	**Type**	**Default**	**Access**
ExmNcompound-String	**ExmCCompound-String**	**XmString**	*NULL*	CSG
XmNalignment	**XmCAlignment**	**unsigned char**	**XmALIGNMENT_-CENTER**	CSG

ExmTabButton(library call)

XmNrecomputeSize	**XmCRecomputeSize**	**Boolean**	*True*	CSG
XmNrenderTable	**XmCRenderTable**	**XmRenderTable**	*dynamic*	CSG

ExmSimple Resource Set				
Name	**Class**	**Type**	**Default**	**Access**
ExmNsimpleShape	**ExmCSimpleShape**	**unsigned char**	**ExmSHAPE_OVAL**	CSG
XmNmarginHeight	**XmCMarginHeight**	**Dimension**	4	CSG
XmNmarginWidth	**XmCMarginWidth**	**Dimension**	4	CSG

XmPrimitive Resource Set				
Name	**Class**	**Type**	**Default**	**Access**
XmNbottom-ShadowColor	**XmCBottom-ShadowColor**	**Pixel**	*dynamic*	CSG
XmNbottom-ShadowPixmap	**XmCBottom-ShadowPixmap**	**Pixmap**	**XmUNSPECIFIED_-PIXMAP**	CSG
XmNconvertCallback	**XmCCallback**	**XtCallbackList**	*NULL*	C
XmNforeground	**XmCForeground**	**Pixel**	*dynamic*	CSG
XmNhelpCallback	**XmCCallback**	**XtCallbackList**	*NULL*	C
XmNhighlightColor	**XmCHighlightColor**	**Pixel**	*dynamic*	CSG
XmNhighlightOnEnter	**XmCHighlightOnEnter**	**Boolean**	*False*	CSG
XmNhighlightPixmap	**XmCHighlightPixmap**	**Pixmap**	*dynamic*	CSG
XmNhighlight-Thickness	**XmCHighlight-Thickness**	**Dimension**	2	CSG
XmNlayoutDirection	**XmCLayout- Direction**	**XmDirection**	*dynamic*	CG
XmNnavigationType	**XmCNavigationType**	**XmNavigationType**	**XmNONE**	CSG
XmNpopup-HandlerCallback	**XmCCallback**	**XtCallbackList**	*NULL*	C
XmNshadow-Thickness	**XmCShadow-Thickness**	**Dimension**	2	CSG
XmNtopShadow-Color	**XmCTopShadow-Color**	**Pixel**	*dynamic*	CSG
XmNtopShadow-Pixmap	**XmCTopShadow-Pixmap**	**Pixmap**	*dynamic*	CSG

XmNtraversalOn	**XmCTraversalOn**	**Boolean**	*True*	CSG
XmNunitType	**XmCUnitType**	**unsigned char**	*dynamic*	CSG
XmNuserData	**XmCUserData**	**XtPointer**	*NULL*	CSG

Core Resource Set				
Name	**Class**	**Type**	**Default**	**Access**
XmNaccelerators	**XmCAccelerators**	**XtAccelerators**	*dynamic*	CSG
XmNancestor-Sensitive	**XmCSensitive**	**Boolean**	*dynamic*	G
XmNbackground	**XmCBackground**	**Pixel**	*dynamic*	CSG
XmNbackground-Pixmap	**XmCPixmap**	**Pixmap**	**XmUNSPECIFIED_-PIXMAP**	CSG
XmNborderColor	**XmCBorderColor**	**Pixel**	**XtDefaultForeground**	CSG
XmNborderPixmap	**XmCPixmap**	**Pixmap**	**XmUNSPECIFIED_-PIXMAP**	CSG
XmNborderWidth	**XmCBorderWidth**	**Dimension**	0	CSG
XmNcolormap	**XmCColormap**	**Colormap**	*dynamic*	CG
XmNdepth	**XmCDepth**	**int**	*dynamic*	CG
XmNdestroyCallback	**XmCCallback**	**XtCallbackList**	*NULL*	C
XmNheight	**XmCHeight**	**Dimension**	*dynamic*	CSG
XmNinitial-ResourcesPersistent	**XmCInitial-ResourcesPersistent**	**Boolean**	*True*	C
XmNmapped-WhenManaged	**XmCMapped-WhenManaged**	**Boolean**	*True*	CSG
XmNscreen	**XmCScreen**	**Screen ***	*dynamic*	CG
XmNsensitive	**XmCSensitive**	**Boolean**	*True*	CSG
XmNtranslations	**XmCTranslations**	**XtTranslations**	*dynamic*	CSG
XmNwidth	**XmCWidth**	**Dimension**	*dynamic*	CSG
XmNx	**XmCPosition**	**Position**	0	CSG
XmNy	**XmCPosition**	**Position**	0	CSG

Translations

ExmTabButton inherits all the translations of its superclass, **ExmCommandButton**. **ExmTabButton** does not provide any additional translations beyond those defined by **ExmCommandButton**.

Action Routines

ExmTabButton provides no action routines of its own.

Related Information

Core(3) and **XmPrimitive**(3).

Index